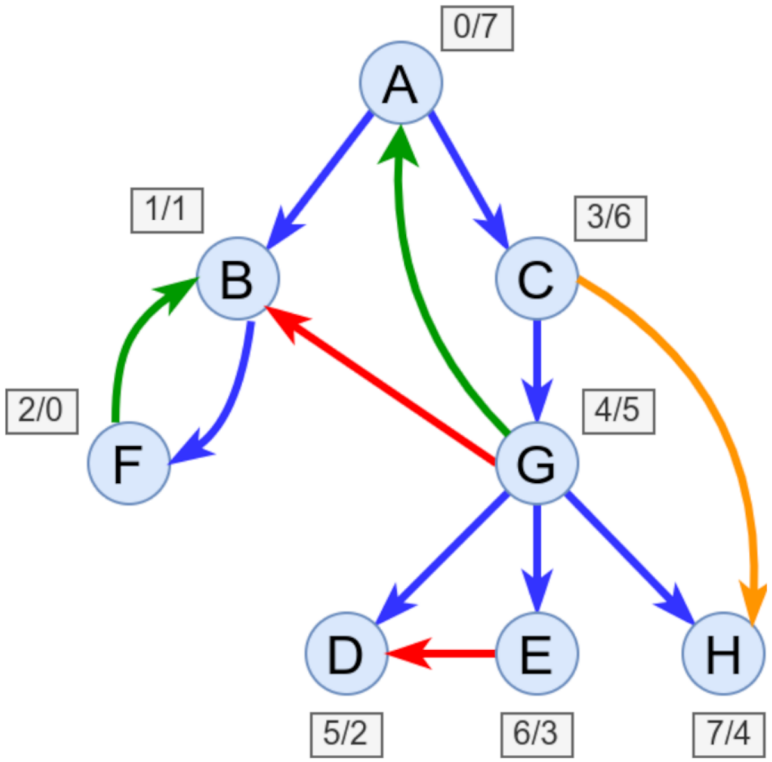
МАТФ
Универзитет у Београду
Математички факултет

ПРВО ИЗДАЊЕ



КОНСТРУКЦИЈА И АНАЛИЗА

АЛГОРИТАМА

ВЕСНА МАРИНКОВИЋ

ФИЛИП МАРИЋ

Весна Маринковић Филип Марић

Конструкција и анализа алгоритама

Математички факултет, Универзитет у Београду

Београд
2024.

Аутори:
др Весна Маринковић,
доцент на Математичком факултету у Београду
др Филип Марић,
редовни професор на Математичком факултету у Београду

КОНСТРУКЦИЈА И АНАЛИЗА АЛГОРИТАМА
Прво издање, 2024.

Издавач:
Математички факултет Универзитета у Београду
Студентски трг 16, 11000 Београд,
(+381) 011 2027 801, matf@matf.bg.ac.rs
За издавача: *проф. др Зоран Ракић*, декан

Рецензенти:
др Миодраг Живковић,
редовни професор на Математичком факултету у Београду, у пензији
др Предраг Јаничић,
редовни професор на Математичком факултету у Београду
др Драган Урошевић,
научни саветник на Математичком институту САНУ

Обрада текста, илустрације и корице: *аутори*
Штампа: Скрипта интернационал, Београд
Тираж: 150

Садржај

Предговор

Уџбеник пред вама је намењен студентима друге године Математичког факултета Универзитета у Београду и користи се за предмет „Конструкција и анализа алгоритама” на другој години студијског програма Информатика.

Претпоставља се да су студенти у ранијем школовању успешно савладали основне елементе програмирања и да су овладали основама алгоритмике (асимптотском анализом сложености и O-нотацијом, основним техникама конструкције и анализе алгоритама и основним структурама података).

Теоријски прегледи алгоритама и структура података дају информације на основу којих би читалац требало да буде у могућности да самостално креира имплементацију у програмском језику који одабере. Ипак, илустрације ради, алгоритми су имплементирани у савременом језику C++ уз интензивно коришћење стандардне библиотеке тог програмског језика. Знање овог језика и библиотеке се, стога, подразумева. Приказане су функције које имплементирају ове алгоритме, док је писање потпуних програма који користе ове функције препуштено читоацу. Комплетна решења су доступна у онлајн издању уџбеника.

Материјал изложен у овом уџбенику увелико превазилази оквире једног једносеместралног курса и на наставнику је да одабере подскуп тема које жели да обради у току курса, имајући у виду примене, потребе наредних курсева, али и предзнање студената. Неке теме се могу и оставити студентима за самостални рад.

Након сваког поглавља дато је неколико задатака који илуструју примене описаних техника. Ти задаци представљају само илустрацију, а студентима који желе да стекну боље

програмерске вештине се саветује да проради већи број задатака који се могу наћи у наменским збиркама задатака из области алгоритмике, али и на многим онлајн порталима посвећеним учењу програмирања (на пример, `petlja.org`, `codeforces.com`, `leetcode.com`, `geeksforgeeks.org` итд.).

Књига има и пратеће онлајн издање, доступно на адреси http://algoritmi.matf.bg.ac.rs/kiaa/index.html, у коме се налазе интерактивни аплети који могу помоћи студентима у разумевању неких тема.

Захвални смо професору Миодрагу Живковићу, који је кроз свој уџбеник и предавања утемељио приступ изучавању алгоритама на Математичком факултету. На веома пажљивом читању и коментарима и сугестијама које су допринеле квалитету књиге захваљујемо рецензентима Миодрагу Живковићу, Предрагу Јаничићу и Драгану Урошевићу и колегиници Тијани Шукиловић. Такође се захваљујемо сарадницима и студентима који су својим коментарима допринели исправљању уочених грешака и пропуста, посебно Матији Лојовићу, Лазару Јовановићу, Марку Цвијетиновићу, Жељку Зекавичићу и Милици Зубљић. Захвални смо и колеги Николи Ајзенхамеру на помоћи око графичког обликовања уџбеника.

Моле се читаоци да на све пропусте и евентуалне грешке укажу ауторима.

Аутори

1. Напредне структуре података

У овом поглављу приказана је имплементација неких напредних структура података. Претпостављамо да је читалац упознат са коришћењем и имплементацијом основних структура података: секвенцијалних структура података (низа, једноструко и двоструко повезане листе), стека, реда, реда са два краја, реда са приоритетом, као и основних асоцијативних структура података (скупа, мултискупа и мапе, тј. речника) коришћењем хеш-табела и балансираних уређених бинарних дрвета. За разлику од ових елементарнијих структура података, напредне структуре по правилу нису део стандардних библиотека програмских језика (на пример, нису укључене у библиотеке језика C++, C#, Python, Java) и потребно их је посебно имплементирати (или преузети неку јавно доступну имплементацију).

У овом поглављу ћемо проучити следеће структуре података.

- **Префиксно дрво** (енгл. trie) омогућава да се на још један начин имплементирају асоцијативне структуре података (поред уређених бинарних дрвета и хеш-табела), код којих се уметање и претрага врши на основу кључева који су обично или ниске карактера или ниске декадних или бинарних цифара. Ова дрвета се обично користе у применама обраде текста, као што су провера правописа и обрада природног језика, захваљујући својој могућности да ефикасно складиште велике речнике који садрже мноштво сличних речи, прецизније речи које деле заједничке префиксе.
- **Фибоначијев хип** представља још један начин да се имплементира ред са приоритетом (поред коришћења класичног бинарног хипа), код којег се, за разлику од класичног хипа, операције уметања, али и смањивање приоритета неког елемента

и спајања два хипа врше у амортизованом константном времену (подсетимо се, ове операције се код класичних хипова врше у логаритамском времену).

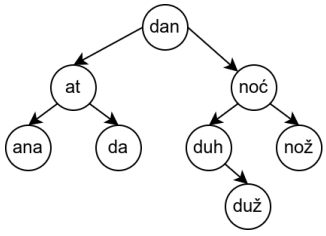
- **Структура за представљање дисјунктних скупова** (енгл. disjoint set union или union find) омогућава да се одржавају колекције дисјунктних скупова (подскупова неког скупа), уз могућност ефикасног проналажења скупа коме дати елемент припада и ефикасног спајања два скупа у један.
- **Структуре за ефикасно извршавање упита распона** (енгл. range queries) омогућавају да се након одређене предобраде (енгл. preprocessing) низа елемената ефикасно извршавају операције над његовим сегментима (поднизовима узастопних елемената), попут, на пример, израчунавања збира елемената сегмента, минимума или максимума елемената сегмента итд. Неке од ових структура допуштају ефикасно комбиновање ажурирања података (промену појединачних елемената или ажурирања целих сегмената увећањем свих елемената за неку вредност) и израчунавања поменутих статистика.

1.1 Префиксно дрво

Структура података са асоцијативним приступом (скуп, мапа тј. речник), код које се приступ елементима врши по кључу, ефикасно се имплементира коришћењем уређеног бинарног дрвета или хеш-табеле[1]. Кључ не мора бити целобројна вредност, већ ниска карактера, ниска битова или нешто друго.

Пример 1.1.1

На слици 1.1 приказано је уређено бинарно дрво[2] које садржи ниске `ana`, `at`, `noć`, `nož`, `da`, `dan`, `duh` *и* `duž` *као кључеве.*



Слика 1.1: Уређено бинарно дрво чији су кључеви ниске.

[1] Код асоцијативних структура података приступ елементима се врши на основу вредности кључа, а не на основу индекса, односно позиције елемента у структури података.

[2] Поред термина дрво (енгл. tree), често се користи и термин стабло.

Приликом свих операција са уређеним бинарним дрветом (претрага, брисање, уметање) у сваком чвору се врши поређење кључева (оног који је записан у чвору и оног који се обрађује) и када су кључеви ниске (али и неки други већи подаци), то поређење може захтевати доста времена и лоше утицати на перформансе.

Још једна структура података у виду дрвета која омогућава ефикасан асоцијативни приступ када су кључеви ниске је *префиксно дрво* (енгл. prefix tree), такође познато под енглеским називом *trie* (од енглеске речи *reTRIEval*). Префиксно дрво ефикасно решава следећи проблем.

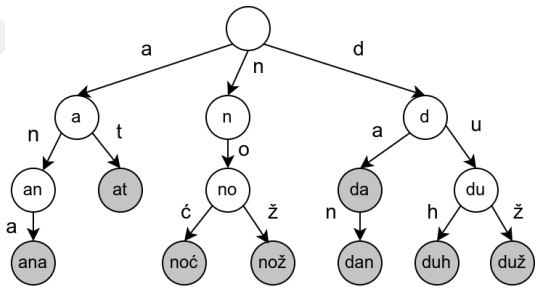
Проблем

Дефинисати асоцијативну структуру података која омогућава интерфејс скупа/мапе (додавање, тражење и брисање елемената) у којој су кључеви ниске или низови, а која подржава и ефикасно проналажење свих елемената чији кључеви имају задати префикс.

1.1.1 Структура префиксног дрвета

Основна идеја ове структуре података је да се кључ придружен чвору добија надовезивањем карактера који се налазе на гранама дуж путање од корена до тог чвора. Корен садржи празну реч, а преласком преко гране се на до тада формирану реч здесна надовезује још један карактер, па сваком чвору одговара неки префикс кључа. Притом, заједнички префикси различитих кључева су представљени истим путањама од корена до тачке разликовања (чвора који одговара најдужем заједничком префиксу). Чворови префиксног дрвета могу имати различит број деце, али максимални број деце одређен је величином азбуке која се користи за кодирање кључева. Ако се помоћу префиксног дрвета имплементира мапа (речник), тада се сваком кључу придружује вредност (податак који се чува у чвору дрвета којим се комплетира тај кључ).

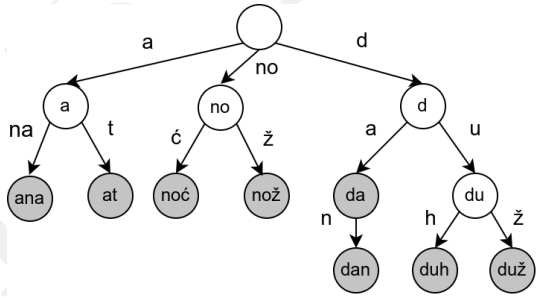
Пример 1.1.2



Слика 1.2: Пример префиксног дрвета.

Пример префиксног дрвета дат је на слици 1.2. Кључеви које ово префиксно дрво чува су исти они које чува уређено бинарно дрво са слике 1.1: ana, at, noć, nož, da, dan, duh *и* duž *(без придружених података). Већина кључева које ово префиксно дрво чува се завршава у неком од листова. Међутим, то у општем случају не мора да важи: кључ* da *се не завршава у листу. Стога је потребно да сваки чвор префиксног дрвета чува и информацију о томе да ли се њиме комплетира неки кључ или не (што се лако имплементира додавањем одговарајуће логичке променљиве у чвор дрвета). Чворови којима се комплетира неки кључ су обојени. Илустрације ради, у чворовима су приказане ознаке акумулиране до тих чворова. Треба имати у виду да се оне не чувају експлицитно у чвору.*

Мана префиксног дрвета може бити то што уз податке које чува заузима и пуно додатне меморије (за чување показивача) и стога је пожељно компримовати га. Ако неки чвор има само једног потомка и не представља крај неког кључа, грана до њега и грана од њега се могу спојити у једну, њихови карактери надовезати, а чвор елиминисати. На овај начин се добија компактнија репрезентација префиксног дрвета, позната и под називом *радикс дрво* (енгл. radix tree), код које сваки унутрашњи чвор има бар два детета (слика 1.3). За разлику од регуларног префиксног дрвета, гране радикс дрвета могу бити означене нискама, а не само појединачним карактерима. Предност радикс дрвета је пре свега у томе што су просторно ефикасније, посебно у случају када се у дрвету чувају дугачке ниске које имају дуге заједничке префиксе.



Слика 1.3: Пример компримованог префиксног дрвета, код кога су гране означене нискама.

Поред тога тога што се префиксним дрветом могу имплементирати опште асоцијативне структуре као скуп и мапа, префиксним дрветом може се имплементирати и коначни речник који омогућава аутоматско комплетирање или проверу исправности, односно аутоматско исправљање речи које корисник уноси на рачунару или мобилном телефону.

Кључеви у префиксном дрвету не морају бити искључиво ниске карактера. На пример, у префиксном дрвету можемо чувати и кључеве који су природни бројеви, при

чему се тада користи низ цифара у њиховом декадном или бинарном запису. Поред ниски, најчешће се користе бинарне репрезентације бројева фиксне ширине (записаних са фиксним бројем бинарних цифара).

1.1.2 Операције над префиксним дрветом

Операције тражења и уметања елемената у префиксно дрво врше се на прилично очигледан начин, док је у случају брисања некада потребно брисати више од једног чвора.

Испитивање да ли се нека реч налази у префиксном дрвету може се реализовати рекурзивном функцијом која као аргументе прима корен дрвета и реч коју тражи у дрвету (током рекурзивних позива у питању је корен одговарајућег поддрвета префиксног дрвета и неки суфикс речи која се претражује). Ако је реч празна, она се налази у дрвету ако и само ако је корен обележен као крај кључа. Ако реч није празна, да би се она могла налазити у дрвету потребно је да постоји дете корена до ког се стиже преко њеног првог слова и тада претрагу настављамо рекурзивно од тог чвора за суфикс речи без тог првог слова.

И операцију уметања речи у префиксно дрво можемо имплементирати као рекурзивну функцију која као аргументе добија прима дрвета и реч коју треба убацити у то дрво. Ако је реч празна, обележавамо да се у корену налази крај речи. У супротном проверавамо да ли постоји дете корена до ког се стиже првим словом те речи и ако не постоји додајемо га. Затим рекурзивно настављамо уметање од тог чвора и умећемо суфикс речи без тог првог слова.

Брисање речи из префиксног дрвета се такође може имплементирати рекурзивном функцијом која као аргументе прима корен дрвета и реч коју треба обрисати (током рекурзивних позива аргумент ће бити корен одговарајућег поддрвета префиксног дрвета и неки суфикс речи која се брише). Ако је реч празна, означавамо да текући чвор више није крај речи. Ако реч није празна, проверавамо да ли постоји дете корена до ког се стиже првим карактером те речи и ако постоји рекурзивно бришемо суфикс речи без првог слова из дрвета чији је корен то дете. На крају рекурзивне функције, проверавамо да ли текући чвор има деце и ако нема и није крај речи, бришемо и тај чвор из дрвета.

У наставку су дате имплементације основних операција над префиксним дрветом на примеру формирања и претраге скупа речи на енглеском језику. У овом примеру кључеви су речи (ниске карактера) и нису им придружене никакве вредности. Потребно је подржати операцију додавања кључева у структуру података и провере да ли кључ постоји у структури. Чвор префиксног дрвета може се дефинисати тако да садржи низ показивача, чијим елементима одговарају сви могући карактери азбуке из које се формирају кључеви (нпр. пошто се у овом примеру кодирају само речи које се састоје од

малих слова енглеске абецеде, можемо користити низ показивача дужине 26). Међутим, ефикасније је у сваком чвору чувати информације само о оним карактерима за које постоји грана из тог чвора, односно у ове сврхе искористити мапе. Дакле, у сваком чвору префиксног дрвета чувамо неуређену (хеш) мапу која карактерима придружује гране које крећу из тог чвора ка његовој деци, при чему користимо библиотечку имплементацију хеш-мапе (класу `unordered_map` декларисану у истоименом заглављу). Операције уметања и претраге се могу једноставно имплементирати било рекурзивно било итеративно (у наставку је приказана рекурзивна имплементација).

```
// osnovna struktura čvora prefiksnog drveta - u svakom čvoru čuvamo
// - mapu koja karakterima pridružuje pokazivače ka potomcima
// - informaciju o tome da li je u čvoru kraj neke reči
struct cvor {
  unordered_map<char, cvor*> grane;
  bool krajKljuca = false;
};

// sufiks koji počinje na poziciji i reči w tražimo
// u drvetu na čiji koren ukazuje pokazivač drvo
bool nadji(cvor *drvo, const string& w, int i) {
  // ako je sufiks prazan, on je u korenu akko je u korenu obeleženo
  // da je tu kraj reči
  if (i == w.size())
    return drvo->krajKljuca;

  // tražimo granu na kojoj piše w[i]
  auto it = drvo->grane.find(w[i]);
  // ako je nađemo, rekurzivno tražimo ostatak sufiksa od pozicije i+1
  if (it != drvo->grane.end())
    return nadji(it->second, w, i+1);

  // nismo našli granu sa w[i], pa reč ne postoji
  return false;
}

// tražimo reč w u drvetu na čiji koren ukazuje pokazivač drvo
bool nadji(cvor *drvo, const string& w) {
  return nadji(drvo, w, 0);
}
```

```
// umetanje sufiksa koji počinje na poziciji i reči w
// u drvo na čiji koren ukazuje pokazivač drvo
void umetni(cvor *drvo, const string& w, int i) {
  // ako je sufiks prazan samo u korenu beležimo da je tu kraj reči
  if (i == w.size()) {
    drvo->krajKljuca = true;
    return;
  }

  // tražimo granu na kojoj piše w[i]
  auto it = drvo->grane.find(w[i]);
  // ako takva grana ne postoji, dodajemo je kreirajući novi čvor
  if (it == drvo->grane.end())
    drvo->grane[w[i]] = new cvor();

  // sada znamo da grana sa w[i] sigurno postoji i preko te grane
  // nastavljamo dodavanje sufiksa koji počinje na poziciji i+1
  umetni(drvo->grane[w[i]], w, i+1);
}

// umetanje reči w u drvo na čiji koren ukazuje pokazivač drvo
void umetni(cvor *drvo, string& w) {
  return umetni(drvo, w, 0);
}

// brisanje drveta sa korenom u čvoru drvo
void obrisi(cvor *drvo) {
  if (drvo != nullptr) {
    for (const auto& p : drvo->grane)
      obrisi(p.second);
    delete drvo;
  }
}
```

Када азбука којом се кодирају кључеви има K елемената и када се у сваком чвору чува неуређена мапа грана, сложеност операција претраживања, уметања и брисања елемента из префиксног дрвета је у најгорем случају $O(mK)$, где је m дужина речи која се тражи, умеће или брише. Заиста, сложеност најгорег случаја претраге неуређене мапе је $O(K)$, а приликом обраде кључа дужине m врши се m таквих претрага.

Са друге стране, амортизована сложеност претраге неуређене мапе је $O(1)$, па је амортизована сложеност ових операција $O(m)$. Ако се ради са нискама карактера и ако се неуређена мапа имплементира помоћу низа од K елемената, онда је и сложеност најгорег случаја $O(m)$. Приметимо да је приликом претраге број операција ограничен и дужинама речи које се налазе у дрвету, па се, прецизније, сложеност може ограничити и са $O(\min(m, M))$, где је M дужина најдуже речи која се налази у дрвету.

Добра страна префиксног дрвета је то што сложеност уметања и претраге зависи од дужине записа кључа, а не од броја елемената који се чувају у дрвету. Мана је потреба за чувањем показивача уз сваки чвор у дрвету. Штавише, просторна сложеност префиксног дрвета у најгорем случају износи $O(MNK)$, где је са N означен број кључева који се чувају у префиксном дрвету, а са M максимална дужина кључа. Наиме, максимални могући број чворова префиксног дрвета једнак је $O(MN)$ и одговара случају када нема никаквог преклапања карактера међу кључевима, док је просторна сложеност сваког чвора једнака $O(K)$ због потребе чувања мапе у сваком чвору. Приметимо да је очекивана просторна сложеност мања јер ће се у случају реалне коначне азбуке (на пример, енглеске абецеде) прво преклапање јавити најкасније након 26 кључева.

Смањење меморијске сложености се може постићи и тако што се смањи величина азбуке (по цену повећања дужине кључа). На пример, уместо 256 различитих 8-битних карактера, можемо сваки карактер поделити на две 4-битне половине. Тако се само 16 различитих 4-битних карактера, али се дужина сваког кључа два пута повећава.

Када би се уместо префиксног дрвета користило балансирано уређено бинарно дрво које би чувало комплетне кључеве у чворовима (слика 1.1), временска сложеност операција претраживања, уметања и брисања би у најгорем случају била $O(M \cdot \log N)$, где је са N означен укупан број кључева који се чувају у дрвету, а са M максимална дужина кључа. Просторна сложеност ове структуре је $O(M \cdot N)$.

Када се користи хеш табела, приликом сваке операције уметања и претраге мора да буде израчуната хеш вредност кључа који се тражи за шта је потребно време $O(M)$. У зависности од броја колизија, врше се поређења хеш вредности (у најгорем случају њих $O(N)$, а амортизовано $O(1)$). На крају се кључ који се тражи и кључ слога који је пронађен на основу једнакости хеш-вредности експлицитно пореде, за шта је такође потребно време $O(M)$ (а ако постоје колизије ово поређење се врши неколико пута). Зато је сложеност најгорег случаја операција $O(N + M)$, а амортизована сложеност $O(M)$. Пошто кључ мора бити записан експлицитно у сваком слогу табеле, просторна сложеност је $O(M \cdot N)$.

У табели 1.1 приказана је (амортизована) сложеност разних имплементација скупова/мапа. Претпостављамо да је дужина речи која се обрађује m, максимална дужина речи M, број речи у колекцији N, а величина азбуке K.

Табела 1.1: Сложеност различитих имплементација речника

	Хеш табела	Балансирано бинарно уређено дрво	Префиксно дрво
Претрага	$O(m)$	$O(m \log N)$	$O(m)$
Уметање	$O(m)$	$O(m \log N)$	$O(m)$
Брисање	$O(m)$	$O(m \log N)$	$O(m)$
Простор	$O(MN)$	$O(MN)$	$O(MNK)$

Дакле, можемо приметити да префиксно дрво нема лошију сложеност од хеш-табела, али подржава нову операцију (проналажење свих речи које имају дати префикс) и стога се користи приликом решавања проблема у којима је та операција корисна.

Задатак: Пар који даје највећи XOR

Напиши програм који међу унетим неозначеним бројевима одређује онај пар који даје највећи резултат при операцији ексклузивне дисјункције (XOR) њихових бинарних записа.

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 100000$), а затим у наредном реду n природних бројева између 0 и 10^{18}.

Опис излаза

На стандардни излаз исписати максималну вредност која се може добити када се ексклузивна дисјункција примени на нека два унета броја.

Пример

Улаз	*Излаз*	*Објашњење*
`5`	`7`	Највећи резултат 7 добија се ексклузивном дисјункцијом бројева
`1 2 3 4 5`		3 и 4 (њихови бинарни записи су `00...0000011` и `000...000100`). Исти резултат добија се и ексклузивном дисјункцијом бројева 2 и 5 (њихови бинарни записи су `0000..0000101` и `0000...00000010`).

Решење

Решење грубом силом подразумева да се на сваки пар бројева примени операција XOR и да се испише максимум добијених резултата. Сложеност овог приступа је $O(n^2)$.

Ефикасније решење можемо добити применом напредних структура података. Покушајмо да за сваки нови унети број ефикасно израчунамо највећи број који се може добити применом операције XOR на њега и неки од претходно унетих бројева (у решењу

грубом силом, то се дешава у унутрашњој петљи). Покушајмо да тај број одредимо бит-по-бит и то кренувши од битова највеће тежине. На месту бита највеће тежине можемо добити 1 ако текући број почиње битом 0 и међу раније учитаним бројевима постоји неки који почиње битом 1 или ако текући број почиње битом 1 и међу раније учитаним бројевима постоји неки који почиње битом 0. У супротном, на месту највеће тежине резултата мора бити бит 0. Након одређивања првог бита, одређујемо наредни, али у случају да смо на водеће место резултата уписали 1, међу учитаним нискама задржавамо само оне које су на водећем месту имали бит супротан водећем биту текућег броја (у случају да смо на водеће место резултата уписали 0, тада су сви раније учитани бројеви почињали истим битом којим почиње текући број и сви се задржавају). Поступак сада понављамо за други бит, при чему разматрамо само ниске које нису раније одбачене.

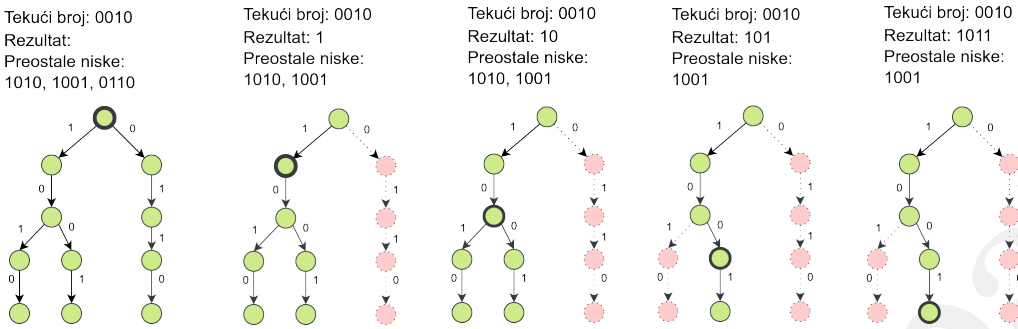
Пример 1.1.3

Претпоставимо да су дати бројеви `1001`, `1010`, `0110` и да је текући број `0010`.

- *На водеће место резултата можемо уписати `1`, при чему задржавамо ниске `1001`, `1010`.*
- *На друго место резултата морамо уписати `0`, јер све задржане ниске на месту другог бита имају `0`, исто као и текући број. Обе ниске се задржавају.*
- *На треће место резултата можемо уписати `1` и тада задржавамо само ниску `1001`.*
- *На крају, на последње место резултата можемо уписати `1`.*

Коначан резултат је, дакле, `1011` и он се добија применом операције XOR на ниске `0010` и `1001`.

Претрагу можемо веома једноставно организовати ако све учитане бинарне записе један по један уписујемо у префиксно дрво. Приликом обраде текућег броја, његове битове пролазимо слева надесно, за сваки бит се спуштајући на наредни ниво дрвета. На сваком нивоу дрвета, дакле, одређујемо један бит резултата. Ако на текућем нивоу постоји грана обележена битом супротном од текућег бита тренутног броја, спуштамо се њоме и у резултат уписујемо јединицу, док се у супротном спуштамо граном на којој се налази бит једнак текућем биту тренутног броја и у резултат уписујемо нулу. Спуштањем на наредни ниво дрвета уједно елиминишемо све оне ниске које на одговарајућем месту немају потребан бит. Овај поступак је илустрован на наредној слици.



Сложеност овог алгоритма је $O(n)$, при чему је константа једнака броју битова којима се записују бројеви (с обзиром на ограничења дата у задатку, то је 64).

```
// cvor binarnog prefiksnog drveta
struct Cvor {
  Cvor* grane[2];
};

// kreiranje novog cvora drveta
Cvor* noviCvor() {
  Cvor* novi = new Cvor();
  novi->grane[0] = novi->grane[1] = nullptr;
  return novi;
}

// ubacivanje broja (njegov binarnog zapisa) u drvo
void ubaci(Cvor* koren, unsigned long long broj) {
  Cvor* cvor = koren;
  unsigned long long mask = 1ull << (8*sizeof(unsigned long long) - 1);
  while (mask != 0) {
    int bit = (broj & mask) != 0;
    if (cvor->grane[bit] == nullptr)
      cvor->grane[bit] = noviCvor();
    cvor = cvor->grane[bit];
    mask >>= 1;
  }
}
```

```
// brisanje drveta
void obrisi(Cvor* koren) {
  if (koren != nullptr) {
    obrisi(koren->grane[0]);
    obrisi(koren->grane[1]);
    delete koren;
  }
}

// maxXOR koji moze da se napravi izmedju elemenata drveta i datog broja
unsigned long long maxXOR(Cvor* koren, ull broj) {
  Cvor* cvor = koren;
  unsigned long long rez = 0;
  // analiziramo sve bitove datog broja
  // maska sadrzi tacno jednu binarnu jedinicu i ona se pomera nalevo
  unsigned long long mask = 1ull << (8*sizeof(unsigned long long) - 1);
  while (mask != 0) {
    // tekuci bit datog broja
    int bit = (broj & mask) != 0;
    if (cvor->grane[!bit] != nullptr) {
      rez = rez | mask;
      cvor = cvor->grane[!bit];
    } else
      cvor = cvor->grane[bit];
    mask >>= 1;
  }
  return rez;
}

int main() {
  // broj brojeva
  int n;
  cin >> n;
  // prefiksno drvo
  Cvor* koren = noviCvor();

  // ubacujemo prvi broj
  unsigned long long x;
  cin >> x;
```

```
  ubaci(koren, x);
  unsigned long long max = 0;
  // analiziramo naredne brojeve
  for (int i = 1; i < n; i++) {
    cin >> x;
    // trazimo maksimalni XOR koji se moze dobiti od broja x
    // i brojeva koji se do sada nalaze u drvetu
    unsigned long long rez = maxXOR(koren, x);
    if (rez > max)
      max = rez;
    ubaci(koren, x);
  }
  cout << max << endl;

  // brisemo drvo
  obrisi(koren);
  return 0;
}
```

1.2 Фибоначијев хип

Редови са приоритетом се обично имплементирају коришћењем структуре података хип. За разлику од бинарног хипа у ком операција уметања елемента има логаритамску сложеност, Фибоначијев[3] хип је структура података код које операција уметања новог елемента има константну амортизовану сложеност, што је чини бржом. Додатно, Фибоначијев хип омогућава и спајање два хипа у један као и смањивање вредности кључа у константној амортизованој сложености, док сложеност избацивања минимума остаје логаритамска. Дакле, Фибоначијев хип успешно решава следећи проблем.

Проблем

Дефинисати структуру података која подржава ефикасно извршавање (у амортизованом константном или логаритамском времену) следећих операција:

- `napravi_hip()` - *прави нови празни хип*
- `umetni(H, v)` - *умеће елемент са вредношћу кључа* `v` *у хип* `H`
- `minimum(H)` - *враћа елемент хипа* `H` *са минималном вредношћу кључа*

[3] Леонардо Пизано Фибоначи (ит. Leonardo Bonacci), (око 1170-1240), италијански математичар.

- `izbaci_minimum(H)` - *брише елемент хипа* `H` *са минималном вредношћу кључа и враћа га као резултат*
- `unija(H1,H2)` - *од два хипа* `H1` *и* `H2` *прави нови хип који садржи све елементе полазних хипова*
- `smanji_kljuc(H, x, v)` - *елементу* `x` *хипа* `H` *се тренутна вредност кључа смањује на нову дату вредност* `v`

Имплементација операције `smanji_kljuc` захтева памћење додатних података у структури, па је имплементација хипа који подржава ту операцију мало компликованија него имплементација хипа који подржава све остале наведене операције.

Амортизована сложеност операција прављења хипа, уметања елемента, одређивања минимума, прављења уније и смањивања вредности кључа је $O(1)$, док је амортизована сложеност операције брисања минималног елемента из хипа $O(\log n)$. Дакле, операције уметања елемента и прављења уније се ефикасније извршавају над Фибоначијевим хипом, него над класичним бинарним хипом (подсетимо се, сложеност додавања елемената у бинарни хип је $O(\log n)$).

Фибоначијев хип је користан када је број операција брисања минималног елемента из хипа мали у односу на остале поменуте операције. На пример, у графовским алгоритмима као што је одређивање разапињућег (повезујућег) дрвета минималне цене (види поглавље 2.10) или најкраћих путева из задатог чвора (види поглавље 2.9), ако је граф густ (садржи велики број грана), операција смањивања вредности кључа се може често јављати, те убрзање са $O(\log n)$ код бинарног хипа на $O(1)$ код Фибоначијевог хипа може донети осетно убрзање. Недостатак Фибоначијевог хипа је то што је доста тежи за имплементацију од класичног, бинарног хипа, то што захтева више меморије, као и то што је сложеност најгорег случаја неких операција велика.

Фибоначијев хип је добио назив према Фибоначијевим бројевима на основу којих се формулише централна инваријанта која гарантује добру сложеност операција. Ову структуру података осмислили су Мајкл Фредман[4] и Роберт Тарџан[5] са циљем да се побољша време извршавања Дајкстриног алгоритма за одређивање најкраћих путева из задатог чвора (види поглавље 2.9.2). Она има примене и у другим графовским алгоритмима, попут Примовог алгоритма за рачунање минималног повезујућег дрвета (види поглавље 2.10.2), за одређивање максималног тока кроз мрежу, али и у другим доменима, попут геометријских алгоритама, обраде слика и математичке оптимизације.

[4] Мајкл Фредман (енгл. Michael Fredman), амерички информатичар.
[5] Роберт Тарџан (енгл. Robert Tarjan), рођен 1948. године, амерички информатичар.

1.2.1 Структура хипа

Фибоначијев мин-хип представља колекцију мин-хипова. Сваки мих-хип је дрво у ком чворови могу имати различит број наследника, а вредност у сваком чвору је мања или једнака вредности у његовим наследницима. Редослед дрвета у Фибоначијевом хипу је произвољан. Фибоначијевом хипу приступамо путем показивача на корен дрвета са минималном вредношћу у целом Фибоначијевом хипу – овај чвор зовемо *минималним чвором* Фибоначијевог хипа (ако постоји више чворова са минималном вредношћу било који од њих се може прогласити минималним чвором). Један пример Фибоначијевог мин-хипа приказан је на слици 1.5. Корени свих дрвета у Фибоначијевом хипу се повезују у кружну, двоструко повезану листу коју називамо *листа коренова* хипа. И сва деца било ког чвора су међусобно повезана у кружну, двоструко повезану листу. Сваки чвор (било да је корен или не) садржи показивач на левог и десног суседа и на неко дете (а ако је потребно вршити и операције смањивања вредности кључева, онда и показивач на родитеља и још неке помоћне податке које ћемо описати у склопу описа те операције). Коришћење кружних листа омогућава уметање новог чвора у листу и брисање чвора на који имамо показивач (самим тим и минималног чвора) у времену $O(1)$. Такође, две овакве листе можемо објединити у нову листу у времену $O(1)$ (видећемо да је ово битно за ефикасно извођење операције формирања уније). Ефикасност ових операција над листама гарантује ефикасно извршавање операција над хипом.

Број деце чвора назива се *степен чвора*. Наведимо сада кључну инваријанту (услов који важи након формирања структуре и након примене било које операције и за који ћемо видети да гарантује сложеност операција).

Инваријанта (број чворова поддрвета): Свако поддрво Фибоначијевог хипа чији је корен степена d садржи бар F_{d+2} чворова, где је са F_k означен k-ти Фибоначијев број ($F_0 = 0$, $F_1 = 1$, $F_k = F_{k-1} + F_{k-2}$).

Дакле, дрво чији је корен степена 0 има бар 1 чвор, степена 1 има бар 2 чвора, степена 2 има бар 3 чвора, степена 3 има бар 5 чворова, степена 4 има бар 8 чворова итд.. Пошто се лако (индукцијом) доказује да је $F_{d+2} \geq \varphi^d$, где је $\varphi = \frac{1+\sqrt{5}}{2}$, за степен d сваког чвора који је корен неког поддрвета у ком се налази m чворова важи $m \geq F_{d+2} \geq \varphi^d$ тј. $d \leq \log_{\varphi} m$. Дакле, број деце сваког чвора који је корен поддрвета са m елемената је $O(\log m)$. Ако у целом Фибоначијевом хипу има n елемената, степен било ког чвора је $O(\log n)$. Дакле, под условом да инваријанта важи, листе деце су релативно кратке и пролазак кроз све елементе такве листе врши се у логаритамској сложености (у односу на укупан број елемената у хипу).

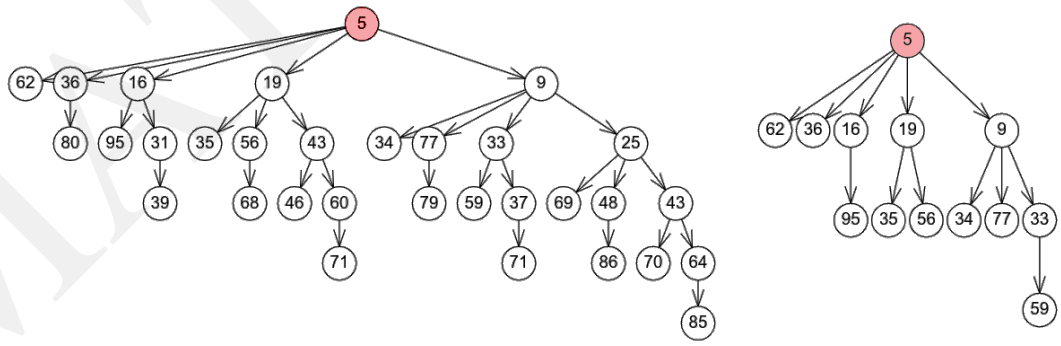
Формулишимо сада другачију инваријанту, коју је лакше проверавати, а која даје гаранције да ће важити инваријанта о броју чворова у поддрветима.

Инваријанта (степени деце): За сваки чвор x степена d важи да су степени деце бар $0, 0, 1, 2, 3, \ldots, d-2$.

Индукцијом доказујемо да инваријанта о степенима деце гарантује инваријанту о броју чворова поддрвета. Ако је $d = 0$, чвор нема деце, а у његовом поддрвету се налази $F_2 = 1$ чвор. Ако је инваријанта о степенима деце испуњена, на основу индуктивне хипотезе важи да поддрвета чији су корени деца чвора x садрже редом $F_2, F_2, F_3, \ldots F_d$ чворова, па, пошто је $F_2 = F_1 = 1$ и $F_0 = 0$, важи да је укупан број чворова у дрвету чији је корен x једнак $1 + F_0 + F_1 + F_2 + F_3 \ldots + F_d$ (прва јединица долази од самог корена x). Индукцијом се рутински може доказати да је $1 + F_0 + F_1 + F_2 + \ldots + F_d = F_{d+2}$, па инваријанта о степенима деце заиста гарантује инваријанту о броју чворова поддрвета.

Ова инваријанта намеће одређене степене деце, али није прецизирано која деца треба да имају тражене степене. Деца се разматрају у редоследу њиховог додавања родитељском чвору и њихови степени у том редоследу задовољавају инваријанту (а не по редоследу којим су смештени у листу деце). Дакле, услов који је довољан да би хип био Фибоначијев је да су степени деце у редоследу додавања $0, 0, 1, 2, 3, \ldots$ тј. да за сваки чвор важи да његово дете са редним бројем додавања i (бројимо од нуле) има степен $d_i \geq i - 1$ (при чему је и $d_i \geq 0$, јер је d_i природан број).

Видећемо ускоро да ће процедура формирања хипа бити таква да су степени деце сваког чвора дрвета непосредно након његовог првог формирања $0, 1, 2, 3, \ldots$ (што је и више него што се захтева инваријантом о степенима деце). На основу овога се може доказати да свако поддрво чији је степен корена d има 2^d чворова, што такође гарантује повољну сложеност операција. Ипак, инваријанта је ослабљена да би било могуће уклањати чворове из формираног дрвета, што је, видећемо, веома важно за реализацију операције смањивања вредности кључа (`smanji_kljuc`).



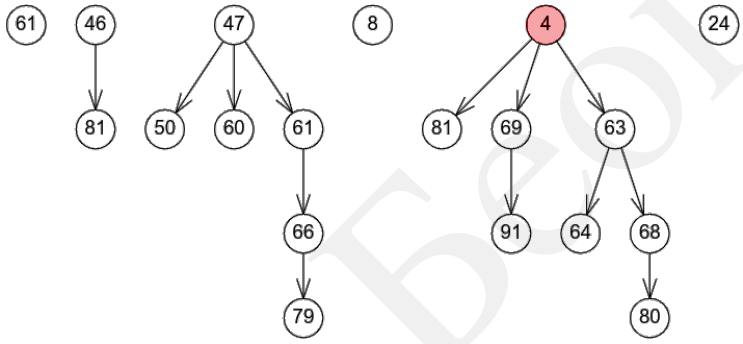
Слика 1.4: Дрво у ком има $2^5 = 32$ чвора (степени чворова су редом, $0, 1, 2, 3, \ldots$). Дрво у ком има $F_7 = 13$ чворова (степени чворова су редом $0, 0, 1, 2, 3, \ldots$).

На слици 1.4 лево приказано је најгушће могуће попуњено дрво (које се добија одмах након формирања дрвета које садржи $2^5 = 32$ чвора), док се десно види најређе могуће дрво које задовољава инваријанту и које садржи $F_{5+2} = 13$ чворова. Избацивањем било ког чвора из десног дрвета била би нарушена инваријанта.

Прикажимо, напокон, један комплетан пример Фибоначијевог хипа.

Пример 1.2.1

На слици 1.5 приказан је Фибоначијев хип који се састоји од 6 *мин-хипова. Минимални елемент у хипу је минимум свих коренова, а то је елемент са кључем* 4.



Слика 1.5: Пример Фибоначијевог хипа.

Директном провером се лако може показати инваријанта о степенима деце. Она је увек тривијално испуњена за чворове чији је степен 0, 1 *и* 2 *(јер су степени деце увек већи или једнаки од 0 што се инваријантом тражи). Деца чвора* 47 *имају редом степене* 0, 0 *и* 1, *а деца чвора* 4 *имају редом редом степене* 0, 1 *и* 2. *Дакле, за сваки чвор јесте задовољена инваријанта о степенима деце (степени деце су на неким местима већи него што је потребно), а она гарантује и инваријанту о величинама тј. укупном броју чворова тих дрвета. Заиста, дрвета која чине Фибоначијев хип редом имају* 1, 2, 6, 1, 8, 1 *и* 1 *чвор, што је понекад и више од одговарајућих Фибоначијевих бројева* 1, 2, 5, 1, 5, 1 *и* 1. *Исто важи и за сва њихова поддрвета, па је централна инваријанта о величинама свих дрвета задовољена.*

1.2.2 Операције над хипом

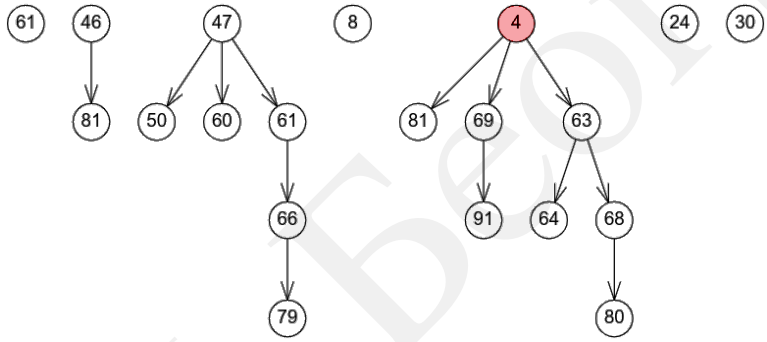
Једна од основних карактеристика операција над Фибоначијевим хипом је лењо извршавање операција, тј. посао се одлаже за што је могуће касније.

1.2.2.1 Уметање

Уметање новог елемента у Фибоначијев хип (операција `insert(H, x)`) се врши тако што се нови елемент додаје у листу коренова, ажурирајући минимални чвор, ако је потребно, што је константне временске сложености. Дакле, уметањем k елемената у празан Фибоначијев хип добија се Фибоначијев хип чија листа коренова има k елемената.

Пример 1.2.2

Уметањем елемента са кључем 30 *у претходни Фибоначијев хип добија се хип приказан на слици* 1.6 *(нови елемент је додат на крај листе коренова, али могао је бити додат и на било које друго место, најчешће је то непосредно иза или испред минималног чвора, на који, подсетимо се, указује посебан показивач).*



Слика 1.6: Уметање елемента 30.

1.2.2.2 Унија два хипа

Унија два Фибоначијева хипа H_1 и H_2 (операција `unija(H1, H2)`) се прави надовезивањем листи коренова ова два хипа и одређивањем новог минималног чвора (то је мањи од два минимална чвора полазних хипова). И ово је операција константне сложености.

1.2.2.3 Одређивање минимума

Одређивање минималног елемента у Фибоначијевом хипу (операција `minimum(H)`) је тривијално, с обзиром на то да се у сваком тренутку чува показивач на минимални елемент у структури. Ово је такође операција константне сложености.

1.2.2.4 Брисање најмањег елемента

Операција `izbaci_minimum(H)`, којом се брише најмањи елемент из Фибоначијевог хипа почиње тако што се деца минималног елемента умећу у листу коренова, и он се брише из листе коренова. Да би се ажурирао минимум хипа потребно је потенцијално проћи кроз целу листу коренова, која може бити веома дуга. Зато се пре ажурирања минимума

врши помоћна операција која се назива *консолидација* хипа. Њоме се хип доводи у стање у ком сви коренови имају различит степен, чиме се њихов број значајно смањује (на основу раније показане везе између степена коренова и величине дрвета, јасно је да су сви степени коренова $O(\log n)$, па их не може бити више од $O(\log n)$).

Консолидација се врши на следећи начин: све док нека два чвора из листе коренова имају исти степен врши се спајање коренова истог степена тј. ради се следеће:

1. проналазе се два чвора x и y у листи коренова која су истог степена (без смањења општости нека важи да је вредност кључа чвора x мања или једнака од вредности кључа чвора y)
2. Чвор y се избацује из листе коренова и проглашава се дететом чвора x. На овај начин, дрво са кореном x остаје хип и степен чвора x се повећава за један. На основу инваријанте о степенима деце знамо да важи да је последње додати наследник чвора x имао бар степен $d-2$, где је d степен чвора x. То значи да би наредни тј. последњи додати наследник, што је управо y, морао да има степен бар $d-1$. Међутим, ми знамо да он има степен d (јер му је степен једнак степену чвора x), тако да је инваријанта задовољена, уз један додатни степен слободе. Наиме, чак и када уклонимо једног наследника чвора y, инваријанта остаје задовољена. Са друге стране, уклањање два наследника би нарушило инваријанту и то не смемо да радимо. Ово је значајно за операцију `smanji_kljuc`, којом се смањује вредности кључа елемената хипа и вратићемо се на ово приликом описа те операције.

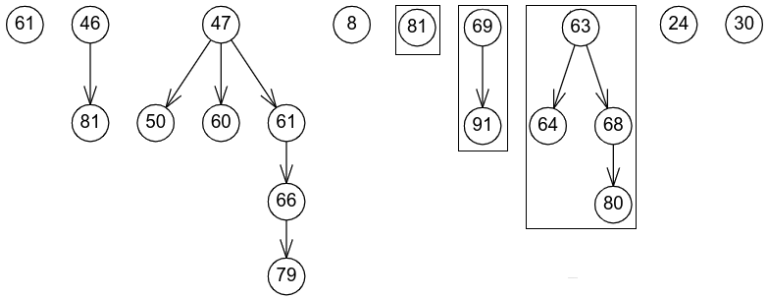
Нагласимо да је избор пара чворова са истим степенима у сваком кораку произвољан (консолидација се не мора вршити ни у каквом посебном редоследу парова чворова).

Обично се за реализовање консолидације користи помоћни низ у који се смештају показивачи на чворове из листе коренова, тако да се на позицији i чува показивач на неки чвор из листе коренова који је степена i. Димензија овог помоћног низа једнака је максималном степену $D(n)$ произвољног чвора у Фибоначијевом хипу – већ смо објаснили да је он једнак $O(\log n)$, где је са n означен број елемената хипа. Коренови који су истог степена се детектују на следећи начин: пролази се редом кроз листу коренова и ако је текући корен степена i, а i-ти елемент низа је још увек празан, он се иницијализује показивачем на дати корен, а ако је i-ти елемент већ постављен, пронађена су два корена истог степена и они се спајају.

Пример 1.2.3

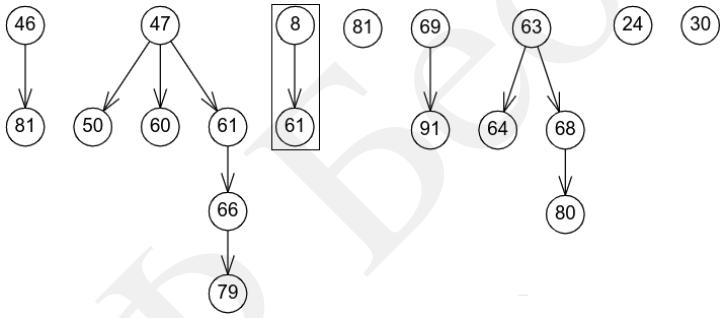
Посматрајмо како се мења претходни хип након извођења операције брисања елемента са минималном вредношћу кључа.

У првом кораку се сва деца уклоњеног минималног чвора 4 *умећу у низ коренова.*

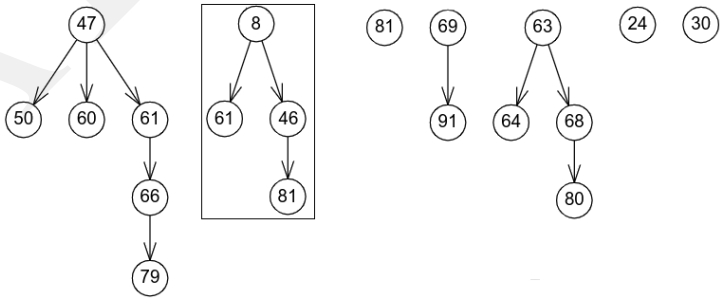


Након тога извршавамо консолидацију (спајање дрвета чији су корени истог степена).

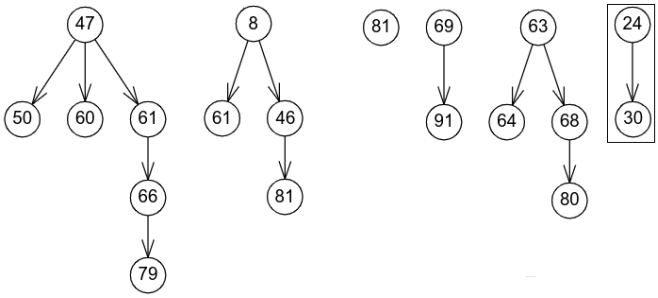
- *Прво спајамо чворове* 8 *и* 61*, који имају степен* 0*.*



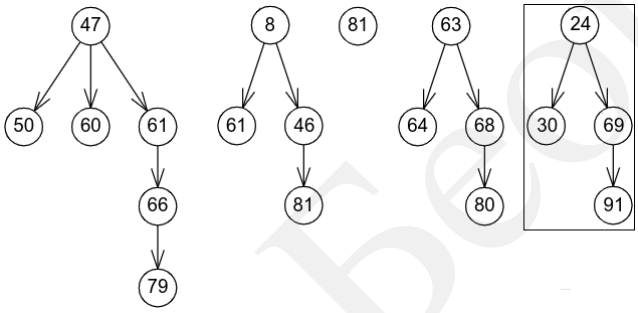
- *Корен добијеног дрвета има степен* 1*, оно се спаја са дрветом чији је корен* 46*, који такође има степен* 1*.*



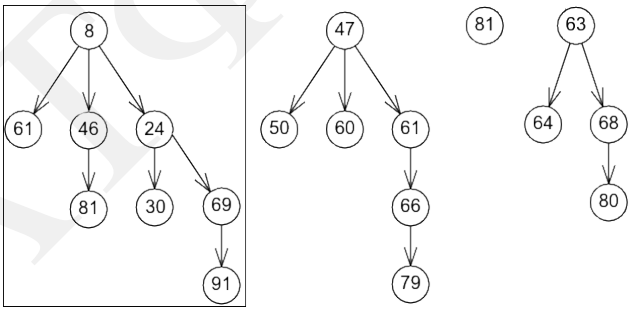
- *Након тога спајају се дрвета чији су корени* 24 *и* 30 *(оба су степена* 0*).*



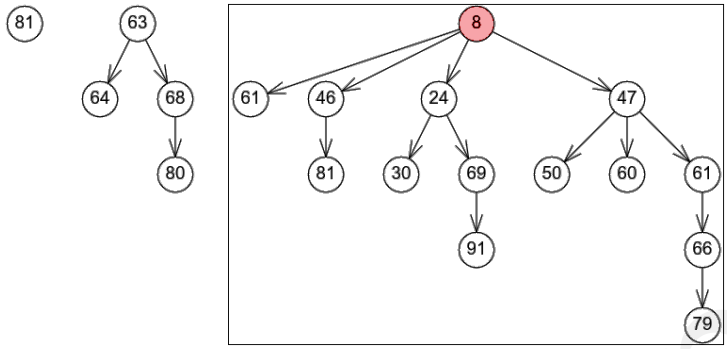
- *Добијено дрво се спаја са дрветом чији је корен* 69 *(оба корена имају степен* 1*).*



- *Добијено дрво се спаја са дрветом чији је корен* 8 *(оба дрвета имају степен* 2*).*



- *На крају се спаја добијено дрво са дрветом чији је корен* 47 *(оба корена имају степен* 3*). Након тог спајања, сва дрвета имају различите степене коренова (степени су* 0*,* 2 *и* 4*). Тиме се консолидација завршава, гарантовано је да је број дрвета* $O(\log n)$ *и затим се нови минимални чвор може ефикасно одредити (то је чвор* 8*).*



Анализирајмо сложеност претходно описаног поступка. Деца се ефикасно, алгоритмом сложености $O(1)$ могу уметнути у листу коренова (ради се о спајању две листе). Након тога се врши консолидација, која у најгорем случају може бити и линеарне сложености (најгори случај је када се n једночланих дрвета спаја). Ипак, може се показати да је амортизована сложеност консолидације $O(\log n)$ – неформално, када се једном изврши консолидација и добије се мали број великих дрвета, сваки наредни корак избацивања најмањег елемента и консолидације биће прилично ефикасан. На крају се проналази нови минимални чвор, обрадом свих коренова. Њих након консолидације може бити највише $O(\log n)$, па је ово ефикасна операција.

Прикажимо сада једну једноставну имплементацију ових операција (једноставности ради, користимо глобалне променљиве и не водимо рачуна о успешности алокације, као ни о деалокацији меморије).

```
// Cvor drveta sadrzi podatak i dvostruko povezanu listu dece
struct Cvor {
  int podatak;
  list<Cvor*> deca;
};

// dvostruko povezana lista korenova
list<Cvor*> hip;
// pokazivac (iterator) koji ukazuje na najmanji cvor
list<Cvor*>::iterator minCvor;

// umetanje novog elementa u hip
void umetni(int podatak) {
  // pravimo novi cvor i upisujemo podatak u njega
  Cvor* novi = new Cvor();
```

```
  novi->podatak = podatak;

  if (hip.empty()) {
    // ako je hip prazan ubacujemo novi cvor u njega i taj novi cvor
    // je minimalan
    hip.push_back(novi);
    minCvor = hip.begin();
  } else {
    // ako hip nije prazana ubacujemo novi cvor u njega
    hip.push_front(novi);
    // minimum azuriramo samo ako je novi podatak manji od dotadasnjeg
    // minimalnog
    if (novi->podatak < (*minCvor)->podatak)
      minCvor = hip.begin();
  }
}

// najmanji element u hipu
int minimum() {
  return (*minCvor)->podatak;
}

// konsolidacija hipa (pomocna operacija kojom se skracuje lista korenova)
void konsoliduj() {
  // za svaki moguci broj dece pamtimo jedan koren koji ima toliko dece
  // u njemu ce se nalaziti svi koreni novog hipa (i svi koreni
  // ce imati razlicit broj dece)
  const int MAKS_DECE = 64;
  Cvor* pom[MAKS_DECE] = {};
  // obradjujemo jedan po jedan koren hipa
  for (Cvor* x : hip) {
    // dok postoji drugi koren (u novom hipu) sa istim brojem dece kao x
    int brojDece = x->deca.size();
    while (pom[brojDece] != nullptr) {
      // spajamo dva korena tako da je manji podatak iznad
      Cvor* y = pom[brojDece];
      if (x->podatak > y->podatak)
        swap(x, y);
      x->deca.push_back(y);
```

```
      // kada mu se pridruzi y, koren x ima jedno dete vise, a u novom
      // hipu vise ne postoji hip koji ima stari broj dece
      pom[brojDece] = nullptr;
      brojDece++;
    }
    // upisujemo x u novi hip
    pom[brojDece] = x;
  }
  // prebacujemo sve korene iz novog hipa u stari
  hip.clear();
  for (Cvor* cvor : pom)
    if (cvor != nullptr)
      hip.push_back(cvor);
}
```

1.2.2.5 Смањивање вредности кључа

Опишимо сада операцију смањивање вредности кључа чвору x (операцију `smanji_kljuc(H, x, v)`). Претпостављамо да је чвор x коме се вредност умањује већ познат тј. да је познат показивач на њега (јер је то случај у многим алгоритмима који користе Фибоначијеве хипове). Ако није, тј. ако претпостављамо да је позната само вредност кључа која се умањује, али не и чвор који садржи ту вредност (то је операција `smanji_vrednost(H, k, v)`) онда се чвор k са вредношћу кључа k може ефикасно пронаћи тако што се чува мапа која слика кључеве у показиваче на чворове (довољно је да се вредност кључа слика у показивач на било који чвор који је садржи).

Да би се ова операција могла ефикасно имплементирати, потребно је чворове проширити са неколико додатних података, које ћемо у тексту описати.

Смањивање вредности кључа се изводи на следећи начин: прво се датом чвору x промени вредност кључа на v, а затим се, ако x није корен, та вредност пореди са вредношћу кључа родитеља. Како би се могло приступити родитељу, у свим чворовима се чува информација о родитељском чвору (она не мора бити дефинисана једино у коренима дрвета). Ако је x корен или је вредност кључа родитеља чвора x мања или једнака v, онда услов хипа није нарушен те нису потребне никакве даље измене. Ако то није случај, услов хипа је нарушен те је хип потребно преуредити. Најпре се врши *одсецање* гране између чвора x и његовог родитеља и поддрво са кореном x се додаје у листу коренова.

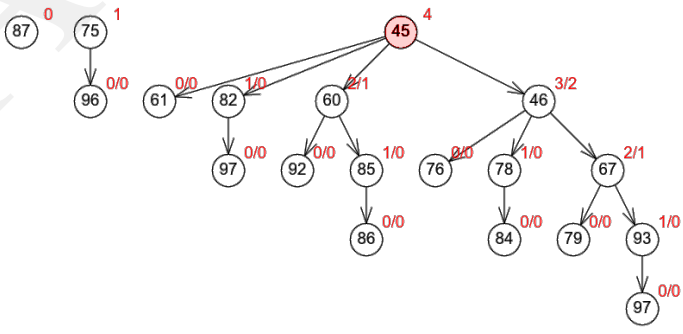
Поставља се питање да ли је дрво из којег је исечен чвор x потребно даље преуређивати. Одговор зависи од тога да ли је избацивањем поддрвета са кореном x нарушена инваријанта о броју чворова у том дрвету, тј. да ли је нарушена инваријанта о степенима

деце. Подсетимо се да су након консолидације чворови у дрвету обично такви да имају једно дете више од онога што би требало да имају на основу инваријанте тј. да им је могуће уклонити једно дете (тј. поддрво којем је оно корен), а да инваријанта за његовог родитеља и даље буде задовољена. Са друге стране, уклањање два детета нарушава инваријанту. Дакле, поставља се питање да ли смо родитељу чвора x већ уклонили неко дете од тренутка када је убачен у тренутно дрво. Да бисмо то знали, сваки чвор поред информације о свом степену (броју своје деце), садржи и *ознаку* да ли је „изгубио" дете од последњег тренутка када је постао дете неког другог чвора. Чворови се иницијално не означавају и сваки пут када чвор постане дете неког другог чвора или корен, уколико је био означен, ознака се уклања. Ако један чвор изгуби два детета, одсецамо и родитеља и поступак се наставља по истом принципу, навише, уз дрво. Прецизније, врши се следећи низ корака:

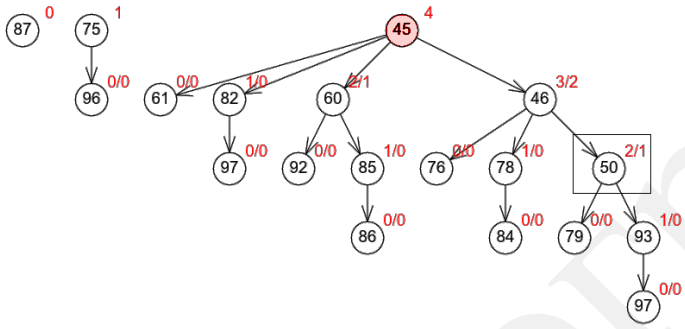
- Смањује се вредност кључа чвора x.
- Ако x није корен и ако је нова вредност v мања од вредности родитеља чвора x, одсеца се цело дрво чији је x корен, чвор x се додаје се у листу коренова (чиме то поддрво постаје самостално дрво) и уклања се ознака (уколико је чвор x био означен).
  - Ако је родитељ p чвора x неозначен (није му до сада било одсечено ниједно дете), он се означава.
  - У противном, одсеца се цело дрво чији је корен родитељски чвор p, чвор p се додаје у листу коренова (чиме и то поддрво постаје самостално дрво) и са њега се уклања ознака.
  - Рекурзивно се понавља претходни корак за све родитеље који нису већ коренови и којима су два пута одсечена деца.

Пример 1.2.4

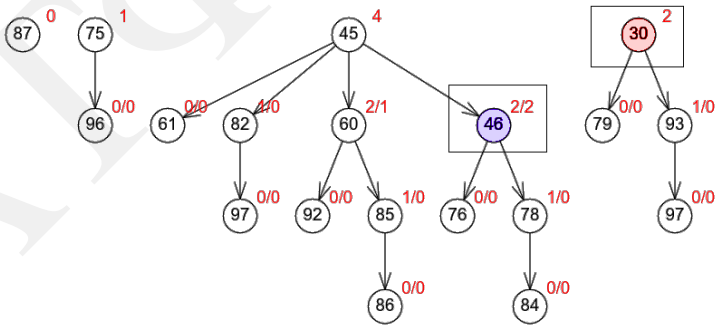
Размотримо хип приказан на наредној слици. Горе-десно од сваког чвора написан је тренутни степен чвора и минимални степен који чвор мора имати на основу инваријанте. На почетку скоро сви чворови имају и већи степен него што је то потребно.



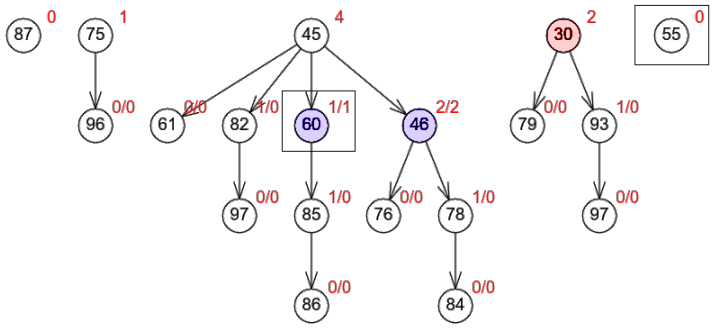
- *Претпоставимо да желимо да смањимо вредност кључа са* 67 *на* 50*. Пошто је након тог смањивања вредност и даље већа од вредности у родитељском чвору, нема потребе вршити даље измене.*



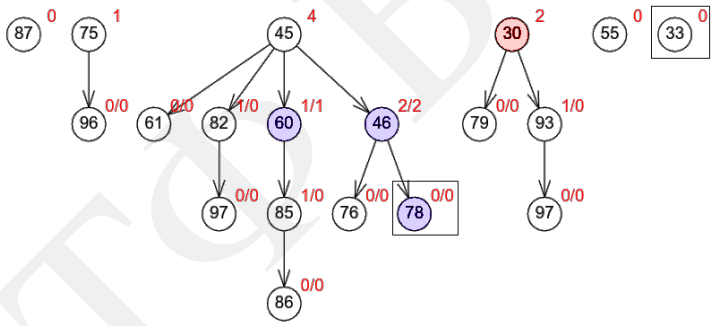
- *Даљим смањивањем вредности кључа* 50 *на* 30 *нарушава се однос вредности у том и родитељском чвору, па се чвор одсеца и ставља у низ коренова. Минимални чвор постаје* 30*, пошто је* 30 *мање од* 45*. Пошто је родитељски чвор* 46 *неозначен, нема потребе даље поправљати дрво након одсецања, већ се само родитељски чвор* 46 *означава. Приметимо да је његов степен у овом тренутку једнак минималној вредности степена на основу инваријанте и њему није могуће даље одсецати децу, а да инваријанта не буде нарушена. Добија се хип приказан на следећој слици.*



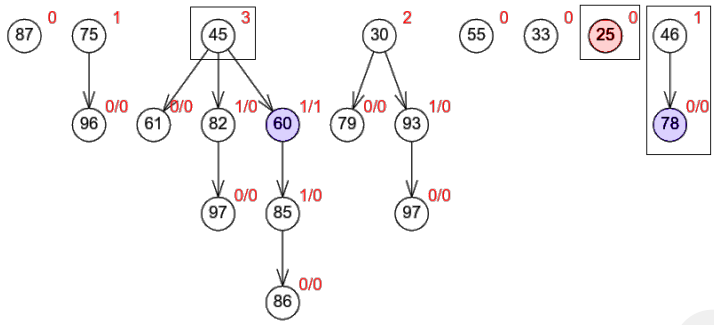
- *Смањимо сада вредност кључу* 92 *на* 55*. Чвор се одсеца и ставља у низ коренова. Пошто је код њему родитељског чвора* 60 *постојао један степен слободе (степен тог чвора је за* 1 *већи од минималне вредности) тј. пошто* 60 *није означен чвор, нема потребе даље модификовати дрво.*



- *Смањимо сада вредност кључа* 84 *на* 33. *Он се одсеца и ставља у низ коренова. Пошто је код и њему родитељског чвора* 78 *постојао један степен слободе тј. пошто* 78 *није означен чвор, нема потребе даље модификовати дрво.*



- *На крају, смањимо вредност кључа* 78 *на* 25. *Он се одсеца и ставља у низ коренова. Међутим, овај пут имамо ситуацију да је његов родитељски чвор* 46 *означен, што значи да је достигао минимални степен, па се додатним одсецањем његовог детета добија степен мањи од минималног. Зато је потребно одсећи и цео родитељски чвор* 46. *Његов родитељски чвор је корен* 45, *па нема потребе за даљим модификацијама и добија се хип приказан на следећој слици (корене нема потребе означавати, јер њихов степен може бити произвољан тј. инваријанта не захтева било какву минималну вредност за степен корена).*



1.3 Структуре података за представљање дисјунктних скупова

Понекад је у програму потребно одржавати неколико дисјунктних скупова (често подскупова неког скупа), при чему је потребно умети за дати елемент ефикасно пронаћи ком скупу припада (ту операцију зовемо `find` тј. `pronadji`) и ефикасно спојити два задата скупа у нови, већи скуп (ту операцију зовемо `union` тј. `unija`). Претпоставићемо да је сваки скуп једнозначно одређен неком ознаком (то може бити редни број скупа, назив скупа или неки канонски представник скупа). Аргументи операције `unija` не морају бити ознаке скупова чију унију треба креирати, већ могу бити произвољни елементи тих скупова. Приликом извођења уније полазни скупови се уклањају из колекције, и у колекцију се додаје њихова унија. Дакле, решавамо следећи проблем.

Проблем

Дефинисати структуру података која омогућава следећи интерфејс:

- `pronadji(x)` – *враћа ознаку скупа ком припада елемент* `x` *(та ознака може бити или неки идентификатор или неки канонски представник тог скупа);*
- `unija(x, y)` – *спаја скуп који садржи елемент* `x` *и скуп који садржи елемент* `y`.

Структура података која подржава овакав интерфејс се често назива по енглеском називу *union find* или *DSU* (енгл. disjoint set union), при чему се под тим именом обично подразумева и специфична, ефикасна имплементација (пре ње, у наставку ћемо приказати и једноставнију, али неефикаснију имплементацију).

Помоћу операције `pronadji` лако можемо за два елемента проверити да ли припадају истом скупу тако што за сваки од њих пронађемо ознаку скупа ком припада и проверимо да ли су ове ознаке једнаке.

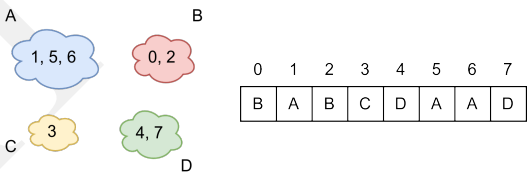
Приметимо да је оваква структура података корисна када год радимо са неким релацијама еквиваленције и када је потребно представити класе еквиваленције (које су дис-

јунктни подскупови скупа на ком је релација дефинисана). Провера да ли су два елемента у релацији се заснива на провери да ли припадају истој класи, а успостављање релације између било која два елемента доводи до спајања њихових класа еквиваленције. Често се употребљава за анализу повезаности неких елемената. На пример, корисници друштвених мрежа се могу груписати у класе еквиваленције на основу својих познанстава са другим корисницима (ако особа A познаје особу B, а особа B познаје особу C сматраћемо и да се особе A и C посредно познају) и коришћењем овакве структуре података можемо лако утврдити да ли су сви корисници међусобно повезани (преко заједничких познаника).

Структуре података за рад са дисјунктним скуповима имају различите примене. Једна од најзначајнијих је одржавање компоненти повезаности у Краскеловом алгоритму за конструкцију минималног повезујућег дрвета у графу (видети поглавље 2.10.3). Могу се користити и за сегментацију слика, тј. партиционисање слике на већи број дисјунктних региона, тако да су сви пиксели истог региона слични у погледу неког својства као што је боја, интензитет или текстура.

1.3.1 Наивна имплементација

Једна могућа имплементација структуре података са оваквим интерфејсом подразумева да се одржава пресликавање сваког елемента у ознаку скупа којем припада. Ако претпоставимо да разматрамо скуп од n елемената и да су сви елементи нумерисани бројевима од 0 до $n-1$, онда ово пресликавање можемо реализовати помоћу обичног низа где се на позицији сваког елемента налази ознака скупа којем он припада (уколико елементи нису нумерисани бројевима, можемо уместо низа да користимо мапу којом се ознаке елемената пресликавају у ознаке скупа ком елемент припада). Пример такве репрезентације приказан је на слици 1.7.



Слика 1.7: Представљање скупова обичним низом.

Операција `pronadji` је тада тривијална: довољно је из низа прочитати ознаку скупа ком елемент припада. Сложеност операције `pronadji` је тада $O(1)$.

Операција `unija` је много спорија јер подразумева да се ознаке свих елемената једног скупа мењају у ознаке другог, што захтева да се прође кроз цео низ. Сложеност операције `unija` је тада $\Theta(n)$.

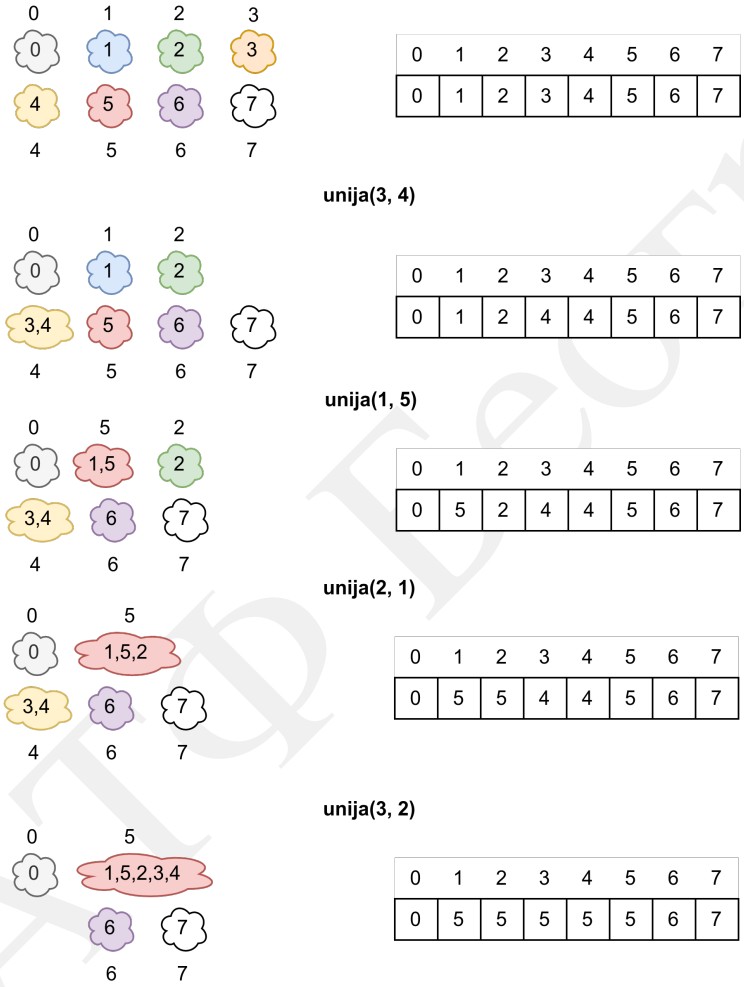
Да бисмо одредили амортизовану сложеност (што је веома релевантно код структура података код којих се одређене операције понављају пуно пута), потребно је да проценимо потребно време извршавања m операција (од којих је свака типа `unija` или `pronadji`). Операција `pronadji` се извршава у времену $O(1)$, али је операција `unija` сложености $O(n)$, где је n број елемената који одржавамо. Најгори случај наступа када стално вршимо уније, па време извршавања низа од m операција можемо да оценимо као $O(m \cdot n)$. Максимални број извршавања уније различитих скупова је $n - 1$, јер ће се након тога сви елементи објединити у исти скуп, па је и за велике вредности m укупно време свих тих операција $O(n^2)$.

Пример 1.3.1

Илуструјмо спровођење операција `pronadji` и `unija` над описаном имплементацијом структуре података за дисјунктне скупове на једном примеру. Претпоставимо да је почетно стање такво да сваки елемент припада засебном скупу. Размотримо на који начин се мења садржај одговарајућег низа након извршавања операција уније. Извршавање неколико операција уније приказано је на слици 1.8. Лево су приказани скупови, а десно низ којим су ти скупови представљени (изнад сваког низа приказани су индекси). Претпостављамо да ће приликом извршавања операције `unija(x,y)` ознака новог скупа бити ознака скупа којем припада елемент `y`.

Прикажимо сада имплементацију ове технике (једноставности ради користимо глобалне промењиве).

```
// oznaka podskupa kome pripada element
vector<int> id;

// na pocetku svaki element pripada zasebnom skupu
void inicijalizuj(int n) {
  id.resize(n);
  for (int i = 0; i < n; i++)
    id[i] = i;
}

// oznaku podskupa kome pripada element x citamo sa pozicije x iz niza
int pronadji(int x) {
   return id[x];
}

// pravimo uniju podskupova kome pripadaju dati elementi
```



Слика 1.8: Пример примена операције уније.

```
void unija(int x, int y) {
  int idx = id[x], idy = id[y];
  // oznake svih elemenata prvog podskupa menjamo u oznaku drugog podskupa
  for (int i = 0; i < id.size(); i++)
    if (id[i] == idx)
      id[i] = idy;
}

// elementi su u istom podskupu ako su im oznake iste
int u_istom_podskupu(int x, int y) {
  return pronadji(x) == pronadji(y);
}
```

Обратимо пажњу да се приликом извођења уније морају користити помоћне променљиве (размислите зашто је наредна имплементација неисправна).

```
// pravimo uniju skupova kome pripadaju dati elementi
void unija(int x, int y) {
  // oznake svih elemenata prvog skupa menjamo u oznaku drugog skupa
  for (int i = 0; i < id.size(); i++)
    if (id[i] == id[x])
      id[i] = id[y];
}
```

1.3.2 Ефикасна имплементација

Размотримо нешто другачију имплементацију ове структуре података у којој је операција `unija` временски ефикаснија.

1.3.2.1 Структура и операције

Кључна идеја на којој се заснива ефикасније решење је да елементе не пресликавамо у ознаке скупова, већ да скупове чувамо у облику дрвета (не нужно бинарних) тако да сваки елемент сликамо у његовог родитеља у дрвету. Сваки корен дрвета сликамо самог у себе и сматрамо га представником скупа представљеног тим дрветом (дакле, представник сваког скупа је корен његовог дрвета). Нагласимо да су у чворовима ових дрвета показивачи усмерени од деце ка родитељима, за разлику од класичних дрвета где показивачи у чворовима указују од родитеља ка деци.

Да бисмо за произвољни елемент сазнали ознаку скупа ком припада тј. да бисмо имплементирали операцију `pronadji`, потребно је да почев од тог елемента прођемо кроз низ родитељских чворова све док не стигнемо до корена. Унију два скупа (тј. операцију `unija`) у овом приступу можемо једноставно реализовати тако што корен једног дрвета

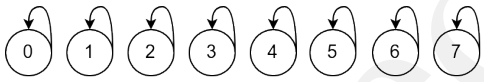
усмеримо ка корену другог.

Наивна имплементација која је описана у претходном поглављу одговара ситуацији у којој особа која промени адресу обавештава све друге особе о својој новој адреси, док имплементација коју тренутно описујемо одговара сценарију у коме само на старој адреси оставља информацију о својој новој адреси. Ово, наравно, мало успорава доставу поште, јер се мора прећи кроз низ преусмеравања, али ако тај низ није предугачак, може бити значајно ефикасније од првог приступа.

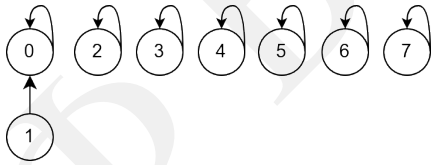
Пример 1.3.2

Прикажимо рад алгоритма на једном примеру. Скупове ћемо представљати дрветима.

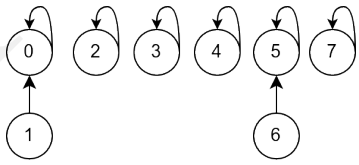
Претпостављамо да су на почетку сви скупови једночлани.



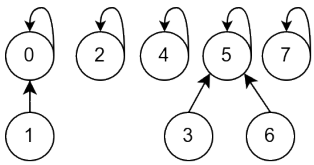
- *Извршава се операција* `unija(1, 0)`. *Представник скупа коме припада* 1 *је* 1, *па чвор* 1 *преусмеравамо ка представнику скупа коме припада* 0, *а то је* 0.



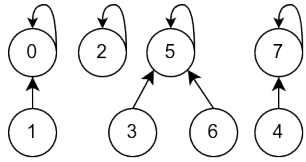
- *Извршава се операција* `unija(6, 5)`. *Представник скупа коме припада* 6 *је* 6, *па чвор* 6 *преусмеравамо ка представнику скупа коме припада* 5, *а то је чвор* 5.



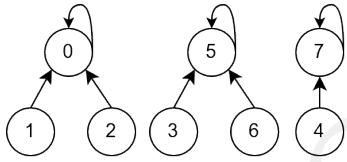
- *Извршава се операција* `unija(3, 6)`. *Представник скупа коме припада* 3 *је* 3, *па чвор* 3 *преусмеравамо ка представнику скупа коме припада* 6, *а то је чвор* 5.



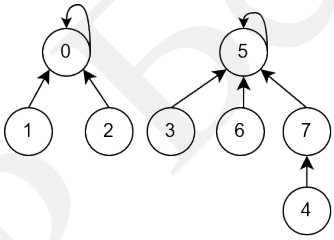
- *Извршава се операција* `unija(4, 7)`. *Представник скупа коме припада* 4 *је* 4, *па чвор* 4 *преусмеравамо ка представника скупа коме припада* 7, *а то је чвор* 7.



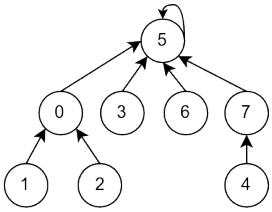
- *Извршава се операција* `unija(2, 0)`. *Представник скупа коме припада* 2 *је* 2, *па чвор* 2 *преусмеравамо ка представнику скупа коме припада* 0, *а то је чвор* 0.



- *Извршава се операција* `unija(4, 3)`. *Представник скупа коме припада* 4 *је* 7, *па чвор* 7 *преусмеравамо ка представнику скупа коме припада* 3, *а то је чвор* 5.



- *Извршава се операција* `unija(2, 6)`. *Представник скупа коме припада* 2 *је* 0, *па чвор* 0 *преусмеравамо ка представнику скупа коме припада* 6, *а то је чвор* 5.



Иако овако описана структура података има дрволику структуру, можемо је имплементирати коришћењем статичког низа. Наиме, пошто сваки елемент у дрвету има јединственог родитеља, на позицији неког елемента у низу можемо чувати индекс његовог родитеља. У случају да је елемент корен неког дрвета, његов родитељ је он сâм.

Пример 1.3.3

За дрво из претходног примера, низ родитеља има следећи садржај:

0	1	2	3	4	5	6	7
5	0	0	5	7	5	5	5

Имајући у виду овакву репрезентацију, кôд је прилично једноставно написати (једноставности ради претпостављамо да се подаци смештају у глобалним променљивим).

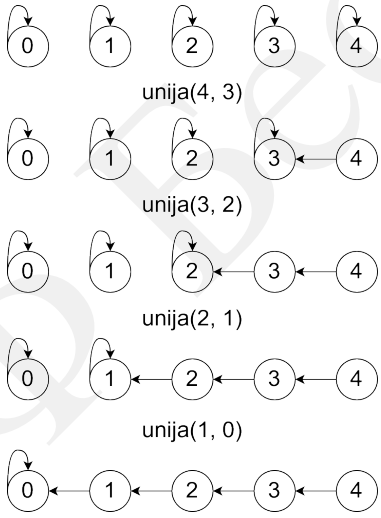
```
// roditelj svakog cvora u drvetu
vector<int> roditelj;

// na pocetku svaki element pripada zasebnom skupu
void inicijalizuj(int n) {
  roditelj.resize(n);
  for (int i = 0; i < n; i++)
    roditelj[i] = i;
}

// naziv podskupa kome element pripada dobijamo kao oznaku korena tog podskupa
int pronadji(int x) {
  // sve dok ne stignemo do korena
  while (roditelj[x] != x)
    // penjemo se u roditeljski cvor
    x = roditelj[x];
  return x;
}

// pravimo uniju podskupova kome pripadaju dati elementi
void unija(int x, int y) {
  int fx = pronadji(x), fy = pronadji(y);

  // x i y imaju istog predstavnika, pa su vec u istom podskupu
  if (fx == fy)
    return;

  // postavljamo da je koren prvog podskupa sin korena drugog podskupa
  roditelj[fx] = fy;
}
```

1.3.2.2 Уравнотежавање дрвета

Сложеност претходног приступа зависи од тога колико су дрвета којима се представљају скупови уравнотежена[6]. У најгорем случају се дрвета могу издегенерисати у листе и тада је сложеност најгорег случаја сваке од операција `unija` и `pronadji` $O(n)$.

Пример 1.3.4

Ако увек приликом извршавања операције `unija(x,y)` *усмеравамо показивач од представника скупа којем припада елемент x ка представнику скупа којем припада елемент y, извршиће се низ измена дрвета који је приказан на слици 1.9. Упит којим се тражи представник скупа којем припада елемент 4 се реализује низом корака којима се прелази преко елемената $3, 2, 1, 0$ и у случају већег броја елемената се добија веома неефикасна имплементација проналажења представника (а самим тим и уније, чија имплементација подразумева проналажење представника).*



Слика 1.9: Илустрација извршавања низа операција `unija(x,y)` када се увек показивач од представника скупа коме припада елемент x усмерава ка представнику скупа коме припада елемент y.

Тренутни алгоритам, дакле, није у најгорем случају ефикаснији од наивног приступа. Наиме, у наивном приступу се проналажење представника врши у времену $O(1)$, али унија увек захтева време $O(n)$, док овде и операција `pronadji` може имати сложеност

[6]Појам уравнотеженог дрвета може се прецизно дефинисати на разне начине. У овом тексту ћемо уравнотеженост разматрати само неформално: дрво сматрамо уравнотеженијим што је растојање листова од корена уједначеније.

$O(n)$. Сложеност тренутног алгоритма се може поправити ако се све време дрвета одржавају уравнотеженим. Доказаћемо да је у тој варијанти сложеност најгорег случаја сваке од операција `unija` и `pronadji` једнака $O(\log n)$. Тиме се постиже да је време извршавања низа од m операција типа `unija` или `pronadji` у најгорем случају једнако $O(m \log n)$ (док је у наивном приступу време извршавања овог низа операција, у случају када је број операција типа `unija` $O(m)$, једнако $O(mn)$).

Приликом прављења уније имамо слободу избора корена ког ћемо усмерити према другом корену и уравнотеженост се постиже тиме што се ова слобода на неки начин искористи. Постоје два уобичајена начина вршења уније: *унија на основу ранга (висине)* и *унија на основу броја елемената (величине)*.

Унија на основу ранга (висине)

Основна идеја вршења *уније на основу ранга (висине)* је да се приликом измена (а оне се врше само у склопу операције уније), ако је могуће, обезбеди да се висина[7] дрвета којим је представљена унија не повећа у односу на висине појединачних дрвета која представљају скупове чија се унија прави. Претпоставићемо да сваком чвору дрвета придружујемо број који представља висину тог чвора тј. дрвета са кореном у том чвору. Тај број ћемо назвати *ранг* (енгл. rank)[8] и рангове свих чворова ћемо одржавати у посебном низу `rang`. Ако се увек изабере да корен дрвета мањег ранга усмеравамо ка корену дрвета већег или једнаког ранга, тада се ранг уније повећава само ако су оба дрвета која унирамо истог ранга.

```
// roditelj svakog čvora drveta
vector<int> roditelj;
// rang (visina) svakog čvora drveta
vector<int> rang;

// na pocetku svaki element pripada zasebnom skupu
// i visina svakog drveta je 0
void inicijalizuj(int n) {
  roditelj.resize(n);
  rang.resize(n);
  for (int i = 0; i < n; i++) {
    roditelj[i] = i;
```

[7] Висину чвора можемо дефинисати као број грана на путањи од тог чвора до најудаљенијег листа у поддрвету са кореном у датом чвору. Висину дрвета рачунамо као висину његовог корена.

[8] У поглављу 1.3.2.3 посвећеном сажимању путева, видећемо да када се врше оптимизације, ранг не мора да буде увек једнак висини, али увек представља њену горњу границу тј. број од ког та висина сигурно није већа.

```
    rang[i] = 0;
  }
}

// pravimo uniju podskupova kojima pripadaju dati elementi
void unija(int x, int y) {
  int fx = pronadji(x), fy = pronadji(y);

  // x i y imaju istog predstavnika, pa su vec u istom podskupu
  if (fx == fy)
    return;

  // usmeravamo koren drveta nizeg ka korenu viseg ranga
  if (rang[fx] < rang[fy])
    swap(fx, fy);
  roditelj[fy] = fx;

  // ako su podskupovi istog ranga
  // unija ce biti za jedan veceg ranga
  if (rang[fx] == rang[fy])
    rang[fx]++;
}
```

Приметимо да су нам за извођење операције уније релевантне само вредности ранга представника скупова.

Докажимо лему која гарантује сложеност.

Лема 1.3.1 **[Унирање на основу ранга даје уравнотежена дрвета]**

Када се унија врши на основу ранга, у сваком дрвету чији је ранг h налази се бар 2^h чворова.

Доказ. Тврђење доказујемо математичком индукцијом.

- База индукције одговара полазном стању у коме је сваки чвор свој представник. Рангови свих дрвета су тада 0 и сва дрвета имају $2^0 = 1$ чвор, па тврђење важи.
- Покажимо да операција уније одржава ову инваријанту. По индуктивној хипотези знамо да ако два дрвета која представљају скупове који се уније имају рангове r_1 и r_2, онда је број чворова у њима редом бар 2^{r_1} и 2^{r_2} чворова. Уколико се унирањем ранг не повећа, инваријанта је тривијално очувана јер се број чворова

увећао, а ранг је остао исти. Једини случај када се унирањем повећава ранг је када је $r_1 = r_2$ и тада унирано дрво има ранг $r = r_1 + 1 = r_2 + 1$ и бар $2^{r_1} + 2^{r_2} = 2^{r_1} + 2^{r_1} = 2 \cdot 2^{r_1} = 2^{r_1+1} = 2^r$ чворова.

Тиме је тврђење доказано. □

Дакле, ранг дрвета које има n чворова је $O(\log n)$. Ако је он увек већи или једнак од висине, и висина дрвета је $O(\log n)$, па је сложеност операције проналажења представника у скупу од n чворова, која директно зависи од висине дрвета, реда $O(\log n)$. Пошто унирање након проналажења представника врши још само $O(1)$ операција, и сложеност унирања два скупа је $O(\log n)$. Дакле, одржавање висина дрвета под контролом нам гарантује логаритамску сложеност операција `pronadji` и `unija` и време извршавања низа од m таквих операција је у најгорем случају једнако $O(m \log n)$.

Унија на основу броја елемената (величине)

У приступу *унија на основу броја елемената (величине)*, уместо висине се уз сваки од скупова одржава и број његових елемената тог скупа (тј. у сваком чвору дрвета чува се информација о броју чворова дрвета којем је тај чвор корен).

```
// pravimo uniju podskupova kome pripadaju dati elementi
void unija(int x, int y) {
  int fx = pronadji(x), fy = pronadji(y);

  // x i y imaju istog predstavnika, pa su vec u istom podskupu
  if (fx == fy)
    return;

  // usmeravamo koren manjeg drveta kao korenu veceg drveta
  if (velicina[fx] < velicina[fy])
    swap(fx, fy);
  roditelj[fy] = fx;

  // azuriramo velicinu novog korena
  velicina[fx] += velicina[fy];
}
```

Ако увек усмеравамо представника скупа са мањим бројем елемената ка представнику скупа са већим бројем елемената, поново добијамо логаритамску сложеност најгорег случаја за обе основне операције. Ово важи зато што и овај начин прављења уније гарантује да не можемо имати високо дрво са малим бројем чворова. Наиме, за добијање дрвета висине 1, потребна су бар два дрвета висине 0, односно бар 2 чвора; за дрво ви-

сине 2 потребна су бар два дрвета висине 1 која имају бар по 2 чвора, тј. дрво висине 2 има бар 4 чвора, итд. Докажимо ово.

Лема 1.3.2 **[Унирање на основу броја елемената даје уравнотежена дрвета]**

Када се унија прави на основу броја елемената, у дрвету чија је висина h налази се бар 2^h чворова тј. за сваки корен r дрвета висине h са s чворова важи $s \geq 2^h$.

Доказ. Докажимо ово тврђење математичком индукцијом.

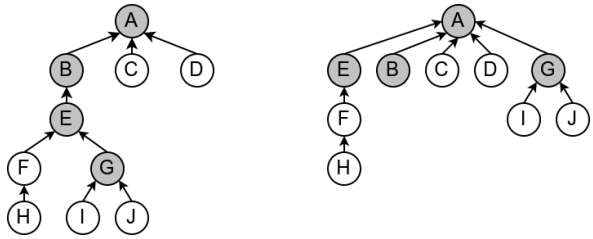
- У почетку је број чворова у сваком дрвету $s = 1$, а висина сваког дрвета је $h = 0$, па важи $s = 2^h$.
- Претпоставимо да тврђење важи пре спајања два дрвета чији су корени r_1 и r_2, који имају редом s_1 и s_2 елемената, висине h_1 и h_2. На основу индуктивне хипотезе знамо да је $s_1 \geq 2^{h_1}$ и $s_2 \geq 2^{h_2}$. Претпоставимо да се дрво r_2 придружује дрвету r_1 (што значи да дрво са кореном r_2 има мање или једнако чворова него оно са кореном r_1 тј. да је $s_2 \leq s_1$), чиме се добија дрво са s чворова висине h.
  - Ако дрво са кореном r_1 има већу висину од оног са кореном r_2, тј ако је $h_1 > h_2$, након спајања се висина дрвета са кореном r_1 не мања, а број чворова му се повећава, па тврђење леме тривијално наставља да важи. Заиста, важи $s = s_1 + s_2 \geq s_1 \geq 2^{h_1} = 2^h$.
  - У супротном се висина дрвета са кореном дрвета r_1 повећава и постаје за један већа од висине дрвета са кореном r_2 тј. важи $h = h_2 + 1$. Тада важи $s = s_1 + s_2 \geq 2 \cdot s_2 \geq 2 \cdot 2^{h_2} = 2^{h_2+1} = 2^h$.

Тиме је тврђење доказано. □

Из ове леме следи да су и у овом приступу висине свих дрвета увек реда $O(\log n)$, што гарантује ефикасност обе операције.

1.3.2.3 Сажимање путева

Иако је сложеност претходно описаних варијанти алгоритма сасвим прихватљива (сложеност извршавања m операција уније је $O(m \log n)$), она се може додатно побољшати веома једноставном техником познатом као *сажимање путева* или *компресија путева* (енгл. path compression). Наиме, приликом проналажења представника можемо све чворове кроз које пролазимо усмерити ка корену. Приметимо да тада поред операције `unija` и операција `pronadji` мења структуру дрвета којим је представљен тај скуп. Један начин да се то уради је да се након проналажења корена, поново прође кроз низ елемената и сви показивачи усмере ка корену (слика 1.10). На овај начин се постиже да будуће операције над тим скупом буду ефикасније.



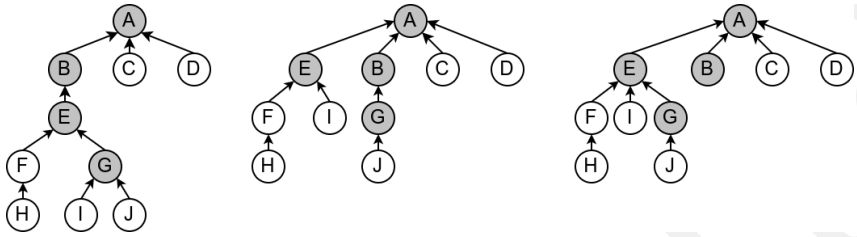
Слика 1.10: Илустрација поступка сажимања путева у два пролаза након тражења представника скупа коме припада елемент G.

```
// naziv podskupa kome element pripada dobijamo kao oznaku korena tog podskupa
int pronadji(int x) {
  int koren = x;
  // nalazimo oznaku podskupa kao koreni element podskupa
  while (koren != roditelj[koren])
    koren = roditelj[koren];
  // svim cvorovima na putanji od x do korena
  // postavljamo da je roditeljski cvor koren tog podskupa
  while (x != koren) {
    int tmp = roditelj[x];
    roditelj[x] = koren;
    x = tmp;
  }
  return koren;
}
```

За све чворове који се обилазе од полазног чвора до корена, дужине путања до корена се смањују на 1, а скраћују се и путање њихових потомака. Ако се врши унирање на основу броја елемената, приликом сажимања се не мењају бројеви елемената дрвета, па подаци придружени чворовима остају коректни. Ако вршимо унију на основу ранга и тумачимо рангове као висине дрвета, приликом сажимања низ рангова постаје неажуран (јер се неке висине смањују, а рангови се не ажурирају). Ипак, ни у овом случају низ не мора да се ажурира. Наиме, пошто се приликом сажимања висине смањују, а рангови се не мењају, рангови не представљају више висине чворова, али представљају њихове горње границе — висина чвора је увек мања или једнака од вредности његовог ранга. Лако се показује да се овим не нарушава сложеност операција. Наиме, пошто се унија и даље врши на основу ранга, и даље важи инваријанта гарантована лемом 1.3.1 да се у сваком дрвету које има ранг h налази бар 2^h чворова. Пошто је висина увек мања или једнака од ранга, важи и да је висина сваког дрвета које садржи n чворова највише $\log n$, што

гарантује сложеност најгорег случаја $O(\log n)$ за обе операције (видећемо да је услед сажимања амортизована сложеност још нижа).

У претходној имплементацији функције `pronadji` се два пута пролази кроз путању од чвора x до корена. Сличне перформансе се могу добити и у само једном пролазу. Постоје два начина на који се ово може урадити.



Слика 1.11: Илустрација два начина сажимања путева у једном пролазу током тражења представника скупа којем припада елемент I (лево је полазно дрво, у средини дрво које се добија првим алгоритмом, а десно дрво које се добија другим алгоритмом).

Први начин је да се сваки чвор кроз који се пролази током проналажења представника (осим детета корена) усмери ка родитељу свог родитеља. За све чворове који се обилазе (на путу од полазног чвора до корена), дужине путања до корена се након овога смањују двоструко (слика 1.11, средина), што је довољно за одличне перформансе.

```
// naziv podskupa kome element pripada dobijamo kao oznaku korena tog podskupa
int pronadji(int x) {
  int tmp;
  // za sve cvorove na putanji od x do korena
  while (x != roditelj[x]) {
    tmp = roditelj[x];
    // novi roditelj od x je roditelj njegovog roditelja
    roditelj[x] = roditelj[roditelj[x]];
    x = tmp;
  }
  return x;
}
```

Други начин подразумева да се, приликом проласка од чвора ка корену, сваки други чвор на путањи усмери ка родитељу свог родитеља (слика 1.11, десно).

```
// naziv podskupa kome element pripada dobijamo kao oznaku korena tog podskupa
int pronadji(int x) {
```

```
  // penjemo se do korena tako sto preskacemo po jedan cvor
  while (x != roditelj[x]) {
    // novi roditelj od x je roditelj njegovog roditelja
    roditelj[x] = roditelj[roditelj[x]];
    x = roditelj[x];
  }
  return x;
}
```

Приметимо да је на овај начин додата само једна линија кода у првобитну имплементацију операције `pronadji`.

Када се примени било који од три наведена облика сажимања путева, амортизована сложеност операција постаје готово константна[9].

Задатак: Први пут кроз матрицу

Логичка матрица димензије $n \times n$ у почетку садржи све нуле. Након тога се насумично додаје једна по једна јединица. Кретање по матрици је могуће само по јединицама и то само на доле, на горе, на десно и на лево. Написати програм који учитава димензију матрице, а затим позицију једне по једне јединице и одређује након колико њих је први пут могуће сићи од врха до дна матрице (са произвољног поља прве врсте које садржи јединицу до произвољног поља последње врсте матрице које садржи јединицу).

Опис улаза

Са стандардног улаза се учитава димензија матрице $1 \leq n \leq 200$, затим број поља m ($1 \leq m \leq n^2$) у које се уписује јединица, а затим у наредних m редова координате тих поља (број врсте и број колоне од 0 до $n-1$, раздвојени размаком).

Опис излаза

На стандардни излаз исписати најмањи број додатих јединица након којих је постало могуће стићи од врха до дна.

---

[9] Прецизније, амортизована сложеност операција када се примени сажимање путева једнака је $O(\alpha(n))$, где је $\alpha(n)$ такозвана инверзна Акерманова функција која јако споро расте. Акерманова функција је позната по томе што јако брзо расте. Она је дефинисана рекурентним релацијама: $A(0,n) = n+1$, $A(m+1,0) = A(m,1)$, $A(m+1,n+1) = A(m,A(m+1,n))$. Функција $\alpha(n)$ је инверзна функција функције $A(n,n)$ и она јако споро расте. За било који број n који је мањи од броја атома у целом универзуму (око 10^{80}) важи да је $\alpha(n) < 5$, тако да је амортизована временска сложеност операција практично константна.

Пример

Улаз	*Излаз*	*Објашњење*
4	8	После 8 учитаних поља, матрица постаје
9		1100
0 0		0101
0 1		1100
1 1		1001
3 3		и врх и дно постају спојени.
1 3		
2 0		
3 0		
2 1		
2 2		

Решење

Основна идеја је да се формирају сви подскупови поља матрице између којих постоји пут (они формирају класе еквиваленције тзв. компоненте повезаности). Сваки пут када се успостави веза између нека два поља матрице, подскупови којима она припадају се спајају. Провера да ли постоји пут између два поља матрице своди се онда на проверу да ли она припадају истом подскупу.

Путања од врха до дна постоји ако и само ако постоји путања од било ког поља у првој врсти матрице до било ког поља у последњој врсти матрице. Међутим, не морамо у сваком кораку морамо да проверавамо све парове елемената из прве и последње врсте. Додаћемо вештачки почетни чвор (назовимо га извор) и спојићемо га са свим чворовима у првој врсти матрице и завршни чвор (назовимо га ушће) који ћемо спојити са свим чворовима у последњој врсти матрице. Тада се у сваком кораку може проверити само да ли су извор и ушће спојени тј. да ли припадају истом подскупу. Приметимо да су додавањем ових помоћних чворова сви елементи прве врсте матрице обједињени у исти подскуп иако у њима пише нуле, међутим, то не нарушава коректност.

Подскупове можемо чувати помоћу структуре података за представљање дисјунктних подскупова.

```
int n;
cin >> n;

// alociramo matricu dimenzije n puta n
vector<vector<bool>> a(n);
for (int i = 0; i < n; i++)
  a[i].resize(n, false);
```

Компоненте повезаности су обојене различитим бојама. Пошто су извор и ушће исто обојени, постоји пут од врха до дна.

```
// svakom polju matrice (x, y) dodeljujemo jedinstveni redni broj (kod)
auto kod = [n](int x, int y) { return x*n + y; };

// dva dodatna veštačka čvora
const int izvor = n*n;
const int usce = n*n+1;

// inicijalizujemo union-find strukturu za sve elemente matrice
// (njih n*n), izvor i ušće
inicijalizuj(n*n + 2);

// spajamo izvor sa svim elementima u prvoj vrsti matrice
for (int i = 0; i < n; i++)
  unija(izvor, kod(0, i));

// spajamo sve elemente u poslednjoj vrsti matrice sa ušćem
for (int i = 0; i < n; i++)
  unija(kod(n-1, i), usce);

// broj jedinica
int m;
cin >> m;
```

```
// korak u kom se spajaju izvor i usce
int korak = -1;

// ucitavamo i obrađujemo jednu po jednu jedinicu
for (int k = 1; k <= m; k++) {
  int x, y;
  cin >> x >> y;
  // ako je u matrici već jedinica, nema šta da se radi
  if (a[x][y]) continue;
  // upisujemo jedinicu u matricu
  a[x][y] = true;
  // povezujemo podskupove u sva četiri smera
  if (x > 0 && a[x-1][y])     unija(kod(x, y), kod(x-1, y));
  if (x + 1 < n && a[x+1][y]) unija(kod(x, y), kod(x+1, y));
  if (y > 0 && a[x][y-1])     unija(kod(x, y), kod(x, y-1));
  if (y + 1 < n && a[x][y+1]) unija(kod(x, y), kod(x, y+1));
  // proveravamo da li su izvor i ušće spojeni
  if (pronadji(izvor) == pronadji(usce)) {
    korak = k;
    break;
  }
}

cout << korak << endl;
```

1.4 Упити распона

Неке структуре података су посебно погодне за проблеме у којима се тражи да се над елементима низа израчунавају статистике (нпр. збир, минимум, максимум) неких сегмената тј. распона узастопних елемената низа. Захтеве тог типа називамо *упити распона* (енгл. range queries). *Статички упити распона* (енгл. static range queries), описани у поглављу 1.4.1, подразумевају да се низ једном иницијализује и након тога не мења, док *динамички упити распона* (енгл. dynamic range queries), описани у поглављу 1.4.2, омогућавају да се низ мења између израчунавања статистика. Ове структуре података обично подржавају неке од следећих врста упита:

- одређивање елемента низа на датој позицији (енгл. *point query*),
- одређивање статистике елемената у датом сегменту низа (енгл. *range query*),

- ажурирање елемента низа на датој позицији (енгл. *point update*),
- ажурирање елемената низа у датом сегменту (енгл. *range update*), обично или тако што се сви елементи поставе на неку дату вредност или тако што се сви елементи увећају за дату вредност.

Приметимо да могућност ефикасног одређивања збира елемената произвољног сегмента низа гарантује уједно и могућност одређивања елемента на датој позицији (јер се вредност тог елемента може одредити као збир једночланог сегмента који садржи само тај елемент). Слично, могућност ефикасног ажурирања произвољног сегмента гарантује могућност ефикасног ажурирања појединачних елемената низа.

- *Низ збирова префикса* (енгл. prefix sum array) омогућава ефикасно рачунање статистика сегмената и вредности појединачног елемента (*range query*, *point query*).
- *Низ разлика суседних елемената* (енгл. difference array) омогућава ефикасно ажурирање сегмената (*range update*).
- *Сегментно дрво* (енгл. segment tree) и *Фенвиково дрво* (енгл. Fenwick tree) омогућавају ефикасно извршавање статистика сегмената и одређивање вредности елемената (*range query*, *point query*), као и ажурирање вредности појединачног елемента (*point update*), при чему се Фенвикова дрвета користе обично само за рачунање збира, док се сегментна дрвета користе за шири спектар статистика.
- *Лењо сегментно дрво* (енгл. lazy segment tree) омогућава ефикасно извршавање све четири наведене врсте упита.

Иако ћемо све структуре података приказати у једнодимензионом облику, у реалним применама се веома често разматрају дводимензионална, па и тродимензионална уопштења. На пример, у дводимензионалном случају могуће је ефикасно очитавати збирове произвољних правоугаоних сегмената матрице.

1.4.1 Статички упити распона

За почетак ћемо разматрати статичке упите распона, код којих је задатак омогућити ефикасно извршавање потенцијално великог броја различитих упита над сегментима датог низа, при чему се вредности у низу не мењају.

1.4.1.1 Збирови префикса

Проблем

Дефинисати структуру података која обезбеђује ефикасно израчунавање збирова сегмената датог низа (сегмент одређен позицијама $[a, b]$ се састоји од узастопних елемената низа од позиције a до позиције b, укључујући и њих).

У решењу грубом силом се сви елементи смештају у низ x и при сваком упиту се изнова рачуна збир елемената одговарајућег сегмента $[a, b]$. Укупно време да се сви упити изврше је реда $O(mn)$, где је са m означен број упита, а са n дужина низа, што је у случају дугачких низова и великог броја упита недопустиво неефикасно. И сложеност најгорег случаја и амортизована сложеност извршавања једног упита су реда $O(n)$.

Једноставно решење је засновано на идеји да уместо да чувамо елементе низа x, чувамо *низ збирова префикса* низа $P_0 = 0$, $P_{i+1} = \sum_{k=0}^{i} x_k$ за $0 \leq i \leq n$. Дакле, збир P_i чува збир првих i елемената низа (пошто бројање позиција почиње од нуле, збир елемената $x_0, \ldots, x_i$ садржи $i + 1$ сабирак и једнак је P_{i+1}). Збир елемената сваког сегмента $[a, b]$ онда можемо разложити на разлику префикса низа x до елемента b и префикса до елемента $a - 1$:

$$\sum_{k=a}^{b} x_k = \sum_{k=0}^{b} x_k - \sum_{k=0}^{a-1} x_k = P_{b+1} - P_a.$$

Сви збирови префикса P_i низа x могу се израчунати алгоритмом сложености $O(n)$ и сместити у додатни (а ако је уштеда меморије битна, онда чак и у оригинални) низ. Након овакве предобраде, збир сваког сегмента се може израчунати алгоритмом сложености $O(1)$, па је укупна сложеност извршавања m упита рачунања збира елемената неког сегмента низа x једнака $O(n + m)$.

Полазни низ x се може реконструисати рачунањем низа разлика суседних елемената низа префикса P: $x_i = P_{i+1} - P_i$ за $0 \leq i < n$.

Дводимензионални збирови префикса

У дводимензионалном случају довољно је за сваки елемент матрице на позицији (v, k) памтити збир $P_{v+1,k+1}$ елемената правоугаоног сегмента којем је горње лево теме поље $(0, 0)$, а доње десно (v, k), при чему се у врсти и колони 0 налазе нуле. Тада се збир произвољног сегмента одређеног горњим-левим теменом (v_1, k_1) и доњим-десним теменом (v_2, k_2) може изразити као $P_{v_2+1,k_2+1} - P_{v_2+1,k_1} - P_{v_1,k_2+1} + P_{v_1,k_1}$.

Пример 1.4.1

Збир елемената у црвеном правоугаонику леве матрице са слике 1.12 једнак је $87 - 30 - 32 + 11$. *Број* 87 *је збир елемената матрице од њеног горњег левог угла, до доњег десног угла црвеног правоугаоника, тј. збир елемената у зеленом, жутом, плавом и црвеном делу матрице (ж+з+п+ц). Број* 30 *је збир елемената матрице од њеног горњег левог угла до доњег десног угла плавог правоугаоника, тј. збир елемената у зеленом и плавом делу матрице (з+п). Број* 32 *је збир елемената матрице од њеног горњег левог угла до доњег*

1	3	2	3	2	3	1	1
4	3	1	4	2	4	2	3
1	2	1	3	2	3	2	4
1	4	2	3	2	1	3	1
3	2	4	3	2	2	2	2
4	2	2	1	4	1	3	4
1	3	4	1	2	2	1	3
1	3	2	4	3	3	4	2

0	0	0	0	0	0	0	0	0
0	1	4	6	9	11	14	15	16
0	5	***11***	14	21	25	***32***	35	39
0	6	14	18	28	34	44	49	57
0	7	19	25	38	46	57	65	74
0	10	24	34	50	60	73	83	94
0	14	***30***	42	59	73	***87***	100	115
0	15	34	50	68	84	100	114	132
0	16	38	56	78	97	116	134	154

Слика 1.12: Дводимензионални префиксни збирови за матрицу лево су приказани у матрици десно.

десног угла зеленог правоугаоника тј. збир елемената у зеленом и жутом правоугаонику (з+ж). Број 11 је збир елемената матрице од њеног горњег левог угла до доњег десног угла зеленог правоугаоника тј. збир елемената у зеленом правоугаонику (з). Изразом $87 - 32 - 30 + 11$*, се дакле, рачуна (з+ж+п+ц) - (ж+з) - (п+з) + з, што је једнако ц.*

Збирови префикса (и једнодимензиони и вишедимензиони) имају различите примене у анализи података, обради података, финансијској анализи и др. У обради слика се дводимензионални збирови префикса користе за израчунавање различитих прорачуна над регионима слике који омогућавају разне операције над сликом попут детекције ивица, замућења и других.

1.4.1.2 Разлике суседних елемената

Донекле сродан проблем је и следећи.

Проблем

Дефинисати структуру података која омогућава ефикасно извршавање упита облика $([a, b], c)$*, који подразумевају да се сви елементи низа* x *на позицијама из сегмента* $[a, b]$ *увећају за вредност* c*. Потребно је одредити садржај низа* x *након извршавања свих упита.*

Директно решење би за сваки упит у петљи увећавало све елементе низа x на позицијама из сегмента $[a, b]$. Сложеност тог наивног приступа је $O(mn)$, где је са m означен број упита, а са n дужина низа.

Много ефикасније решење се може добити ако се уместо елемената низа памти *низ разлика свака два суседна елемента низа*: $R_0 = x_0, R_i = x_i - x_{i-1}$ за свако i из-

међу 1 и $n-1$. Током увећавања свих елемената сегмента $[a,b]$ низа x за вредност c, мењају се само разлике између елемената на позицијама a и $a-1$ (разлика R_a се увећава за c) као и између елемената на позицијама $b+1$ и b (разлика R_{b+1} се умањује за c), па је измену могуће извршити у времену $O(1)$. Ако знамо све елементе низа, тада низ разлика суседних елемената можемо веома једноставно израчунати алгоритмом сложености $O(n)$. Са друге стране, ако знамо низ разлика неког низа, тада тај низ можемо такође веома једноставно реконструисати, израчунавајући збирове префикса низа разлика, у времену $O(n)$. На тај начин, на основу разлика елемената финалног низа, можемо лако реконструисати тај финални низ.

Ако не желимо да у имплементацији први и последњи елемент низа третирамо другачије од осталих, можемо претпоставити да почетни низ и низ разлика проширујемо са по једном нулом са леве и десне стране (тада је $R_i = x_{i+1} - x_i$, за свако i од 0 до $n-1$).

Оригинални низа се реконструише израчунавањем префиксних збирова низа разлика суседних елемената, што указује на дубоку везу између ове две технике. Заправо, разлике суседних елемената представљају одређени дискретни аналогон извода функције, док префиксни збирови онда представљају аналогон одређеног интеграла. Израчунавање збира сегмента као разлике два збира префикса одговара Њутн-Лајбницовој формули.

1.4.2 Динамички упити распона

За разлику од претходних, статичких упита над распонима, овде ћемо разматрати тзв. динамичке упите над распонима који дозвољавају да се низ мења током времена, тако да је потребно развити напредније структуре података које омогућавају извршавање оба типа упита (читање и ажурирање) ефикасно.

Наиме, низ збирова префикса омогућава ефикасно израчунавање збирова сегмената низа код низова чији се садржај не мења. Са друге стране, низ збирова префикса не омогућава ефикасно ажурирање елемената низа, јер је при свакој измени неког елемента низа потребно ажурирати и збирове префикса, што је нарочито неефикасно када се ажурирају елементи близу почетка низа (сложеност најгорег случаја је $O(n)$), па се не могу примењивати у ситуацијама у којима су упити израчунавања збирова сегмената испреплетени са упитима ажурирања вредности елемената низа.

Слично, низ разлика суседних елемената допушта стална ажурирања елемената низа. Међутим, извршавање упита којима се захтева одређивање елемената низа подразумева реконструкцију садржаја низа на основу низа разлика, што је сложености $O(n)$. Стога низ разлика није пожељно употребљавати у ситуацијама када су упити увећавања сегмената и очитавања вредности елемената низа испреплетани.

Проблеми које ћемо разматрати у овом поглављу су специфични по томе што омогућа-

вају да се упити ажурирања низа и очитавања његових статистика јављају испреплетано. За почетак размотримо мало једноставнији проблем.

Проблем

Дефинисати структуру података која обезбеђује ефикасно израчунавање збирова сегмената датог низа одређених позицијама $[a, b]$ (самим тим и појединачних елемената низа), као и ефикасно мењање вредности појединачних елемената низа.

Видећемо да неке структуре података допуштају да се уместо збира користе и неке друге операције. У наредним поглављима ћемо размотрити и сложенији проблем у ком ће бити допуштено ажурирање сегмената низа (а не само појединачних елемената).

Дакле, за почетак претпостављамо да имамо дат низ од m операција од којих је свака или израчунавање збира неког сегмента или ажурирање појединачног елемента низа и циљ је минимизовати укупно време извршавања овог низа операција. У овом случају нам није од користи структура података код које се једна од ове две операције извршава ефикасно, а друга неефикасно јер се може десити да је већина (или су све) од m датих операција баш тог другог типа. Наравно, треба узети у обзир и време иницијализације структуре података, међутим, пошто се иницијализација врши само једном, за разлику од упита за које претпостављамо да се извршавају пуно пута, фокусираћемо се на сложеност времена извршавања упита.

У наставку ћемо размотрити две различите, али донекле сличне структуре података које дају ефикасно решење претходног и њему сличних проблема: *сегментно дрво* и *Фенвиково дрво*. Идеја обе ове структуре података на неки начин наликује коришћењу префиксних сума: чувају се збирови неких унапред погодно изабраних сегмената и они се користе за ефикасно рачунање збира произвољног сегмента.

1.4.2.1 Сегментна дрвета

Једна структура података која омогућава прилично једноставно и ефикасно решавање претходно описаног проблема је *сегментно дрво* (енгл. segment tree). Слично као код низова префикса, током фазе предобраде израчунавају се збирови сегмената полазног низа, а онда се збир елемената произвољног сегмента полазног низа изражава путем тих унапред израчунатих збирова. Разлика је то што се не израчунавају збирови свих префикса, већ само посебно одабраних сегмената, као и то што се збир произвољног сегмента у општем случају израчунава обрадом неколико збирова сегмената (не само два). Сегментна дрвета се не користе само за рачунање збира елемената, већ се могу користити и за друге статистике сегмената које се израчунавају асоцијативним операцијама (на пример за одређивање производа, најмањег или највећег елемента, нзд-а или нзс-а свих елемената и слично). По истом принципу дефинисане су структуре података под називом *квад-дрвета* (енгл. quad tree) и *окт-дрвета* (енгл. oct tree) која представљају

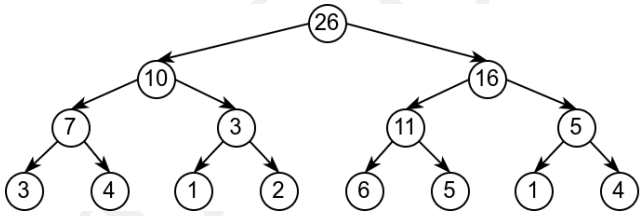
дводимензионална и тродимензионална уопштења сегментних древета.

Сегментна дрвета имају примену у области рачунарске геометрије, за проблеме који се решавају техником покретне праве, као што је проналажење пресечних тачака између дужи у равни, испитивање припадности тачке скупу интервала, израчунавање површине покривене скупом правоугаоника у равни и сл. Користе се и у другим областима, попут обраде слика и географских информационих система.

Опишимо процес конструкције сегментног дрвета. Претпоставимо да је дужина низа степен броја 2; ако није, низ се може допунити до најближег степена броја 2, у случају рачунања збирова нулама[10]. Чланови низа представљају листове дрвета. Групишемо два по два суседна чвора и на сваком претходном нивоу дрвета чувамо родитељске чворове који садрже збирове своја два детета.

Пример 1.4.2

Сегментно дрво којим се ефикасно могу рачунати збирови сегмената низа 3, 4, 1, 2, 6, 5, 1, 4 *приказано је на слици 1.13.*



Слика 1.13: Пример сегментног дрвета.

Пошто је дрво потпуно, најједноставнија имплементација подразумева да се чува имплицитно у низу (слично као у случају хипа). Претпоставићемо да елементе дрвета смештамо од позиције 1, јер је тада аритметика са индексима мало једноставнија (елементи полазног низа могу бити индексирани и уобичајено, кренувши од нуле). Корен се смешта на позицију 1, његова два детета на позиције 2 и 3, њихова деца на позиције 4, 5, 6 и 7 итд.

[10] На допуњена места треба уписати неутрални елемент за операцију која се спроводи, тј. елемент n који за свако x задовољава да је $f(x, n) = x$. На пример, ако се сегментним дрветом рачунају производи сегмената, низ се може допунити до степена двојке јединицама, ако се рачуна максимум, низ се може допунити вредностима $-\infty$, ако се рачуна минимум низ се може допунити вредностима ∞, ако се рачуна највећи заједнички делилац, низ се може допунити нулама, а ако се рачуна најмањи заједнички садржалац, низ се може допунити јединицама.

Пример 1.4.3

Дрво из претходног примера се представља следећим низом (први ред садржи позиције, а други елементе тог низа):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-	26	10	16	7	3	11	5	3	4	1	2	6	5	1	4

Корен дрвета је смештен на позицији 1. Ако полазни низ садржи n елемената, онда се у сегментном дрвету елементи полазног низа налазе се на позицијама $[n, 2n-1]$. Елемент који се у полазном низу налази на позицији p, се у сегментном дрвету налази на позицији $p+n$. Лево дете чвора на позицији k налази се на позицији $2k$, а десно на позицији $2k+1$. Дакле, на парним позицијама се налазе лева деца својих родитеља, а на непарним десна. Родитељ чвора k налази се на позицији $\lfloor \frac{k}{2} \rfloor$.

Пример 1.4.4

На слици 1.13 на којој је представљено сегментно дрво за низ од 8 елемената можемо уочити да се елементи полазног низа налазе на позицијама $[8, 15]$. *Лево дете чвора са вредношћу* 10 *који се налази на позицији* 2 *је чвор са вредношћу* 7 *који се налази на позицији* $2 \cdot 2 = 4$, *а десно дете је чвор са вредношћу* 3 *који се налази на позицији* $2 \cdot 2 + 1 = 5$. *Родитељ чвора на позицији* 4 *и чвора на позицији* 5 *је чвор који се налази на позицији* $\lfloor \frac{4}{2} \rfloor = \lfloor \frac{5}{2} \rfloor = 2$.

Формирање сегментног дрвета

Формирање сегментног дрвета на основу датог низа је веома једноставно.

Итеративни поступак

Најпре се елементи полазног низа прекопирају у дрво, кренувши од позиције n. Затим се сви унутрашњи чворови дрвета (од позиције $n-1$, па уназад до позиције 1) попуњавају као збирови своје деце (на позицију k уписујемо збир елемената на позицијама $2k$ и $2k+1$). Коректност овог поступка лако се доказује индукцијом.

```
// na osnovu datog niza a dužine n u kom su elementi smešteni od
// pozicije 0 formira se segmentno drvo i elementi mu se smeštaju u
// niz drvo krenuvši od pozicije 1
vector<int> formirajSegmentnoDrvo(const vector<int>& a) {
  int n = stepenDvojke(a.size());
  vector<int> drvo(2*n, 0);
  // kopiramo originalni niz u listove (u niz drvo, od pozicije n nadalje)
```

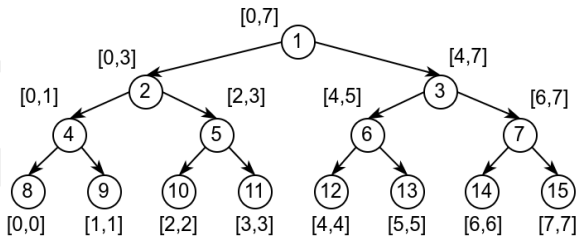
```
  copy(begin(a), end(a), next(begin(drvo), n));
  // ažuriramo roditelje već upisanih elemenata
  for (int k = n-1; k >= 1; k--)
    drvo[k] = drvo[2*k] + drvo[2*k+1];
  return drvo;
}
```

Сложеност ове операције је очигледно линеарна у односу на дужину низа n.

Претходни приступ формира дрво одоздо навише (прво се попуне листови, па онда унутрашњи чворови све док се не дође до корена).

Рекурзивни поступак

Још један начин да се сегментно дрво формира је рекурзивно, одозго наниже. Иако је ова имплементација компликованија и мало неефикаснија (додуше не асимпотски) услед рекурзивних позива, приступ одозго наниже је у неким каснијим операцијама неизбежан, па га илуструјемо на овом једноставном примеру. Сваки чвор дрвета представља збир одређеног сегмента позиција полазног низа. Сегмент је једнозначно одређен позицијом k у низу који одговара сегментном дрвету, али да бисмо олакшали имплементацију, границе тог сегмента можемо кроз рекурзију прослеђивати као параметар функције, заједно са вредношћу k (нека је то сегмент $[x, y]$). На слици 1.14 за сваки чвор сегментног дрвета дат је његов индекс у одговарајућем низу `drvo` и границе сегмента које тај чвор покрива.



Слика 1.14: Унутар сваког чвора је приказан његов индекс k у низу којим је дрво представљено, а поред чвора је приказан сегмент $[x, y]$ који тај чвор покрива.

Дрво крећемо да градимо од корена где је $k = 1$ и $[x, y] = [0, n - 1]$. Ако родитељски чвор покрива сегмент $[x, y]$, тада лево дете покрива сегмент $[x, \lfloor \frac{x+y}{2} \rfloor]$, а десно дете покрива сегмент $[\lfloor \frac{x+y}{2} \rfloor + 1, y]$. Дрво попуњавамо рекурзивно, тако што најпре попунимо лево поддрво, затим десно поддрво и на крају вредност у корену израчунавамо као збир вредности у левом и десном детету. Излаз из рекурзије представљају листови, које препознајемо по томе што покривају сегменте дужине 1, и у њих само копирамо

елементе са одговарајућих позиција полазног низа.

```
// od elemenata niza a sa pozicija [x, y] formira se segmentno drvo i
// elementi mu se smeštaju u niz drvo krenuvši od pozicije k
void formirajSegmentnoDrvo(const vector<int>& a, vector<int>& drvo,
                           size_t k, size_t x, size_t y) {
  if (x == y)
    // u listove prepisujemo elemente polaznog niza
    drvo[k] = x < a.size() ? a[x] : 0;
  else {
    // rekurzivno formiramo levo i desno poddrvo
    int s = (x + y) / 2;
    formirajSegmentnoDrvo(a, drvo, 2*k, x, s);
    formirajSegmentnoDrvo(a, drvo, 2*k+1, s+1, y);
    // izračunavamo vrednost u korenu
    drvo[k] = drvo[2*k] + drvo[2*k+1];
  }
}

// na osnovu datog niza a duzine n u kom su elementi smesteni od
// pozicije 0 formira se segmentno drvo i elementi mu se smeštaju u
// niz drvo krenuvši od pozicije 1
vector<int> formirajSegmentnoDrvo(const vector<int>& a) {
  // niz implicitno dopunjujemo nulama tako da mu duzina postane
  // najblizi stepen dvojke
  int n = stepenDvojke(a.size());
  vector<int> drvo(n * 2);
  // krećemo formiranje od korena, koji se nalazi u nizu drvo
  // na poziciji 1 i pokriva elemente na pozicijama [0, n-1]
  formirajSegmentnoDrvo(a, drvo, 1, 0, n - 1);
  return drvo;
}
```

Временску сложеност претходне рекурзивне имплементације можемо описати наредном рекурентном једначином: $T(n) = 2T(n/2) + O(1), T(1) = O(1)$. Њено решење се добија директно из мастер теореме и износи $O(n)$, што је исто као и у случају итеративног формирања сегментног дрвета.

Рачунање збира елемената сегмента

И збир елемената можемо израчунати било итеративним, било рекурзивним поступком.

Итеративни поступак

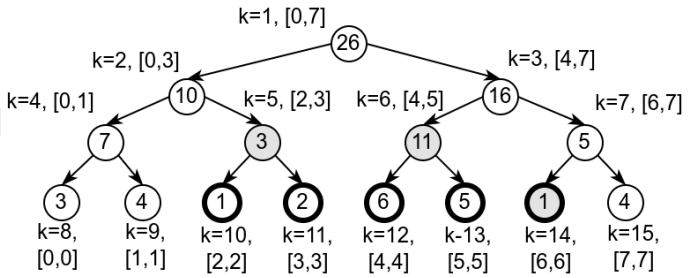
Илуструјмо итеративни поступак (кажемо и поступак одоздо навише) помоћу неколико примера.

Пример 1.4.5

Размотримо како бисмо нашли збир елемената низа $3, 4, 1, 2, 6, 5, 1, 4$ *на позицијама из сегмента* $[2, 6]$*, тј. збир елемената* $1, 2, 6, 5$ *и* 1*. У сегментном дрвету тај сегмент је смештен на позицијама из сегмента* $[2+8, 6+8] = [10, 14]$ *(слика 1.15). Јасно је да не треба ићи до листова и редом сабирати елементе, већ на паметан начин искористити већ израчунате збирове сегмената које се налазе у дрвету.*

Збир прва два елемента $1, 2$ *(који су на позицијама* 10 *и* 11*) се налази у чвору изнад њих (на позицији* 5*), збир наредна два елемената* 6 *и* 5 *(који су на позицијама* 12 *и* 13*) се налази такође у чвору изнад њих (на позицији* 6*), док се у родитељском чвору елемента* 1 *(који је на позицији* 14*) налази његов збир са елементом* 4 *(који је на позицији* 15 *и који не припада сегменту који сабирамо). Зато збир елемената на позицијама из сегмента* $[10, 14]$ *у сегментном дрвету можемо разложити на збир елемената на позицијама из сегмента* $[5, 6]$ *и елемента на позицији* 14 *који одмах засебно додајемо на тражени збир. На овај начин пењемо се на ниво изнад текућег (долазимо до родитеља тих чворова) и настављамо поступак рачунајући збир елемената на позицијама из сегмента* $[5, 6]$*.*

На позицији 5 *налази се елемент* 3*, а на позицији* 6 *елемент* 11*. У родитељском чвору елемента* 3 *(на позицији* 5*) налази се збир са елементом лево од њега, који не припада сегменту који треба да саберемо. Слично, у родитељском чвору елемента* 11 *(на позицији* 6*) налази се збир овог елемента са елементом десно од њега који не припада сегменту чији збир рачунамо. Стога оба елемента* 3 *и* 11 *додајемо засебно на збир и остајемо са празним сегментом, чиме се поступак рачунања збира датог сегмента завршава.*



Слика 1.15: Рачунање збира елемената из сегмента $[2, 6]$ приступом одоздо навише. Подебљани су листови који одговарају траженом сегменту, а обојени су чворови чије се вредности сабирају.

Пример 1.4.6

Размотримо како бисмо рачунали збир елемената на позицијама из сегмента $[3, 7]$*, тј. збир елемената* $2, 6, 5, 1, 4$*. У сегментном дрвету тај сегмент је смештен на позицијама* $[3 + 8, 7 + 8] = [11, 15]$.

У родитељском чвору елемента 2 *(који је на позицији* 11*) налази се његов збир са елементом* 1*, лево од њега који не припада сегменту који сабирамо. Стога елемент* 2 *одмах додајемо на збир.*

Збирови елемената 6 *и* 5 *(на позицијама* 12 *и* 13*) и елемената* 1 *и* 4 *(на позицијама* 14 *и* 15*) се налазе у чворовима изнад њих (на позицијама* 6 *и* 7*), па се проблем своди на израчунавање збира елемената на позицијама* 6 *и* 7.

Тај збир је већ израчунат, износи 16 *и налази се на позицији* 3*, тако да је само потребно да и њега додамо на збир.*

Дакле, резултат се добија сабирањем елемента 2 *на позицији* 11 *и елемента* 16 *на позицији* 3.

Генерално, за све унутрашње елементе сегмента чији збир рачунамо смо сигурни да се њихов збир налази у чворовима изнад њих. Једини изузетак могу да буду елементи на крајевима сегмента.

- Ако је елемент на левом крају сегмента лево дете (што је еквивалентно томе да се налази на парној позицији) тада се у његовом родитељском чвору налази његов збир са елементом десно од њега који такође припада сегменту који треба сабрати (осим евентуално у случају једночланог сегмента).
- У супротном (ако се налази на непарној позицији), у његовом родитељском чвору је његов збир са елементом лево од њега, који не припада сегменту који сабирамо. У тој ситуацији, тај елемент ћемо посебно додати на збир и искључити из сегмента који сабирамо помоћу родитељских чворова.
- Ако је елемент на десном крају сегмента лево дете (ако се налази на парној позицији), тада се у његовом родитељском чвору налази његов збир са елементом десно од њега, који не припада сегменту који сабирамо. И у тој ситуацији, тај елемент ћемо посебно додати на збир и искључити из сегмента који сабирамо помоћу родитељских чворова.
- Коначно, ако се крајњи десни елемент налази у десном чвору (ако је на непарној позицији), тада се у његовом родитељском чвору налази његов збир са елементом лево од њега, који припада сегменту који сабирамо (осим евентуално у случају једночланог сегмента).

```
// izračunava se zbir elemenata polaznog niza dužine n koji se
// nalaze na pozicijama iz segmenta [a, b] na osnovu segmentnog drveta
// koje je smešteno u nizu drvo, krenuvši od pozicije 1
int zbirSegmenta(const vector<int>& drvo, int a, int b) {
  int n = drvo.size() / 2;
  a += n; b += n;
  int zbir = 0;
  // sve dok je segment neprazan
  while (a <= b) {
    // ako je levi kraj segmenta desno dete, dodajemo ga posebno u zbir
    if (a % 2 == 1) zbir += drvo[a++];
    // ako je desni kraj segmenta levo dete, dodajemo ga posebno u zbir
    if (b % 2 == 0) zbir += drvo[b--];
    // penjemo se na nivo iznad, na roditeljske cvorove
    a /= 2;
    b /= 2;
  }
  return zbir;
}
```

Пошто се у сваком кораку дужина сегмента $[a, b]$ полови, а она је иницијално сигурно мања или једнака n, сложеност ове операције је $O(\log n)$.

Рекурзивни поступак

Претходна имплементација врши израчунавање одоздо навише. И за ову операцију можемо направити рекурзивну имплементацију која врши израчунавање одозго наниже. Функција као аргумент прима чвор дрвета (који је одређен позицијом k и који садржи збир елемената на позицијама из сегмента $[x, y]$). У општем случају, неки елементи на позицијама сегмента $[a, b]$ чији збир елемената рачунамо су лево од текућег чвора, а неки десно (или су сви лево или сви десно). За сваки чвор у сегментном дрвету функција враћа колики је допринос сегмента који одговара том чвору и његовим наследницима траженом збиру елемената на позицијама из сегмента $[a, b]$ у полазном низу. На почетку крећемо од корена и рачунамо допринос целог дрвета збиру елемената из сегмента $[a, b]$. Постоје три различита могућа односа између сегмента $[x, y]$ који одговара текућем чвору и сегмента $[a, b]$ чији збир елемената тражимо.

- Ако су сегменти дисјунктни, тј. ако је $y < a$ или је $x > b$, допринос текућег чвора збиру сегмента $[a, b]$ је нула.
- Ако је сегмент $[x, y]$ у потпуности садржан у сегменту $[a, b]$, тј. ако је $a \leq x$ и $y \leq b$, тада је допринос потпун, тј. цео збир сегмента $[x, y]$ (а то је број уписан у

низу на позицији k) доприноси збиру елемената на позицијама из сегмента $[a, b]$.

- У супротном, сегменти се секу и тада је допринос текућег чвора једнак збиру доприноса његовог левог и десног детета.

Из овог разматрања следи наредна имплементација.

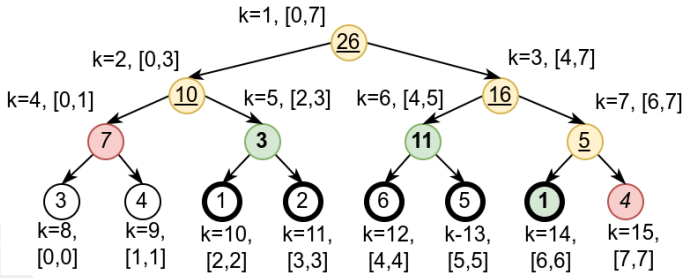
```
// izračunava se zbir onih elemenata polaznog niza, koji se nalaze na
// pozicijama iz segmenta [a, b] i koji se ujedno nalaze u delu
// segmentnog drveta ciji je koren u nizu drvo na poziciji k (taj deo
// drveta pokriva elemente na pozicijama segmenta [x, y] originalnog niza)
int zbirSegmenta(const vector<int>& drvo, int k, int x, int y, int a, int b) {
  // segmenti [x, y] i [a, b] su disjunktni
  if (b < x || a > y) return 0;
  // segment [x, y] je potpuno sadržan unutar segmenta [a, b]
  if (a <= x && y <= b)
      return drvo[k];
  // segmenti [x, y] i [a, b] se seku
  int s = (x + y) / 2;
  return zbirSegmenta(drvo, 2*k, x, s, a, b) +
         zbirSegmenta(drvo, 2*k+1, s+1, y, a, b);
}

// izračunava se zbir elemenata polaznog niza na pozicijama iz
// segmenta [a, b], na osnovu segmentnog drveta smeštenog u nizu drvo
int zbirSegmenta(const vector<int>& drvo, int a, int b) {
  int n = drvo.size() / 2;
  // krećemo od drveta smeštenog od pozicije 1, koje
  // pokriva elemente polaznog niza na pozicijama iz segmenta [0, n-1]
  return zbirSegmenta(drvo, 1, 0, n-1, a, b);
}
```

Пример 1.4.7

Размотримо како се овим приступом одређује збир елемената на позицијама из сегмента $[2, 6]$ у оригиналном низу (у сегментном дрвету приказаном на слици 1.16, ти елементи су на позицијама $[2 + 8, 6 + 8] = [10, 14]$)

- *Извршавање креће од корена. Сегмент $[0, 7]$ се сече са сегментом $[2, 6]$ те ће збир бити једнак суми доприноса сегмената $[0, 3]$ и $[4, 7]$.*
- *Сегмент $[0, 3]$ се сече са сегментом $[2, 6]$ те опет правимо два рекурзивна позива за сегменте $[0, 1]$ и $[2, 3]$.*

- *Сегмент* $[0, 1]$ *је дисјунктан са сегментом* $[2, 6]$ *па је његов допринос траженој суми* 0.
- *Сегмент* $[2, 3]$ *је садржан у сегменту* $[2, 6]$ *те је његов допринос потпун – једнак је вредности* 3 *коју тај чвор чува.*
- *С друге стране, сегмент* $[4, 7]$ *се сече са сегментом* $[2, 6]$ *те опет правимо два рекурзивна позива за сегменте* $[4, 5]$ *и* $[6, 7]$.
- *Сегмент* $[4, 5]$ *је у потпуности садржан у сегменту* $[2, 6]$ *те је његов допринос потпун и износи* 11.
- *Сегмент* $[6, 7]$ *се сече са сегментом* $[2, 6]$, *па из њега стартујемо два рекурзивна позива: за сегменте* $[6, 6]$ *и* $[7, 7]$.
- *Сегмент* $[6, 6]$ *је у потпуности садржан у сегменту чији збир рачунамо, те је његов допринос потпун и износи* 1.
- *Сегмент* $[7, 7]$ *је дисјунктан са сегментом чији збир рачунамо, па је његов допринос једнак нула. Дакле, укупна вредност суме сегмента биће једнака* $3 + 11 + 1 = 15$.

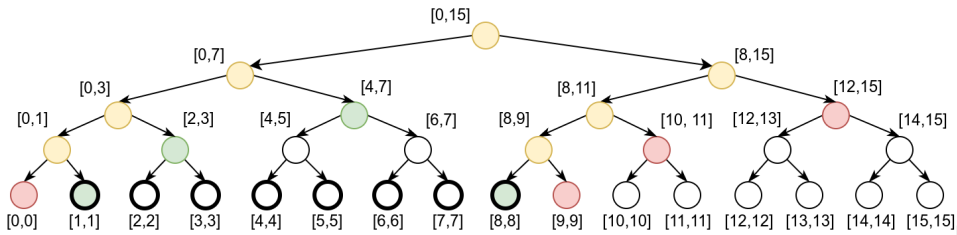


Слика 1.16: Рачунање збира елемената из сегмента $[2, 6]$ приступом одозго наниже. Жутом бојом и подвученим бројевима су означени чворови који одговарају сегментима који се секу са траженим сегментом, црвеном бојом и искошеним бројевима чворови који су дисјунктни, а зеленом бојом и подебљаним бројевима чворови који одговарају сегментима који су у потпуности садржани у сегменту који се сабира.

И ова имплементација има сложеност $O(\log n)$. Докажимо то.

Лема 1.4.1 **[Број посећених чворова на сваком нивоу]**

Приликом рекурзивног обиласка сегментног дрвета, за сваки ниво сегментног дрвета обилазе се највише по четири чвора.



Слика 1.17: Рачунање збира елемената из сегмента $[1, 8]$ приступом одозго наниже. Жутом бојом су означени чворови који одговарају сегментима који се секу са траженим сегментом, црвеном бојом чворови који су дисјунктни, а зеленом бојом чворови који одговарају сегментима који су у потпуности садржани у сегменту који се сабира.

Доказ. Докажимо ово тврђење принципом математичке индукције.

На првом нивоу се посећује само један чвор, корен дрвета, тако да се на овом нивоу посећује мање од четири чвора и база индукције важи.

Размотримо сада произвољни ниво дрвета: према индуктивној хипотези на њему се посећује највише четири чвора.

- Ако се посећује највише два чвора, у наредном нивоу се посећује највише четири чвора јер сваки чвор може да произведе највише два рекурзивна позива.
- Претпоставимо да се на текућем нивоу посећују три или четири чвора. Они могу бити суседни (слика 1.16, ниво 2), али и не морају (слика 1.17, ниво 3). Пошто чворови на једном нивоу сегментног дрвета садрже суме дисјунктних сегмената, једини чворови из којих се могу покренути рекурзивни позиви су они који садрже границе сегмента чију суму рачунамо (сиви чворови на слици): остали сегменти ће бити или у потпуности садржани или дисјунктни са сегментом чију суму рачунамо (или зелени, или црвени на слици). Стога, на сваком нивоу постоје само два чвора из којих се могу направити рекурзивни позиви, тако да ће и наредни ниво задовољавати полазно тврђење.

□

Пошто је висина сегментног дрвета $O(\log n)$, укупно се посећује највише $4 \log n$ чворова сегментног дрвета, те је сложеност операције израчунавања збира сегмента приступом одозго наниже такође $O(\log n)$.

Ажурирање вредности елемента

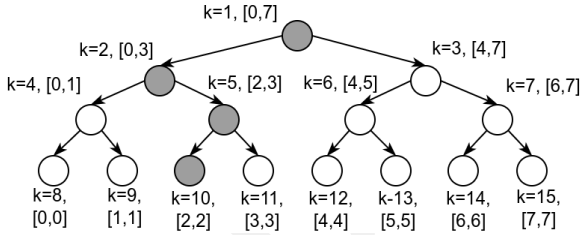
И функцију за ажурирање сегментног дрвета при ажурирању вредности неког елемента полазног низа можемо имплементирати итеративно и рекурзивно.

Итеративни поступак

Приликом ажурирања неког елемента потребно је ажурирати све чворове на путањи од тог листа до корена. С обзиром на то да знамо позицију родитеља сваког чвора, ова операција се може веома једноставно итеративно имплементирати.

Пример 1.4.8

На слици 1.18 је на једном примеру приказано које све чворове је потребно ажурирати након измене елемента на позицији 2 у оригиналном низу.



Слика 1.18: Ажурирање елемента на позицији 2 у полазном низу. Чворови чије се вредности ажурирају су обојени.

```
// ažurira segmentno drvo smešteno u niz drvo od pozicije 1
// koje sadrži elemente polaznog niza a dužine n u kom su elementi
// smešteni od pozicije 0, nakon što se na poziciju i polaznog
// niza upiše vrednost v
void upisi(vector<int>& drvo, int i, int v) {
  int n = drvo.size() / 2;
  // prvo ažuriramo odgovarajući list
  int k = i + n;
  drvo[k] = v;
  // ažuriramo sve roditelje izmenjenih čvorova
  for (k /= 2; k >= 1; k /= 2)
    drvo[k] = drvo[2*k] + drvo[2*k+1];
}
```

Пошто се k полови у сваком кораку петље, а креће од вредности највише $2n - 1$, и сложеност ове операције је $O(\log n)$.

Рекурзивни поступак

И ову операцију можемо имплементирати одозго наниже.

```
// ažurira segmentno drvo smešteno u niz drvo od pozicije k,
// koje sadrži elemente polaznog niza a dužine n sa pozicija iz
// segmenta [x, y], nakon što se na poziciju i niza upiše vrednost v
void upisi(vector<int>& drvo, int k, int x, int y, int i, int v) {
   if (x == y)
     // ažuriramo vrednost u listu
     drvo[k] = v;
   else {
     // proveravamo da li se pozicija i nalazi levo ili desno
     // i u zavisnosti od toga ažuriramo odgovarajuće poddrvo
     int s = (x + y) / 2;
     if (x <= i && i <= s)
       upisi(drvo, 2*k, x, s, i, v);
     else
       upisi(drvo, 2*k+1, s+1, y, i, v);
     // pošto se promenila vrednost u nekom od dva poddrveta
     // moramo ažurirati vrednost u korenu
     drvo[k] = drvo[2*k] + drvo[2*k+1];
   }
}

// ažurira segmentno drvo smešteno u niz drvo od pozicije 1,
// koje sadrži elemente polaznog niza a dužine n u kom su elementi
// smešteni od pozicije 0, nakon što se na poziciju i polaznog
// niza upiše vrednost v
void upisi(vector<int>& drvo, int i, int v) {
  int n = drvo.size() / 2;
  // krećemo od drveta smeštenog od pozicije 1 koje
  // sadrži elemente polaznog niza na pozicijama iz segmenta [0, n-1]
  upisi(drvo, 1, 0, n-1, i, v);
}
```

Сложеност претходне имплементације можемо описати рекурентном једначином: $T(n) = T(n/2) + O(1), T(1) = O(1)$ и њено решење износи $O(\log n)$. Наиме, дужина интервала $[x, y]$ се у сваком наредном позиву смањује два пута, а његова почетна дужина је једнака n.

Уместо функције `upisi` често се разматра функција `uvecaj` која елемент на позицији `i`

полазног низа увећава за дату вредност `v` и у складу са тим ажурира сегментно дрво. Свака од ове две функције се једноставно изражава преко оне друге.

Друге операције

Осим за проналажење збира, сегментно дрво се може користити и за многе друге асоцијативне операције. Најчешће су то проналажење минимума или максимума, НЗД или НЗС свих елемената сегмента и слично. У тим случајевима се на свим местима у коду операција сабирања мења одговарајућом бинарном асоцијативном операцијом (`min`, `max`, `nzd`, `nzs` и слично). Притом, ако се низ допуњава до дужине која је степен броја 2, уместо вредности 0, користе се неутрални елементи одговарајућих операција (на пример, за `min` то је вредност $+\infty$, за `max` то је $-\infty$, за `nzd` то је 0, а за `nzs` то је 1).

Неке мало напредније модификације омогућавају и извршавање сложенијих упита. На пример, можемо да направимо дрво које за сваки сегмент одређује максималну вредност тог сегмента и уједно број појављивања те максималне вредности у том сегменту. Тада се у сваком чвору чува пар (m, n) сачињен од те две вредности. У листовима дрвета се чува уређен пар елемента низа и вредности 1. Парови се комбинују на следећи начин. Ако је вредност у левом детету чвора (m_l, n_l), а у десном (m_d, n_d), тада, ако је $m_l > m_d$, важи $(m, n) = (m_l, n_l)$, ако је $m_d > m_l$, важи $(m, n) = (m_d, n_d)$, а ако је $m_l = m_d$, важи $(m, n) = (m_l, n_l + n_d)$. Ако дужина низа није степен броја 2, низ се допуњава вредностима $(-\infty, 0)$, јер је $-\infty$ неутрал за операцију `max`, при чему се та вредност заправо не јавља у низу (ако се зна да су сви елементи низа ненегативни, уместо $-\infty$ се може користити вредност 0).

Наведимо још неке типичне операције: одређивање броја елемената у сегменту који задовољавају дати услов (на пример, одређивање броја нула у сваком сегменту или броја парних бројева у сваком сегменту), одређивање позиције k-тог по реду елемента у сегменту који задовољава дати услов, одређивање позиције првог елемента у сегменту који је већи од дате вредности, одређивање прве позиције у сегменту низа који садржи само ненегативне вредности такве да је збир префикса сегмента до те позиције већи или једнак од дате вредности и слично.

Напоменимо да се неке од ових операција могу реализовати помоћу обичног сегментног дрвета и бинарне претраге тако да им је сложеност $O(\log^2 n)$, а прилагођавањем сегментног дрвета сложеност постаје $O(\log n)$. На пример, у низу ненегативних бројева за дати сегмент одређен позицијама $[l, d]$, бинарном претрагом интервала $[l, d]$ можемо одредити најмању вредност $k \in [l, d]$ тако да је збир префикса одређеног позицијама $[l, k]$ већи од дате вредности x. Бинарна претрага захтева $O(\log (d - l + 1)) = O(\log n)$ корака, а у сваком кораку збир сегмента одређеног позицијама $[l, k]$ можемо одређивати помоћу сегментног дрвета у времену $O(\log n)$ (јер је префикс одређен позицијама $[l, k]$ сегмента одређеног позицијама $[l, d]$ уједно сегмент оригиналног низа). Са

друге стране, можемо дефинисати рекурзивну функцију која у једном проласку кроз сегментно дрво проналази тражену вредност k. Наиме, ако је вредност левог детета (збира елемената у левој половини тренутног сегмента) већа или једнака од вредности x, онда префикс рекурзивно тражимо у левом детету, а ако је мања од вредности x, онда у десном детету тражимо најкраћи префикс чији је збир већи или једнак од разлике вредности x и вредности левог детета (што је збир елемената у левој половини тренутног сегмента). Тиме се упит извршава у времену $O(\log n)$.

1.4.2.2 Фенвикова дрвета (BIT)

Размотримо сада још једну структуру података која омогућава ефикасно рачунање статистика сегмената низа и ажурирање појединачних елемената: то су *Фенвикова*[11] *дрвета* (енгл. Fenwick tree), тј. *бинарно индексирана дрвета* (енгл. binary indexed tree, BIT). Видећемо да се она мало једноставније имплементирају од сегментних дрвета, користе мало мање меморије и могу да буду мало бржа од њих (иако су им и временска и просторна сложеност асимптотски једнаке). С друге стране, за разлику од сегментних дрвета, која су погодна за различите операције, Фенвикова дрвета су специјализована само за асоцијативне операције које имају одговарајуће инверзне (супротне) операције (нпр. збир или производ елемената сегмената се може налазити уз помоћ Фенвикових дрвета, јер сабирању одговара одузимање, а множењу дељење, али не и минимум, највећи заједнички делилац и слично). Потребу да за разматрану операцију постоји инверзна операција ћемо детаљније дискутовати када будемо говорили о реализацији самих операција. Сегментна дрвета могу даураде све што и Фенвикова, док обратно не важи.

Слично као што је случај код сегментних дрвета, иако се назива дрветом, Фенвиково дрво заправо представља низ вредности збирова неких паметно изабраних сегмената оригиналног низа a. Избор сегмената је у тесној вези са бинарном репрезентацијом индекса. Кључна Фенвикова идеја је следећа:

Као што се сваки природан број може добити сабирањем степена двојке, одређених његовом бинарном репрезентацијом (свака јединица у бинарној репрезентацији одређује неки степен двојке), тако се и сваки префикс низа може добити надовезивањем неких сегмената, одређених бинарном репрезентацијом границе тог сегмента (свака јединица у бинарној репрезентацији одређује један такав сегмент).

Поново ћемо, једноставности ради, претпоставити да се вредности смештају од позиције 1 (вредност на позицији 0 је ирелевантна) и то и у полазном низу a, и у низу у ком се смешта Фенвиково дрво. Прилагођавање кода ситуацији у којој су у полазном низу елементи смештени од позиције нула веома је једноставно, само је на почетку сваке функције која ради са дрветом индекс полазног низа потребно увећати за један

[11] Структуру је иницијално предложио Борис Ријабко, а име је добила по Питеру Фенвику, информатичару са Универзитета Окланд у Новом Зеленду.

пре даље обраде. Дакле, ако је полазни низ дужине n, елементи дрвета се смештају у посебан низ на позиције $[1, n]$.

Дефиниција Фенвиковог дрвета је следећа:

У дрвету се на позицији k чува збир вредности полазног низа са позиција из интервала $(f(k), k]$, где је $f(k)$ број који се добија од броја k тако што се у бинарном запису броја k прва јединица здесна замени нулом.

Пример 1.4.9

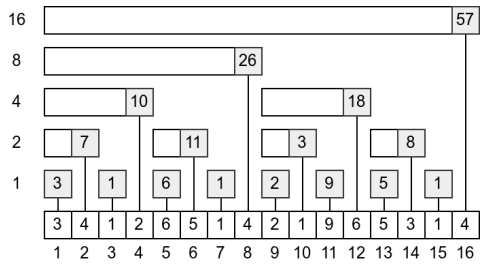
- *У Фенвиковом дрвету се на позицији $k = 21$ записује збир елемената полазног низа са позиција из интервала $(20, 21]$. Наиме, број 21 се бинарно записује као $(10101)_2$ и заменом прве јединице здесна нулом у његовом бинарном запису добија се бинарни запис $(10100)_2$, тј. број 20 (важи $f(21) = 20$).*
- *На позицији 20 у Фенвиковом дрвету налази се збир елемената полазног низа са позиција из интервала $(16, 20]$, јер се брисањем крајње десне јединице из његовог бинарног записа $(10100)_2$ добија бинарни запис $(10000)_2$ тј. број 16 (важи $f(20) = 16$).*
- *На позицији 16 у Фенвиковом дрвету чува се збир елемената полазног низа са позиција из интервала $(0, 16]$, јер се брисањем крање десне јединице из бинарног записа броја 16 добија 0 (важи $f(16) = 0$).*

На слици 1.19 је за низ дужине 16 приказано који се збирови сегмената полазног низа чувају на свакој од позиција у Фенвиковом дрвету. Приметимо да непарне позиције одговарају сегментима дужине 1, а да за степене броја 2 сегменти представљају префиксе полазног низа до те позиције.

Пример 1.4.10

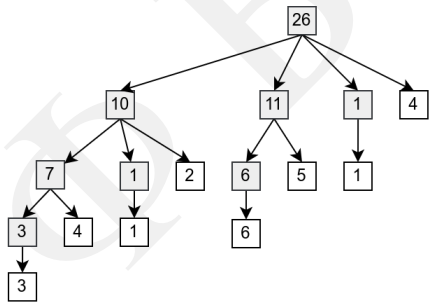
Размотримо низ a са вредностима $3, 4, 1, 2, 6, 5, 1, 4$.

0	1	2	3	4	5	6	7	8	k
	0001	0010	0011	0100	0101	0110	0111	1000	*k бинарно*
	0000	0000	0010	0000	0100	0100	0110	0000	*$f(k)$ бинарно*
	0	0	2	0	4	4	6	0	$f(k)$
	$(0, 1]$	$(0, 2]$	$(2, 3]$	$(0, 4]$	$(4, 5]$	$(4, 6]$	$(6, 7]$	$(0, 8]$	*интервал*
	3	4	1	2	6	5	1	4	*низ a*
	3	7	1	10	6	11	1	26	*Фенвиково дрво*



Слика 1.19: Приказ сегмената чији су збирови елемената смештени у елементима Фенвиковог дрвета дужине 16. На дну је приказан полазни низ, сегменти су означени правоугаоницима, а збир елемената сваког сегмента је приказан у сивом квадрату који се налази на позицији на којој је тај збир смештен у Фенвиковом дрвету када се оно представи помоћу низа.

На слици 1.20 приказана је зависност збирова сегмената од њихових подсегмената, одакле се јасно види да је заиста у питању дрволика структура (отуда и потиче назив Фенвиково дрво). У листовима ове структуре налазе се елементи полазног низа, док се у унутрашњим чворовима налазе елементи Фенвиковог дрвета.



Слика 1.20: Дрволика структура Фенвиковог дрвета.

Имплементација разних операција над Фенвиковим дрветом је веома једноставна, ако се зна начин да се из бинарног записа броја уклони прва јединица здесна тј. да се за дати број k израчуна вредност $f(k)$. Означимо са $b(k)$ бинарни број који садржи само једну јединицу и то на месту последње јединице у бинарном запису броја k. Важи да је $f(k) = k - b(k)$.

Под претпоставком да су бројеви записани у потпуном комплементу, вредност $b(k)$ се може израчунати изразом `k & -k`. Одузимањем те вредности од броја k тј. изразом `k - (k & -k)` добијамо ефекат брисања последње јединице у бинарном запису броја k и то представља имплементацију функције f.

Пример 1.4.11

Броју $k = 20$ *одговара бинарни запис* $(00010100)_2$*, а броју* $-k$ *бинарни запис* $(11101100)_2$*, те је* `k & -k` *једнако* $(00000100)_2$ *и представља број који садржи само једну јединицу и то на месту последње јединице у запису броја* k*. Одузимањем ове вредности од броја* k *добијамо број* 16 *коме одговара бинарна репрезентација* $(00010000)_2$*.*

Докажимо исправност операције уклањање последње јединице из бинарног записа броја.

Лема 1.4.2 **[Рачунање броја $f(k)$]**

За било који неозначени бинарни број k *већи од нуле изразом* `k & -k` *се израчунава број* $b(k)$ *тј. број који се састоји само од последње јединице у бинарном запису броја* k*, а изразом* `k - (k & -k)` *се израчунава број* $f(k)$ *добијен од броја* k *брисањем последње јединице из његовог бинарног записа.*

Доказ. Потпуни комплемент броја k се добија тако што се броју k инвертују сви битови и добијени резултат се увећа за 1. Нека број k има своју последњу јединицу на позицији p (ако има битова десно од позиције p, они су сви нуле). Када се негирају битови добија се број који има нулу на позицији p иза које до краја следе јединице (ако их уопште има). Сабирањем са 1 добија се број који на позицији p има 1 иза чега се налазе нуле. Дакле, лево од позиције p битови броја k и $-k$ су супротни, на позицији p оба имају јединице, а десно од позиције p оба имају нуле. Битовска конјункција бројева k и $-k$ даје резултат $b(k)$ који има само једну јединицу и то на позицији p (то је једино место где и k и $-k$ имају вредност 1). □

Други начин да се израчуна вредност $f(k)$ јесте рачунањем вредности израза `k & (k-1)`. Исправност овог израза се такође може једноставно доказати. Наиме, број `k-1` има све исте битове од крајњег левог до позиције последњег постављеног бита у броју k, а све инвертоване битове после крајњег десног постављеног бита у броју k.

Пример 1.4.12

Бинарни запис броја $k = 20$ *је* $(00010100)_2$*, а броја* $k-1 = 19$ *је* $(00010011)_2$ *и изразу* `k & (k-1)` *одговара бинарни запис* $(00010000)_2$*.*

Ипак, израз `k - (k & -k)` се чешће користи, због сличности са изразом `k + (k & -k)`, који се користи приликом израчунавања функције g која је дефинисана у поглављу 1.4.2.2 посвећеном ажурирању вредности елемената низа.

Рачунање збира елемената сегмента

Збир елемената у било ком префиксу полазног низа може да се добије као збир неколико елемената записаних у Фенвиковом дрвету.

Пример 1.4.13

Интервал позиција $(0, 21]$ *тј. префикс низа до позиције* 21 *се добија надовезивањем интервала* $(0, 16]$, $(16, 20]$ *и* $(20, 21]$.

Ове интервале је једноставно одредити. На позицији k се налази збир елемената са позиција из интервала $(f(k), k]$, на позицији $f(k)$ збир елемената са позиција из интервала $(f(f(k)), f(k)]$, итд. Поступак се наставља сабирајући елементе са позиција k, $f(k)$, $f(f(k))$, $f(f(f(k)))$ итд., све док се не дође до позиције нула.

Број елемената Фенвиковог дрвета чијим се сабирањем добија збир префикса полазног низа је $O(\log n)$. Наиме, у сваком кораку се број јединица у бинарном запису текућег индекса смањује, а број n се бинарно записује са највише $O(\log n)$ бинарних јединица.

Збир елемената префикса на позицијама из интервала $(0, k]$ полазног низа a смештеног у Фенвиково дрво `drvo` можемо онда израчунати наредном функцијом.

```
// na osnovu Fenvikovog drveta smeštenog u niz drvo
// izračunava zbir prefiksa (0, k] polaznog niza
int zbirPrefiksa(const vector<int>& drvo, int k) {
  int zbir = 0;
  while (k > 0) {
     zbir += drvo[k];
     k -= k & -k; // k = f(k)
  }
}
```

Када умемо да израчунамо збирове префикса полазног низа, збир произвољног сегмента одређеног позицијама $[a, b]$ полазног низа можемо израчунати као разлику збира префикса одређеног позицијама $(0, b]$ и збира префикса одређеног позицијама $(0, a-1]$. Пошто се оба префикса рачунају у времену $O(\log n)$, и збир сваког сегмента можемо израчунати у времену $O(\log n)$. Истакнимо на овом месту да је због ове операције важно да асоцијативна операција која се користи у Фенвиковом дрвету има инверзну операцију (у овом случају, пошто се ради о операцији сабирања, да бисмо могли да одузимањем две вредности префикса добијемо збир произвољног сегмента[12]).

[12] Обратимо пажњу да ако се Фенвиковим дрветом рачунају производи сегмената, елементи дрвета морају бити различити од нуле, због операције дељења.

Ажурирање вредности елемента

Идеју рачунања збира елемената произвољног сегмента као разлике два збира префикса смо видели већ раније, када смо у те сврхе одржавали низ свих збирова префикса. Основна предност Фенвикових дрвета у односу на низ свих збирова префикса је то што се могу ефикасно ажурирати. Наиме, ажурирање једног елемента полазног низа може резултовати изменом вредности великог броја префиксних збирова, нарочито ако је елемент који мењамо близу почетка низа (у најгорем случају број збирова које је потребно ажурирати је $O(n)$). Стога је операција ажурирања елемента када се користе префиксни збирови у најгорем случају сложености $O(n)$.

Размотримо функцију која ажурира Фенвиково дрво након увећања елемента у полазном низу на позицији k за вредност x. Тада је за x потребно увећати све оне збирове у дрвету у којима се као сабирак јавља и елемент на позицији k. Потребно је ажурирати све оне позиције m чији придружени сегмент садржи вредност k, тј. све оне позиције m такве да је $k \in (f(m), m]$, тј. позиције m за које важи

$$f(m) < k \leq m \tag{1.1}$$

Те позиције се израчунавају веома једноставно кренувши од полазне позиције k, слично као у претходној функцији, с тим што се уместо да се од броја k одузима вредност $b(k)$, број k увећава за вредност израза $b(k)$. Обележимо са $g(k)$ број $k+b(k)$, који се добија од броја k тако што се k сабере са бројем који има само једну јединицу у свом бинарном запису и то на позицији на којој се налази последња јединица у бинарном запису броја k. У имплементацији се број $g(k)$ лако може израчунати по формули `k + (k & -k)`.

Пример 1.4.14

За број $k = (101100)_2$ *важи* $g(k) = (101100)_2 + (100)_2 = (110000)_2$.

Доказаћемо да је потребно и довољно ажурирати вредности на позицијама k, $g(k)$, $g(g(k))$ итд., све док се не дође до позиције која је строго већа од дужине низа n.

Пример 1.4.15

Ако би се у претходном примеру елемент на позицији 3 *у полазном низу увећао за вредност* 4, *било би потребно повећати за* 4 *вредности елемената Фенвикове дрвета на позицијама* 3, 4 *и* 8. *До низа ових позиција бисмо могли да дођемо на следећи начин.*

- *Прва позиција је број* $k = 3$.

- *Наредну позицију $g(3)$ добијамо тако што бинарном запису броја 3 који износи $(0011)_2$, додајемо број 1 (број $b(3)$ који садржи тачно једну јединицу на позицији последње јединице у бинарном запису датог броја) и добијамо бинарни запис $g(3) = (0100)_2$ који одговара броју 4.*
- *Наредну позицију $g(g(3)) = g(4)$ бисмо добили тако што бисмо вредност 4 сабрали са $(0100)_2$ (бројем $b(4)$ који садржи тачно једну јединицу у свом бинарном запису и то на позицији последње јединице), чиме бисмо добили $g(4) = (1000)_2$ (бинарни запис броја 8). Овде се процедура завршава пошто смо стигли до последњег елемента у Фенвиковом дрвету.*

С обзиром на то да се број n бинарно записује са $O(\log n)$ битова и да се у сваком кораку број крајњих нула у запису броја повећава (иако у сваком кораку вредност броја расте), укупан број корака је једнак $O(\log n)$. Дакле, број елемената Фенвиковог дрвета чије је вредности потребно изменити износи $O(\log n)$.

```
// ažurira Fenvikovo drvo smešteno u niz drvo nakon što se
// u originalnom nizu element na poziciji k uveća za x
void uvecaj(vector<int>& drvo, int n, int k, int x) {
  while (k <= n) {
    drvo[k] += x;
    k += k & -k;
  }
}
```

Докажимо коректност описаног поступка. Кренимо од наредне леме.

Лема 1.4.3

За свако x између 1 и n, најмањи број m који задовољава услов $f(m) < x < m$ је $g(x)$.

Доказ. Покажимо најпре да $g(x)$ задовољава овај услов. Очигледно важи $x < g(x)$. Пошто $g(x)$ има све нуле од позиције последње јединице у бинарном запису броја x (укључујући и њу), па до краја, брисањем његове последње јединице тј. израчунавањем вредности $f(g(x))$ се сигурно добија број који је строго мањи од x.

Покажимо сада да је $g(x)$ најмањи број строго већи од x који задовољава дати услов, односно да ниједан број m између x и $g(x)$ не може да задовољи услов $f(m) < x$. Наиме, сви бројеви из интервала $(x, g(x))$ се поклапају са бројем x на свим позицијама пре крајњих нула, а на позицијама крајњих нула броја x имају бар неку јединицу, чијим се брисањем добија број који је већи или једнак x. □

Пример 1.4.16

У претходно разматраном примеру за $k = (101100)_2$*, као што смо већ видели важи* $g(k) = (110000)_2$ *и једини бројеви између* k *и* $g(k)$ *су* $(101101)_2$*,* $(101110)_2$ *и* $(101111)_2$*. Брисањем последње јединице у бинарном запису ових бројева добијају се редом бројеви* $(101100)_2$*,* $(101100)_2$ *и* $(101110)_2$ *и сви они су већи или једнаки од* k*.*

Докажимо и другу помоћну лему.

Лема 1.4.4

За свако $1 \leq x \leq n$ *важи* $f(g(x)) \leq f(x)$*.*

Доказ. Функцијом g се први низ узастопних јединица на који се наилази здесна у запису броја x мења нулама, а нула испред њих мења у јединицу, која се затим применом функције f враћа на нулу. С друге стране, ако је тај низ јединица дужи од 1 применом функције f на број x остаје нека од тих јединица, па је $f(g(x)) < f(x)$. Ако је низ најдеснијих јединица дужине 1, добија се $f(x) = f(g(x))$.

Формално, важи да је $f(g(x)) = g(x) - b(g(x)) = x + b(x) - b(g(x))$ и $f(x) = x - b(x)$. Зато је довољно доказати да је $x + b(x) - b(g(x)) \leq x - b(x)$ тј. да је $2b(x) \leq b(g(x))$. Нека $b(x)$ има јединицу на позицији p. Број $2b(x)$ има јединицу на позицији $p + 1$ (бројано здесна). Број $g(x)$ сигурно има све нуле од позиције p до краја, па се његова последња јединица налази или на позицији $p + 1$ или лево од ње, на основу чега лако следи да је он већи или једнак од $2b(x)$, који има јединицу на позицији $p + 1$. □

Докажимо сада и главно тврђење.

Теорема 1.4.1

Низ вредности k*,* $g(k)$*,* $g(g(k))$*, итд. које су мање или једнаке од дужине низа* n *одређује све вредности* m *такве да важи* $k \in (f(m), m]$ *тј. такве да важи једнакост (1.1).*

Доказ. Једнакост (1.1) не може да важи за бројеве $m < k$, а сигурно важи за број $m = k$, јер је $f(k) < k$ када је $k > 0$ (а ми претпостављамо да је $1 \leq k \leq n$). Дакле прва позиција у дрвету коју треба ажурирати је позиција k.

За све бројеве $m > k$, сигурно важи десна неједнакост и једино је потребно утврдити да ли важи лева тј. да ли је $f(m) < k$. На основу примене леме 1.4.3 на број k знамо да је $g(k)$ најмањи број m строго већи од k за који важи $f(m) < k$.

Даље, на основу примене леме 1.4.3 на број $g(k)$ знамо да је најмањи број већи од $g(k)$ за који важи $f(m) < g(k)$ број $g(g(k))$. Он задовољава услов (1.1). Заиста, важи $k < g(k) < g(g(k))$. На основу леме 1.4.4 примењене на $g(k)$ важи је $f(g(g(k))) \leq f(g(k)) < k$. Број $g(g(k))$ је и најмањи број који већи од $g(k)$ који задовољава услов (1.1). Ако би неки број m између $g(k)$ и $g(g(k))$ задовољио услов (1.1), важило би да је $f(m) < k < g(k)$, што је у супротности са тиме да је $g(g(k))$ најмањи број m већи од $g(k)$ такав да је $f(m) < g(k)$.

Доказ се даље наставља и завршава по потпуно истом принципу (што се може формализовати индукцијом). □

Претходна теорема гарантује да су једине позиције које треба ажурирати у дрвету приликом ажурирања елемента на позицији k у полазном низу управо позиције из серије k, $g(k)$, $g(g(k))$, итд., све док су оне мање или једнаке n, па су претходни поступак и његова имплементација коректни.

Формирање Фенвиковог дрвета

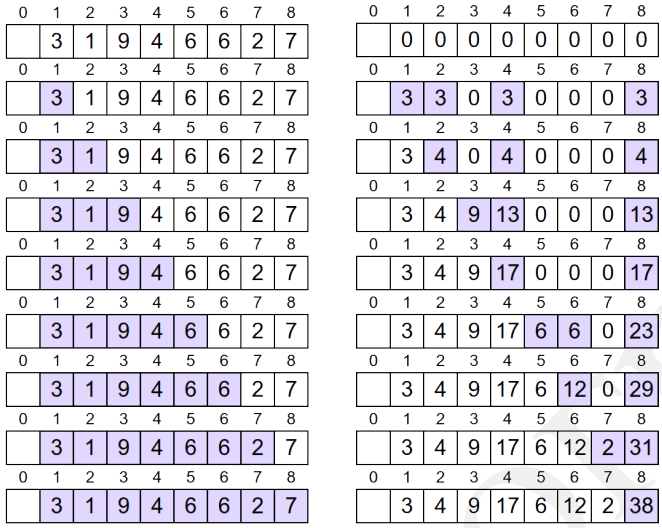
Остаје још питање како иницијално формирати Фенвиково дрво. Формирање се може свести на то да се креира дрво попуњено само нулама, а да се затим увећава вредност једног по једног елемента низа претходном функцијом.

```
// na osnovu niza a u kom su elementi smešteni
// na pozicijama iz segmenta [1, n] formira Fenvikovo drvo
// i smešta ga u niz drvo (na pozicije iz segmenta [1, n])
vector<int> formirajDrvo(int n, const vector<int>& a) {
  vector<int> drvo(n+1);
  fill_n(drvo + 1, n, 0);
  for (int k = 1; k <= n; k++)
     uvecaj(drvo, n, k, a[k]);
  return drvo;
}
```

Пример 1.4.17

Размотримо проблем формирања Фенвиковог дрвета за низ вредности $3, 1, 9, 4, 6, 6, 2, 7$. *Кренувши од низа који садржи само нуле, увећавамо један по један елемент низа, како је приказано на слици 1.21.*

Ова имплементација n пута позива функцију `uvecaj` која је сложености $O(\log n)$, те је укупна сложеност овог алгоритма $O(n \log n)$. Међутим, показује се да можемо и боље од овог. Наиме, сваки елемент Фенвиковог дрвета садржи збир елемената неког



Слика 1.21: Формирање Фенвиковог дрвета. Оригинални низ је представљен лево, а Фенвиково дрво десно.

сегмента полазног низа, па за израчунавање елемената Фенвиковог дрвета можемо искористити збирове префикса. Наиме, елемент на позицији k у Фенвиковом дрвету добијамо као разлику збира префикса $(0, k]$ и збира префикса $(0, f(k)]$, ако је $f(k) > 0$, док ако је $f(k) = 0$ елемент на позицији k биће једнак баш збиру префикса $(0, k]$.

```
vector<int> formirajDrvoPrefiksneSume(int n, const vector<int>& a) {
  vector<int> drvo(n+1);
  vector<int> sume_prefiksa(n+1,0);
  sume_prefiksa[1] = a[1];
  for (int k = 2; k <= n; k++)
    sume_prefiksa[k] = sume_prefiksa[k-1] + a[k];

  fill_n(drvo + 1, n, 0);
  for (int k = 1; k <= n; k++) {
    // racunamo f(k)
    int f_k = k - (k & -k);
    // od sume prefiksa do k oduzimamo sumu prefiksa do f(k)
    // ako je f(k)=0, onda ne vrsimo oduzimanje
    drvo[k] = sume_prefiksa[k] - (f_k > 0 ? sume_prefiksa[f_k] : 0);
  }
```

```
  return drvo;
}
```

Израчунавање збирова свих префикса полазног низа је временске сложености $O(n)$, али се након тога елементи Фенвиковог дрвета рачунају у времену $O(1)$, те је укупна временска сложеност овог приступа $O(n)$. Ова имплементација пак захтева додатни меморијски простор величине $O(n)$ за смештање префиксних збирова.

Приметимо да су за израчунавање k-тог елемента Фенвиковог дрвета потребни само елементи низа префиксних збирова на позицијама $j \leq k$. Ово опажање омогућава да користимо само један низ који ћемо иницијализовати на низ префиксних збирова, а затим почев од последњег елемента низа идући ка првом елементу мењати елемент по елемент са префиксног збира на елемент Фенвиковог дрвета.

```
vector<int> formirajDrvoPrefiksneSumeOpt(int n, const vector<int>& a) {
  vector<int> drvo(n+1);
  fill_n(drvo+1, n, 0);

  drvo[1] = a[1];
  for (int k = 2; k <= n; k++)
    drvo[k] = drvo[k-1] + a[k];

  for (int k = n; k >= 1; k--){
    int f_k = k - (k & -k);
    if (f_k > 0) drvo[k] -= drvo[f_k];
  }
  return drvo;
}
```

Ова имплементација алгоритма за формирање Фенвиковог дрвета је временске сложености $O(n)$, а додатне просторне сложености $O(1)$.

Задатак: Максимални подсегмент

Дат је низ целих бројева. Написати програм који омогућава извршавање следеће две врсте упита:

- Израчунавање максималног збира неког подсегмента датог сегмента одређеног позицијама $[a, b]$ (подсегмент је или празан или је одређен позицијама $[a', b']$ таквим да је $a \leq a' \leq b' \leq b$).
- Промена вредности елемента низа на некој датој позицији.

Опис улаза

Са стандардног улаза се учитава вредност n ($1 \leq n \leq 10^5$), а затим n целих бројева из интервала $[-100, 100]$, који представљају почетне вредности низа. Након тога се учитава природан број q ($1 \leq q \leq 10^5$) који представља број упита које је потребно обрадити. Након тога се учитава q упита. Упити могу бити следећег облика:

- `s a b` – потребно је одредити максималну вредност збира неког подсегмента сегмента одређеног позицијама $[a, b]$, за $0 \leq a \leq b < n$.
- `p k v` – потребно је елементу низа на позицији k доделити вредност v, за $0 \leq k < n$ и цео број v између -100 и 100.

Опис излаза

За сваки упит типа `s` на стандардни излаз исписати максималну вредност збира датог подсегмента.

Пример

Улаз	*Излаз*
`9`	`6`
`-2 1 -3 4 -1 2 1 -5 4`	`4`
`5`	`8`
`s 0 8`	`8`
`s 1 4`	
`p 4 1`	
`s 0 8`	
`s 2 6`	

Решење

Каданов алгоритам

Решење грубом силом подразумева да за сваки сегмент из почетка рачунамо вредност максималног збира неког његовог подсегмента. Чак и када се за то користи неки ефикасан алгоритам (на пример Каданов), ово решење је веома неефикасно. Ажурирање вредности низа врши се у времену $O(1)$, али се максимални подсегмент одређује у времену $O(n)$, па је сложеност најгорег случаја $O(qn)$.

Сегментно дрво

Ефикасно решење се може добити помоћу сегментног дрвета. Ако се неки сегмент подели на два мања подсегмента, његов максимални подсегмент може бити цео садржан у левом подсегменту, цео садржан у десном подсегменту или се састојати од максималног суфикса левог и максималног префикса десног подсегмента. Максимални префикс целог сегмента је или максимални префикс његовог левог подсегмента или садржи цео леви подсегмент и максимални префикс десног подсегмента. Слично, максимални су-

фикс целог сегмента је или максимални суфикс његовог десног подсегмента или садржи цео десни подсегмент и максимални суфикс левог подсегмента. Зато сегментно дрво организујемо тако да сваки чвор садржи наредна 4 податка:

- збир свих елемената сегмента који одговара том чвору;
- максимални збир неког подсегмента сегмента који одговара том чвору;
- максимални збир префикса сегмента који одговара том чвору;
- максимални збир суфикса сегмента који одговара том чвору.

Формирање сегментног дрвета је сложености $O(n)$. Одређивање максималног збира подсегмента за дати сегмент захтева један пролаз кроз дрво и сложености је $O(\log n)$. Такође, и ажурирање елемента захтева један пролаз кроз дрво (одоздо навише) и сложености је $O(\log n)$. Дакле, сложеност овог приступа је $O(n + q \log n)$.

```
// cvor segmentnog drveta
typedef struct {
  int zbir;        // zbir segmenta
  int maksSegment; // maksimalni zbir podsegmenta
  int maksPrefiks; // maksimalni zbir prefiksa
  int maksSufiks;  // maksimalni zbir sufiksa
} Cvor;

// sve 4 vrednosti su inicijalizovane na nulu
Cvor nula = {};

// odredjuje cvor roditelja na osnovu poznatog levog i desnog deteta
Cvor kombinuj(const Cvor& l, const Cvor& d) {
  Cvor c;
  c.zbir = l.zbir + d.zbir;
  c.maksSegment = max({l.maksSegment, d.maksSegment,
                       l.maksSufiks + d.maksPrefiks});
  c.maksPrefiks = max({l.maksPrefiks, l.zbir + d.maksPrefiks});
  c.maksSufiks  = max({d.maksSufiks, d.zbir + l.maksSufiks});
  return c;
}

// kreira se list drveta na osnovu date vrednosti v
Cvor list(int v) {
  Cvor c;
  c.zbir = v;
```

```
  c.maksSegment = max(v, 0);
  c.maksPrefiks = max(v, 0);
  c.maksSufiks = max(v, 0);
  return c;
}

// formira segmentno drvo na osnovu datog niza
vector<Cvor> formirajDrvo(const vector<int>& a) {
  int n = stepenDvojke(a.size());
  vector<Cvor> drvo(2*n, nula);
  for (int k = 0; k < a.size(); k++)
    drvo[n + k] = list(a[k]);

  for (int k = n-1; k > 0; k--)
    drvo[k] = kombinuj(drvo[2*k], drvo[2*k+1]);
  return drvo;
}

// rekurzivna funkcija koja odredjuje doprinos cvora k u segmentnom
// drvetu (koji pokriva segment [x, y] u originalnom nizu) maksimalnom
// zbiru podsegmenta segmenta [a, b]
Cvor maksPodsegment(const vector<Cvor>& drvo,
                    int k, int x, int y, int a, int b) {
  // segmenti [x, y] i [a, b] su disjunktni pa ne doprinosi nista
  if (b < x || a > y)
    return nula;
  // segment [x, y] je ceo sadrzan unutar [a, b] pa se ceo uracunava
  if (a <= x && y <= b)
    return drvo[k];
  // segmenti [x, y] i [a, b] se preklapaju, pa se segment [x, y] deli
  // na dve polovine
  int s = (x + y) / 2;
  return kombinuj(maksPodsegment(drvo, 2*k, x, s, a, b),
                  maksPodsegment(drvo, 2*k+1, s+1, y, a, b));
}

// na osnovu segmentnog drveta odre]uje se maksimalni z
int maksPodsegment(const vector<Cvor>& drvo, int a, int b) {
  // pozivamo pomocnu rekurzivnu funkciju krenuvsi od korena drveta
```

```
  // koji se nalazi u cvoru 1 i pokriva ceo niz tj. segment [0, n-1]
  int n = drvo.size() / 2;
  Cvor c = maksPodsegment(drvo, 1, 0, n-1, a, b);
  return c.maksSegment;
}

// nakon upisivanja vrednosti v na poziciju k u originalnom nizu
// azurira se segmentno drvo
void azurirajDrvo(vector<Cvor>& drvo, int k, int v) {
  int n = drvo.size() / 2;
  // azuriramo list
  drvo[n+k] = list(v);
  // azuiramo sve njegove pretke
  int r = (n+k) / 2;
  while (r > 0) {
    drvo[r] = kombinuj(drvo[2*r], drvo[2*r+1]);
    r = r / 2;
  }
}
```

Задатак: Број инверзија

Напиши програм који одређује колико у низу има инверзија (позиција $0 \leq i < j < n$, таквих да је $a_i > a_j$).

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 10^5$) и затим n целих бројева, сваки у посебном реду.

Опис излаза

На стандардни излаз исписати само тражени број инверзија.

Пример

Улаз	*Излаз*
5 3 1 4 2 5	3

Решење

Један начин да се задатак ефикасно реши је да одржавамо структуру података у коју се ефикасно може уметнути нови елемент и којој се ефикасно може добити одговор на

питањем колико је елемената у структури строго мање од дате вредности. Ако имамо овакву структуру података на располагању, инверзије можемо пребројати на следећи начин. Низ у коме бројимо инверзије обилазимо са десног краја, додајући у структуру један по један елемент, након што га обрадимо. То значи да ће у тренутку обраде било ког елемента структура садржати тачно оне елементе који су у оригиналном низу десно од њега. За сваки елемент проверавамо колико је елемената у структури мање од њега, увећавамо бројач инверзија за тај број и након тога умећемо елемент у дрво.

Једна таква структура је Фенвиково дрво над низом који на позицији x чува број појављивања вредности x у структури. Бројање елемената мањих од x се онда своди на израчунавање префиксне суме до $x-1$, а додавање елемента x се своди на увећавање вредности на позицији x за 1 (и ажурирање одговарајућих збирова у дрвету). Напоменимо да се у описаном алгоритму низ обилази здесна налево, јер нам Фенвиково дрво једноставније даје одговор на то колико је елемената мање од дате вредности, него колико је елемената веће од дате вредности.

Пошто величина Фенвиковог дрвета (па и меморијска, али и временска сложеност операција) зависи од вредности највећег елемента у њему, а нама за број инверзија нису важне апсолутне вредности елемената, него само њихов међусобни однос, пре примене алгоритма можемо сваки елемент заменити са његовим рангом у сортираном низу (исти елементи могу да имају исти ранг). Ово је могуће урадити, на пример, тако што сортирамо низ, а затим бинарном претрагом за сваки елемент пронађемо прву позицију његовог појављивања у сортираном низу.

```
long long brojInverzija(vector<int>& a) {
  int n = a.size();
  // svaki element u nizu a menjamo rangom tj. prvom pozicijom na
  // kojoj se javlja u sortiranom redosledu (brojimo pozicije od 1)
  vector<int> b = a;
  sort(begin(b), end(b));
  for (int i = 0; i < n; i++)
    a[i] = distance(begin(b), lower_bound(begin(b), end(b), a[i])) + 1;

  // Fenvikovo drvo
  vector<int> drvo(n+1, 0);
  // broj inverzija
  long long broj = 0;
  for (int i = n-1; i >= 0; i--) {
    // odredjujemo koliko elemenata u drvetu je strogo manje od a[i]
    broj += zbirPrefiksa(drvo, a[i]-1);
```

```
    // dodajemo a[i] u drvo
    dodaj(drvo, a[i], 1);
  }
  return broj;
}
```

Задатак: К-ти парни број

Написати програм који омогућава да се у низу природних бројева који је на почетку испуњен нулама, али чији се елементи често мењају током извршавања програма ефикасно проналази позиција k-тог парног броја по реду.

Опис улаза

Са стандардног улаза се учитава дужина низа n ($1 \leq n \leq 50000$), а затим и број упита m ($1 \leq m \leq 50000$). Извршавањем упита облика $u\ p\ x$ се у низ на позицију $1 \leq p \leq n$ уписује број x, док се упитом $c\ k$ на стандардни излаз исписује позиција k-тог парног броја у текућем садржају низа (позиције се броје од 1).

Опис излаза

На стандардном излазу приказати резултате извршавања упита `c`. Ако у неком случају у низу има мање парних бројева од вредности k, тада уместо позиције исписати `-`.

Пример

Улаз	*Излаз*

`5`
`8`
`u 3 1`
`c 4`
`u 1 7`
`u 2 5`
`c 1`
`u 1 2`
`c 2`
`c 4`
	`5`
	`4`
	`4`
	`-`

Објашњење

- Низ на почетку садржи 5 нула
- Након извршавање упита `u 3 1` низ садржи елементе `0 0 1 0 0`.
- Упитом `c 4` се израчунава позиција четвртог парног броја у низу и то је 5.
- Након извршавања упита `u 1 7`, па затим и `u 2 5` садржај низа је `7 5 1 0 0`.
- Упитом `c 1` се израчунава позиција првог парног броја у низу и то је 4.
- Након извршавања упита `u 1 2` садржај низа је `2 5 1 0 0`.
- Упитом `c 2` се израчунава позиција другог парног броја у низу и то је 4.
- Упитом `c 4` се израчунава позиција четвртог парног броја у низу. Он не постоји (низ садржи 3 парна броја: 2, 0 и 0).

Решење

Директно решење подразумева да се елементи уписују у низ и да се позиција k-тог парног одређује применом линеарне претраге. Ако имамо m_1 операција ажурирања и m_2 операција претраге сложеност таквог решења је $O(m_1 + m_2 \cdot n)$, што може бити прилично неефикасно.

Можемо формирати низ нула и јединица такав да се јединице налазе на месту парних елемената у полазном низу. Тада је позиција k-тог по реду парног броја најмања позиција таква да је збир свих јединица закључно са том позицијом једнак k. Ако низ нула и јединица одржавамо у Фенвиковом или сегментном дрвету, врло ефикасно можемо да израчунавамо збирове сваког фиксираног префикса. Захваљујући чињеници да су збирови префикса монотоно неопадајући (када се префикси продужавају), тражену позицију можемо ефикасно одредити алгоритмом бинарне претраге. Ако имамо m_1 операција ажурирања и m_2 операција претраге укупна сложеност ће бити $O(m_1 \log n + m_2 \log^2 n)$, тј. $O(m \log^2 n)$.

```
// vraca prvu poziciju p tako da je zbir niza na pozicijama [1, p]
// veci ili jednak k
int prefiksK(const vector<int>& drvo, int k) {
  // poziciju pronalazimo binarnom pretragom po vrednosti zbira prefiksa
  int l = 1, d = drvo.size() - 1;
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (zbirPrefiksa(drvo, s) < k)
      l = s + 1;
    else
      d = s - 1;
  }
  return l;
}

int main() {
  int n;
  cin >> n;
  // gradimo Fenvikovo drvo i inicijalizujemo ga jedinicama
  vector<int> drvo(n + 1, 0);
  for (int k = 1; k <= n; k++)
    dodaj(drvo, k, 1);
  // elementi niza
```

```
  vector<int> niz(n + 1, 0);

  // broj upita
  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    char c;
    cin >> c;
    if (c == 'u') {
      // upit upisa elementa u niz
      int p, x;
      cin >> p >> x;
      if (x % 2 == 0 && niz[p] % 2 != 0)
        // upisan je novi paran element na poziciju p
        dodaj(drvo, p, 1);
      else if (x % 2 != 0 && niz[p] % 2 == 0)
        // upisan je novi neparan element na poziciju p
        dodaj(drvo, p, -1);
      niz[p] = x;
    } else if (c == 'c') {
      // upit odredjivanja k-tog parnog elementa
      int k;
      cin >> k;
      // trazimo najkraci prefiks ciji je zbir jednak k
      int p = prefiksK(drvo, k);
      // proveravamo da li takav prefiks zaista postoji
      if (p <= n)
        cout << p << "\n";
      else
        cout << "-" << "\n";
    }
  }
  return 0;
}
```

Задатак: Број различитих елемената у сегментима

Напиши програм који одређује број различитих елемената у сваком од m датих сегмената низа од n елемената.

Опис улаза

Са стандардног улаза се учитава број n ($1 \leq n \leq 50000$), а затим n целих бројева $a_1, \ldots, a_n$, чије су вредности између 1 и 100000. Након тога се учитава број m ($1 \leq m \leq 50000$) а затим у наредних m редова лева и десна граница затвореног сегмента $[l_i, d_i]$, раздвојене са по једним размаком ($1 \leq l_i \leq d_i \leq n$).

Опис излаза

На стандардни излаз исписати m бројева од којих сваки представља број различитих елемената у сегменту $a_{l_i}, \ldots, a_{d_i}$.

Пример

Улаз	*Излаз*
5	3
1 1 2 1 3	2
3	3
1 5	
2 4	
3 5	

Решење

Решење грубом силом подразумева да за сваки сегмент избројимо различите елементе (на пример, коришћењем структуре података скуп), што је прилично неефикасно.

Основна идеја ефикасног решења задатка је да је број различитих елемената низа једнак броју последњих појављивања тих елемената. Можемо креирати бинарни низ који садржи јединице на местима на којима се елемент јавља последњи пут и нуле на местима на којима се налазе елементи који се у том сегменту јављају и касније и број различитих елемената тог низа добити сабирањем тог бинарног низа.

Пример 1.4.18

На пример, ако је дато 10 *елемената низа* $3, 4, 1, 3, 2, 5, 4, 7, 2, 2$*, тада можемо направити бинарни низ* $0, 0, 1, 0, 0, 1, 1, 1, 0, 1$*. Збир његових елемената је* 5*, што значи да у оригиналном низу постоји* 5 *различитих елемената.*

Додатно, за сваки сегмент позиција облика $[l, n)$, тј. за сваки суфикс низа, број различитих елемената у том суфиксу можемо добити израчунавањем броја јединица у одговарајућем суфиксу бинарног низа. Заиста, за сваку групу једнаких елемената постоји само једна јединица и сваки елемент се броји само једном. За сваки елемент који припада суфиксу одређеном позицијама $[l, n)$ постоји бар једна јединица у том сегменту која му одговара (ако је тренутно појављивање елемента последње, онда је јединица на

његовом месту, а ако није, онда сигурно постоји јединица негде иза њега која му одговара). Приметимо да то не мора да важи за сегменте који се не завршавају на последњој позицији у низу.

Пример 1.4.19

Збир последња три елемента у бинарном низу из претходног примера је 2*, што значи да у оригиналном низу постоје* 2 *различита елемента (то су* 7 *и* 2*).*

Бинарни низ можемо креирати инкрементално током проласка полазног низа слева надесно. Наиме, за сваки нови елемент полазног низа на крај низа додајемо јединицу (ово његово појављивање је сигурно последње). Додатно, проверавамо да ли се тај елемент раније појављивао и ако јесте проналазимо позицију његовог ранијег последњег појављивања и мењамо је у нулу. Позиције последњих појављивања елемената можемо чувати у засебној мапи.

Ово нам указује на то да би добро било унете сегменте сортирати на основу десних крајева и обрађивати их у том редоследу. Низ нула и јединица можемо чувати у Фенвиковом или сегментном дрвету, што нам омогућава да ефикасно ажурирамо појединачне вредности и одређујемо збирове његових суфикса.

Пример 1.4.20

Прикажимо како се извршава пример из поставке задатка. Упите обрађујемо у сортираном редоследу на основу десног краја.

- *Прво се обрађује упит* $[2, 4]$*. То значи да се креира бинарни низ закључно са позицијом* 4 *из оригиналног низа.*
  - *Додајемо елемент* 1 *(са позиције* 1*) и бинарни низ је* $[1]$*.*
  - *Додајемо елемент* 1 *(са позиције* 2*) и бинарни низ постаје* $[0, 1]$*.*
  - *Додајемо елемент* 2 *(са позиције* 3*) и бинарни низ постаје* $[0, 1, 1]$*.*
  - *Додајемо елемент* 1 *(са позиције* 4*) и бинарни низ постаје* $[0, 0, 1, 1]$*.*

  Упит се извршава сабирајући елементе са позиција $[2, 4]$ *у бинарном низу, чиме се добија резултат* 2*.*
- *Сада се обрађује упит* $[1, 5]$*. Ово захтева да се у бинарни низ дода и елемент са позиције 5 из оригиналног низа.*
  - *Додајемо елемент* 3 *(са позиције* 4*) и бинарни низ постаје* $[0, 0, 1, 1, 1]$*.*

  Упит се изршава сабирајући елементе са позиција $[1, 5]$ *у бинарном низу, чиме се добија резултат* 3*.*

- *На крају се обрађује упит* $[3, 5]$. *Бинарни низ је већ проширен до позиције* 5.

  Упит се извршава сабирајући елементе са позиција $[3, 5]$ *у бинарном низу, чиме се добија резултат* 3.

На крају исписујемо резултате упита у редоследу у ком су унети (а не у редоследу у ком су обрађени).

Приметимо да смо одговарање на упите одложили за крај програма, што нам је омогућило да све упите учитамо и затим сортирамо. Ова техника се некада назива *офлајн обрада упита* (енгл. offline queries) и иако се понекада може употребити за ефикасно решење задатка, у реалним ситуацијама она није увек примењива (у неким применама се одговор на сваки упит тражи одмах чим се упит постави).

```
// ucitavamo elemente u niz a (od pozicije 1)
// ...
// ucitavamo broj upita i upite
struct Upit {
  int l, d, i;
};

int m;
cin >> m;
vector<Upit> upiti(m);
for (int i = 0; i < m; i++) {
  cin >> upiti[i].l >> upiti[i].d;
  upiti[i].i = i;
}

// sortiramo upite po desnom kraju
sort(begin(upiti), end(upiti),
     [](const auto& u1, const auto& u2) {
       return u1.d < u2.d;
     });

// prethodna pozicija na kojoj se javlja data vrednost iz niza
unordered_map<int, int> prethodna_pozicija;
// Fenvikovo drvo
vector<int> drvo(n + 1, 0);
// za svaki upit se cuva rezultat upita
```

```
vector<int> rezultat(m);
int tekuci_upit = 0;
for (int i = 1; tekuci_upit < upiti.size() && i <= n; i++) {
  auto it = prethodna_pozicija.find(a[i]);
  if (it != prethodna_pozicija.end()) {
    dodaj(drvo, it->second, -1);
    it->second = i;
  } else {
    prethodna_pozicija[a[i]] = i;
  }
  dodaj(drvo, i, 1);
  while (tekuci_upit < upiti.size() && upiti[tekuci_upit].d == i) {
    rezultat[upiti[tekuci_upit].i] =
      zbirSegmenta(drvo, upiti[tekuci_upit].l, i);
    tekuci_upit++;
  }
}

// ispisujemo rezultate upita smestene u niz rezultat
// ...
```

1.4.3 Ажурирање сегмената

И сегментна и Фенвикова дрвета подржавају ефикасно израчунавање збирова одређених сегмената низа (енгл. range query), па самим тим и одређивање појединачних вредности низа (енгл. point query), као и ажурирање[13] појединачних елемената низа (енгл. point update), док ажурирање целих сегмената низа одједном (енгл. range update) није директно подржано. Наиме, ако се оно сведе на појединачно ажурирање свих елемената унутар сегмента, добија се лоша сложеност (у најгорем случају врши се n ажурирања која имају појединачну сложеност $O(\log n)$, па је укупна сложеност $O(n \log n)$). Са друге стране, чување низа разлика омогућава ефикасно ажурирање целих сегмената (енгл. range update), па самим тим и појединачних елемената (енгл. point update), али не и ефикасно одређивање појединачних елемената низа (енгл. point query) нити збирова сегмената (енгл. range query).

[13] Под ажурирањем се подразумева или постављање вредности елемента на дату вредност или увећање тренутне вредности елемента за дату вредност. У наставку ћемо разматрати само увећање тренутне вредности.

	point query	*range query*	*point update*	*range update*
Разлике суседних елемената	лоше / $O(n)$	лоше / $O(n)$	добро / $O(1)$	добро / $O(1)$
Префиксни збирови	добро / $O(1)$	добро / $O(1)$	лоше / $O(n)$	лоше / $O(n)$
Фенвиково дрво	добро / $O(\log n)$	добро / $O(\log n)$	добро / $O(\log n)$	лоше / $O(n \log n)$
Сегментно дрво	добро / $O(1)$	добро / $O(\log n)$	добро / $O(\log n)$	лоше / $O(n \log n)$

У наставку ћемо видети неколико начина да се направе структуре које ефикасно подржавају све наведене операције.

1.4.3.1 Дрво над низом разлика

Проблем

Дефинисати структуру података која обезбеђује ефикасно ажурирање сегмената датог низа одређених позицијама $[a, b]$ *(*range update*), па самим тим и ажурирање појединачних елемената низа (*point update*) увећавањем свих елемената за дату вредност, као и ефикасно одређивање вредности појединачних елемената низа (*point query*).*

Приметимо да се не захтева могућност ефикасног одређивања статистика сегмената (*range query*).

Основна идеја решења је да се одржава низ разлика суседних елемената полазног низа и да се тај низ разлика чува у Фенвиковом дрвету (или, аналогно, у сегментном дрвету).

- Увећавање свих елемената сегмената полазног низа за неку вредност v, своди се на промену два елемента низа разлика, што се уз одржавање дрвета може урадити у времену $O(\log n)$.
- Реконструкција елемента полазног низа на основу низа разлика своди се на израчунавање збира одговарајућег префикса, што се помоћу Фенвиковог дрвета може урадити веома ефикасно, у времену $O(\log n)$.

На овај начин, и ажурирање целих сегмената низа одједном и очитавање појединачних елемената можемо постићи алгоритмима сложености $O(\log n)$, што није било могуће само уз коришћење низа разлика (увећавања свих елемената неког сегмента је тада било сложености $O(1)$, али је очитавање вредности из низа било сложености $O(n)$).

Дакле, ако Фенвиково или сегментно дрво изградимо над низом разлика уместо над оригиналним низом, губимо могућност ефикасног израчунавања статистика сегмената (*range query*), али добијамо могућност ажурирања сегмената (*range update*).

	point query	*range query*	*point update*	*range update*
Дрво разлика	добро / $O(\log n)$	лоше / $O(n \log n)$	добро / $O(\log n)$	добро / $O(\log n)$

1.4.3.2 Два дрвета разлика

На крају, описаћемо неколико структура података које омогућавају ефикасно извршавање све четири врсте упита.

Проблем

Дефинисати структуру података која обезбеђује ефикасно ажурирање сегмената датог низа одређених позицијама $[a, b]$ (самим тим и ажурирање појединачних елемената низа) увећавањем свих елемената за дату вредност, као и ефикасно одређивање збирова сегмената одређених позицијама $[a, b]$ (самим тим и одређивање вредности појединачних елемената низа).

Ефикасно увећање свих елемената у датом сегменту за исту вредност и израчунавање збирова сегмената могуће је имплементирати одржавањем два Фенвикова или сегментна дрвета.

Прво од њих ће чувати разлике оригиналног низа и омогућиће нам да ефикасно увећавамо сегменте и израчунавамо појединачне елементе оригиналног низа.

Ако је оригинални низ иницијализован нулама, након увећања свих елемената са позиција из сегмента $[a, b]$ за вредност v добијамо низ:

			a				b			
0	…	0	v	v	…	v	v	0	…	0

који се може представити низом разлика:

			a				b	$b+1$		
0	…	0	v	0	…	0	0	$-v$	…	0

Сабирањем префикса низа разлика можемо добити било који појединачни елемент почетног низа (ако је низ разлика у Фенвиковом или сегментном дрвету, сложеност овог поступка је $O(\log n)$).

Циљ нам је да израчунамо префиксне збирове оригиналног низа (њега не можемо чувати у Фенвиковом или сегментном дрвету, зато што нам је потребно ажурирање његових сегмената, а не појединачних елемената).

Префиксни збирови оригиналног низа су:

			a				b			
0	…	0	v	$2v$	…	$(b-a)v$	$(b-a+1)v$	$(b-a+1)v$	…	$(b-a+1)v$

- Префиксни збирови P_k на позицијама k лево од позиције a (тј. $k < a$) једнаки су нули (јер су вредности A_k оригиналног низа лево од те позиције једнаки нули).
- Префиксни збирови P_k на позицијама k између a и b (тј. $a \leq k \leq b$) једнаки су $(k-a+1) \cdot v$, где је $v = A_k$ вредност која се налази на позицији k у оригиналном низу.
- Префиксни збирови P_k на позицијама k десно од позиције b (тј. $k > b$) једнаки су $(b - a + 1) \cdot v$.

Дакле, важи:

$$P_k = \sum_{i=1}^{k} A_i = \begin{cases} 0 & k < a \\ (k - (a-1)) \cdot v & a \leq k \leq b \\ (b - a + 1) \cdot v & k > b \end{cases}$$

Кључна идеја је да упоредимо ове тражене збирове са вредностима низа који на свакој позицији k садржи вредност $k \cdot A_k$, тј. да размотримо разлике $X_k = kA_k - P_k$.

- У првом случају (када је $k < a$) је $A_k = P_k = 0$, па је и $X_k = 0$.
- У другом случају (када је $a \leq k \leq b$) је $P_k = (k - a + 1) \cdot A_k$, па је $X_k = kA_k - (k - a + 1)A_k = (a-1)A_k$.
- У трећем случају (када је $k > b$) је $P_k = (b - a + 1) \cdot v$, док је $A_k = 0$, па је $X_k = -(b - a + 1) \cdot v$.

Другим речима, сваки збир префикса P_k смо разложили на разлику $kA_k - X_k$:

$$P_k = \sum_{i=1}^{k} A_i = k \cdot A_k - X_k = \begin{cases} k \cdot 0 - 0 & k < a \\ k \cdot v - (a-1) \cdot v & a \leq k \leq b \\ k \cdot 0 - (-(b - a + 1)) \cdot v & k > b \end{cases}$$

Низ X_k, дакле, након ажурирања оригиналног низа има следеће вредности:

			a			b			
0	…	0	$(a-1)v$	…	$(a-1)v$	$(a-1)v$	$-(b-a+1)v$	…	$-(b-a+1)v$

Ако бисмо у сваком тренутку познавали вредности у низу X_k, тада бисмо вредности P_k лако могли израчунати помоћу формуле $P_k = kA_k - X_k$. Вредности A_k лако израчунавамо помоћу Фенвиковог или сегментног дрвета у коме чувамо разлике тог низа. Међутим, и низ X_k је такав да му се при сваком ажурирању повезани сегменти увећавају тј. умањују за исту вредност, тако да се и он може лако реконструисати ако би се његове разлике чувале у другом Фенвиковом или сегментном дрвету. Наиме, низ разлика овог низа се мења овако:

			a				b	$b+1$			
0	…	0	$(a-1)v$	0	…	0	0	$-bv$	0	…	0

Дакле, одржавамо два дрвета: D_A у коме чувамо разлике низа A и D_X у коме чувамо разлике низа X. Прво иницијализујемо разликама низа A, а друго нулама.

Операција увећања свих елемената са позиција из $[a, b]$ за вредност v се своди на следеће операције над дрветима:

- *uvecaj*(D_A, a, v)
- ако је $b < n$ онда *uvecaj*$(D_A, b+1, -v)$
- *uvecaj*$(D_X, a, (a-1)v)$
- ако је $b < n$ онда *uvecaj*$(D_X, b+1, -bv)$

Операција израчунавање вредности низа A на позицији k тј. вредности A_k се своди на израчунавање префиксног збира првих k елемената Фенвиковог дрвета D_A.

- $A_k =$ *zbir_prefiksa*(D_A, k)

Операција израчунавања збира префикса елемената низа A дужине k се своди на израчунавање вредности

- $P_k = kA_k - X_k = k \cdot$ *zbir_prefiksa*$(D_A, k) -$ *zbir_prefiksa*(D_X, k)

Операција израчунавања зира сегмента $[a, b]$ низа A се своди на израчунавање вредности:

- $S_{ab} = P_b - P_{a-1}$

При том је $P_0 = 0$.

Дакле, на овај начин можемо постићи добру сложеност за сва 4 типа операција.

	point query	*range query*	*point update*	*range update*
Два дрвета разлика	добро / $O(\log n)$	добро / $O(\log n)$	добро / $O(\log n)$	добро / $O(\log n)$

Пример 1.4.21

Прикажимо један пример употребе ове структуре података. Претпоставимо да низ A има 10 елемената и да су у почетку сви једнаки нули. У програму би се одржавала само дрвета над низовима D_A и D_X, а илустрације ради ми ћемо приказати и садржај низа A, његових префиксних сума P и помоћног низа X. Елементи оригиналног низа (као и низова разлика) смештени су на позицијама од 1 до 10.

	0	1	2	3	4	5	6	7	8	9	10
D_A	-	0	0	0	0	0	0	0	0	0	0
D_X	-	0	0	0	0	0	0	0	0	0	0
A	-	0	0	0	0	0	0	0	0	0	0
kA_k	-	0	0	0	0	0	0	0	0	0	0
X	-	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0

Увећајмо елемент A_5 за 4. То се може свести на увећавање елемената сегмента $[a, b] = [5, 5]$ за $v = 4$.

	0	1	2	3	4	5	6	7	8	9	10
D_A	-	0	0	0	0	4	-4	0	0	0	0
D_X	-	0	0	0	0	16	-20	0	0	0	0
A	-	0	0	0	0	4	0	0	0	0	0
kA_k	-	0	0	0	0	20	0	0	0	0	0
X	-	0	0	0	0	16	-4	-4	-4	-4	-4
P	0	0	0	0	0	4	4	4	4	4	4

Умањимо све елементе низа A за 1. То се може свести на увећавање елемената сегмента $[a, b] = [1, 10]$ за $v = -1$.

	0	1	2	3	4	5	6	7	8	9	10
D_A	-	-1	0	0	0	4	-4	0	0	0	0
D_X	-	0	0	0	0	16	-20	0	0	0	0
A	-	-1	-1	-1	-1	3	-1	-1	-1	-1	-1
kA_k	-	-1	-2	-3	-4	15	-6	-7	-8	-9	-10
X	-	0	0	0	0	16	-4	-4	-4	-4	-4
P	0	-1	-2	-3	-4	-1	-2	-3	-4	-5	-6

Одредимо $S_{[3,7]}$ тј. збир елемената низа у сегменту $[3, 7]$. Важи да је $S_{[3,7]} = P_7 - P_2$, па је потребно да одредимо P_7 и P_2.

P_7 се одређује тако што се помоћу Фенвиковог дрвета ефикасно одреди $A_7 =$ zbir_prefiksa$(D_A, 7) = -1$, затим $X_7 =$ zbir_prefiksa$(D_X, 7) = -4$ и на крају се израчуна $7A_7 - X_7 = 7 \cdot (-1) - (-4) = -3$.

P_2 се одређује тако што се помоћу Фенвиковог дрвета ефикасно одреди $A_2 =$ zbir_prefiksa$(D_A, 2) = -1$, затим $X_2 =$ zbir_prefiksa$(D_X, 2) = 0$ и на крају се израчуна $2A_2 - X_2 = 2 \cdot (-1) - 0 = -2$. Тражени збир сегмента је онда $-3 - (-2) = -1$. Јасно је да је то тачан резултат, јер се у том сегменту налазе елементи $[-1, -1, 3, -1, -1]$.

1.4.3.3 Лењо ажурирање сегментног дрвета

Опишимо сада још једно решење претходног проблема, тј. још једну структуру података која омогућава ефикасно извршавање све четири врсте упита. За разлику од претходне, ова структура омогућава и израчунавање неких других статистика (не само збира елемената).

Подсетимо се, сегментна дрвета омогућавају ефикасно ажурирање појединачних елемената и израчунавање збирова сегмената (самим тим и одређивање вредности појединачних елемената), међутим, не омогућавају ефикасно ажурирање свих елемената датог сегмента $[a, b]$ (тј. њихово увећање за дату вредност или њихову замену датом вредношћу). Ово се може постићи у времену $O(\log n)$ ако се на сегментно дрво примени техника *лење пропагације* (енгл. lazy propagation). Она је пример стратегије лењог извршавања којим се израчунавања одлажу све док одговарајуће вредности нису неопходне. Техника лење пропагације се доста ослања на рад са сегментним дрветом одозго наниже. Једноставности ради, технику ћемо представити кроз пример упита увећавања сегмената за дату вредност.

Сваки чвор у сегментном дрвету чува збир неког сегмента оригналног низа. Ако се тај сегмент у целости садржи унутар сегмента који се ажурира, можемо унапред изра-

чунати за колико се повећава вредност у том чвору. Наиме, ако се свака вредност у сегменту повећава за v, тада се вредност збира тог сегмента повећава за $k \cdot v$, где је k број елемената у том сегменту. Вредност збира у том чвору тиме бива ажурирана у константном времену, али вредности збирова унутар поддрвета којима је тај чвор корен (укључујући и вредности у листовима које одговарају вредностима полазног низа) и даље остају неажурне. Њихово ажурирање захтевало би линеарно време, што је недопустиво скупо. Кључна идеја је да се ажурирање тих вредности одложи и да се оне не ажурирају одмах, већ само када затребају током неког накнадног упита, тј. током неке касније посете тим чворовима (која се иначе врши у склопу тог каснијег упита, а не посебно у циљу ажурирања вредности). Наиме, не желимо да те чворове посећујемо само због овог ажурирања, већ ћемо ажурирање урадити успут, током неке друге посете тим чворовима која би се свакако морала десити.

Поставља се питање како да сигнализирамо да вредности збирова у неком поддрвету нису ажурне и додатно оставимо упутство на који начин их треба ажурирати. У том циљу у сваком од чворова поред вредности збира сегмента чувамо и додатни *коефицијент лење пропагације*. Ако дрво у свом корену има коефицијент лење пропагације c који је различит од нуле, то значи да вредности збирова у целом том дрвету нису ажурне и да је сваки од листова тог дрвета потребно повећати за c и у односу на то ажурирати и вредности збирова у свим унутрашњим чворовима тог дрвета (укључујући и корен). Ажурирање се може одлагати све док вредност збира у неком чвору не постане заиста неопходна, а то је тек приликом упита израчунавања вредности збира неког сегмента. Ипак, вредности збирова у чворовима ћемо ажурирати и чешће и то заправо приликом сваке посете чвору – било у склопу операције увећања вредности из неког сегмента позиција полазног низа, било у склопу упита израчунавања збира неког сегмента. На почетку обе рекурзивне функције можемо проверити да ли је вредност коефицијента лење пропагације текућег чвора различита од нуле и ако јесте, ажурирати вредност збира у том чвору тако што ћемо га увећати за производ тог коефицијента и броја елемената који тај чвор покрива, затим коефицијенте лење пропагације оба његова детета увећати за тај коефицијент, а коефицијент лење пропагације тог чвора поставити на нулу (тиме корен дрвета који тренутно посећујемо постаје ажуран, а његовим поддрветима се даје упутство како их у будућности ажурирати). На овај начин се избегава ажурирање целог дрвета одједном, већ се ажурира само корен, што је операција сложености $O(1)$.

Размотримо како се може ажурирати чвор дрвета након увећања свих елемената неког сегмента. Инваријанта је да сви чворови у дрвету или садрже ажурне вредности збирова или су исправно обележени за каснија ажурирања (преко коефицијента лење пропагације). Након процеса ажурирања сегмента, чвор које се тренутно ажурира ће садржати актуелну вредност збира. Након почетног обезбеђивања да коефицијент лење пропагације у текућем чвору постане нула, могућа су три следећа случаја.

- Ако је сегмент у текућем чвору дисјунктан у односу на сегмент који се ажурира, тада су сви чворови у поддрвету којем је он корен или већ ажурни или исправно обележени за касније ажурирање и није потребно ништа урадити.
- Ако је сегмент који одговара текућем чвору потпуно садржан у сегменту чији се елементи увећавају, тада се његова вредност ажурира (увећавањем за $k \cdot v$, где је k број елемената сегмента који одговара текућем чвору, а v вредност увећања), а његовој деци се коефицијент лење пропагације увећава за вредност v.
- На крају, ако се ова два сегмента секу, тада се прелази на рекурзивну обраду оба детета. Након извршавања функције над њима, сигурни смо да ће сви чворови у левом и десном поддрвету задовољавати услов инваријанте и да ће корени и левог и десног поддрвета имати ажурне вредности. Ажурну вредност у корену добијамо сабирањем вредности његова два детета.

Лењо сегментно дрво чуваћемо коришћењем два низа: `drvo`, у ком се чувају елементи сегментног дрвета и `lenjo`, у ком се чувају коефицијенти лење пропагације.

```
// ažurira elemente lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije k u kome se čuvaju zbirovi
// elemenata originalnog niza sa pozicija iz segmenta [x, y]
// nakon što su u originalnom nizu svi elementi sa pozicija iz
// segmenta [a, b] uvećani za vrednost v
void promeni(vector<int>& drvo, vector<int>& lenjo, int k, int x, int y,
             int a, int b, int v) {
   // ažuriramo vrednost u korenu, ako nije ažurna
   if (lenjo[k] != 0) {
      drvo[k] += (y - x + 1) * lenjo[k];
      // ako nije u pitanju list, propagaciju prenosimo na decu
      if (x != y) {
         lenjo[2*k] += lenjo[k];
         lenjo[2*k+1] += lenjo[k];
      }
      // vrednost u korenu je sad ažurna
      lenjo[k] = 0;
   }
   // ako su intervali disjunktni, ništa nije potrebno raditi
   if (b < x || y < a) return;
   // ako je interval [x, y] sadržan u intervalu [a, b]
   // vrsimo uvecanje cvora za k * v
   if (a <= x && y <= b) {
```

```
      drvo[k] += (y - x + 1) * v;
      // ako nije u pitanju list, propagaciju prenosimo na decu
      if (x != y) {
         lenjo[2*k] += v;
         lenjo[2*k+1] += v;
      }
   } else {
      // u suprotnom se intervali seku,
      // pa rekurzivno obilazimo poddrveta
      int s = (x + y) / 2;
      promeni(drvo, lenjo, 2*k, x, s, a, b, v);
      promeni(drvo, lenjo, 2*k+1, s+1, y, a, b, v);
      // azurnu vrednost u korenu dobijamo kao zbir vrednosti dece
      drvo[k] = drvo[2*k] + drvo[2*k+1];
   }
}

// ažurira elemente lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije 1 u kome se čuvaju zbirovi
// elemenata originalnog niza sa pozicija iz segmenta [0, n-1]
// nakon što su u originalnom nizu svi elementi sa pozicija iz
// segmenta [a, b] uvećani za vrednost v
void promeni(vector<int>& drvo, vector<int>& lenjo, int n,
             int a, int b, int v) {
   promeni(drvo, lenjo, 1, 0, n-1, a, b, v);
}
```

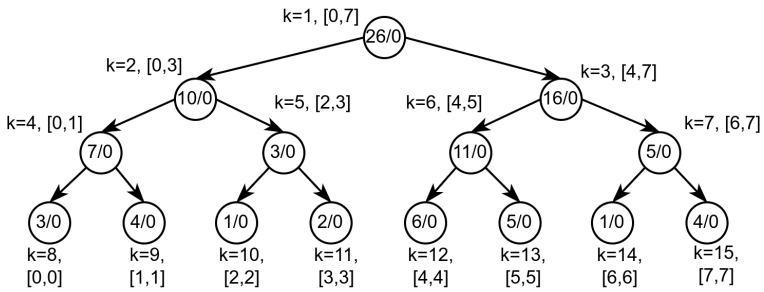
Сложеност функције ажурирања свих елемената датог сегмента у лењом сегментном дрвету износи $O(\log n)$. Наиме, као и у случају операције рачунања збира сегмента у сегментном дрвету, на сваком нивоу се разматра највише 4 чвора, те је максимални број чворова који се посећује једнак $4 \log n$.

Пример 1.4.22

Прикажимо рад ове функције на примеру лењог сегментног дрвета са слике 1.22.

Прикажимо како бисмо све елементе из сегмента $[2, 7]$ *увећали за* 3.

- *Крећемо од корена који покрива сегмент* $[0, 7]$. *Сегменти* $[0, 7]$ *и* $[2, 7]$ *се секу, па стога ажурирање препуштамо деци и након њиховог ажурирања, при повратку из рекурзије вредност корена одређујемо као збир ажурираних вредности његове деце.*



Слика 1.22: Лењо сегментно дрво: у сваком чвору чувамо вредности збира одговарајућег сегмента и коефицијента лење пропагације. Иницијално су вредности свих коефицијената лење пропагације једнаки нула.

– *На левој страни се сегмент* $[0, 3]$ *сече са сегментом* $[2, 7]$ *па и он препушта ажурирање наследницама и ажурира се тек при повратку из рекурзије.*

  – *Сегмент* $[0, 1]$ *је дисјунктан у односу на сегмент* $[2, 7]$ *и ту онда није потребно ништа радити.*

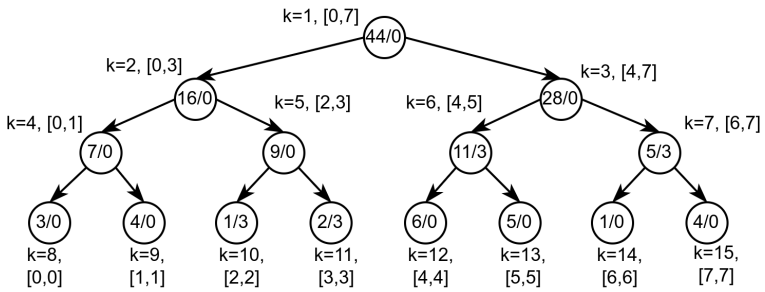
  – *Сегмент* $[2, 3]$ *је садржан у сегменту* $[2, 7]$*, и за њега директно знамо како се збир увећава: пошто овај чвор покрива два елемента и сваки се увећава за* 3*, збир овог сегмента се увећава укупно за* $2 \cdot 3 = 6$ *и поставља се на* 9*. У овом тренутку избегавамо ажурирање свих вредности у дрвету у ком је тај елемент корен, већ само наследницима уписујемо да је потребно пропагирати увећање за* 3*, али саму пропагацију одлажемо за тренутак када она постане неопходна.*

– *У повратку из рекурзије, вредност* 10 *ажурирамо и нова вредност је једнака* $7 + 9 = 16$*.*

– *Што се тиче десног поддрвета, сегмент* $[4, 7]$ *је садржан у сегменту* $[2, 7]$*, па можемо директно израчунати нову вредност збира у овом чвору. Наиме, пошто се* 4 *елемента увећавају за по* 3*, укупан збир се увећава за* $4 \cdot 3 = 12$*. Зато се вредност* 16 *мења у* $16 + 12 = 28$*. Пропагацију ажурирања кроз поддрво са кореном у овом чвору одлажемо и само његовој деци бележимо да је увећање за* 3 *потребно извршити у неком каснијем тренутку.*

• *При повратку из рекурзије вредност у корену ажурирамо са* 26 *на* $16 + 28 = 44$*.*

Након извршавања ових операција добија се дрво приказано на слици 1.23. Ово лењо сегментно дрво одговара низу $3, 4, 4, 5, 9, 8, 4, 7$*.*



Слика 1.23: Лењо сегментно дрво након ажурирања свих елемената сегмента $[2, 7]$ за вредност 3.

Претпоставимо да је у добијеном сегментном дрвету потребно елементе из сегмента $[0, 5]$ *увећавати за вредност* 2.

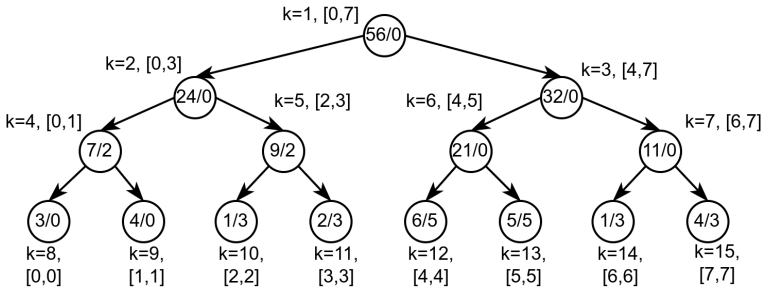
- *Поново се креће од корена лењог сегментног дрвета и када се установи да се сегмент* $[0, 7]$ *сече са сегментом* $[0, 5]$ *ажурирање се препушта деци и вредност у корену се ажурира тек при повратку из рекурзије.*
  - *Сегмент* $[0, 3]$ *је садржан у сегменту* $[0, 5]$*, па се вредност* 16 *увећава за* $4 \cdot 2 = 8$ *и поставља на* 24*. Поддрвета се не ажурирају одмах, већ се само њиховим коренима уписује да је све вредности потребно ажурирати за* 2.
  - *У десном поддрвету сегмент* $[4, 7]$ *се сече са* $[0, 5]$*, па се рекурзивно обрађују поддрвета.*
    - *При обради чвора са вредношћу* 11*, примећује се да је он требало да буде ажуриран јер је његов коефицијент лење пропагације различит од нула, међутим још није, па се најпре његова вредност ажурира и увећава за* $2 \cdot 3$ *и са* 11 *мења на* 17*. Његови наследници се не ажурирају одмах и њима се само уписује лења вредност* 3*.*

      Након тога се утврђује да се сегмент $[4, 5]$ *садржи у сегменту* $[0, 5]$*, па се вредност* 17 *увећава за* $2 \cdot 2 = 4$ *и поставља на* 21*. Поддрвета се не ажурирају одмах, већ само по потреби тако што се у њиховим коренима постави вредност лењог коефицијента. Пошто је у њима већ уписана вредност* 3*, она се сада увећава за* 2 *и поставља на* 5*.*
    - *Сада се прелази на обраду поддрвета у чијем је корену вредност* 5 *и пошто оно није ажурно, најпре се вредност* 5 *увећава за* $2 \cdot 3 = 6$ *и поставља на* 11*, а његовој деци се лењи коефицијент поставља на* 3*.*

Након тога се примећује да је сегмент $[6, 7]$ *дисјунктан са* $[0, 5]$ *и не ради се ништа.*

- *У повратку кроз рекурзију се ажурирају вредности родитељских чворова.*

Након ових операција добија се дрво приказано на слици 1.24. Ово лењо сегментно дрво одговара низу $5, 6, 6, 7, 11, 10, 4, 7$.



Слика 1.24: Лењо сегментно дрво након увећања свих елемената из сегмента $[0, 5]$ за 2.

Функција израчунавања вредности збира сегмента остаје практично непромењена, осим што се при уласку у сваки чвор врши његово ажурирање, ако је потребно. Стога и њена сложеност остаје непромењена и износи $O(\log n)$.

```
// na osnovu lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije k u kome se čuvaju zbirovi
// elemenata polaznog niza sa pozicija iz segmenta [x, y]
// izračunava se zbir elemenata polaznog niza
// sa pozicija iz segmenta [a, b]
int saberi(vector<int>& drvo, vector<int>& lenjo, int k, int x, int y,
           int a, int b) {
  // ažuriramo vrednost u korenu, ako nije ažurna
  if (lenjo[k] != 0) {
     drvo[k] += (y - x + 1) * lenjo[k];
     if (x != y) {
       lenjo[2*k] += lenjo[k];
       lenjo[2*k+1] += lenjo[k];
     }
     lenjo[k] = 0;
  }
```

```
  // intervali [x, y] i [a, b] su disjunktni
  if (b < x || a > y) return 0;
  // interval [x, y] je potpuno sadržan unutar intervala [a, b]
  if (a <= x && y <= b)
      return drvo[k];
  // intervali [x, y] i [a, b] se seku
  int s = (x + y) / 2;
  return saberi(drvo, lenjo, 2*k, x, s, a, b) +
         saberi(drvo, lenjo, 2*k+1, s+1, y, a, b);
}

// na osnovu lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije 1 u kome se čuvaju zbirovi
// elemenata polaznog niza sa pozicija iz segmenta [0, n-1]
// izračunava se zbir elemenata polaznog niza sa pozicija
// iz segmenta [a, b]
int saberi(vector<int>& drvo, vector<int>& lenjo, int n, int a, int b) {
  // računamo doprinos celog niza,
  // tj. elemenata iz intervala [0, n-1]
  return saberi(drvo, lenjo, 1, 0, n-1, a, b);
}
```

Приметимо да се у овој функцији дрво може мењати, па аргументи функције не смеју бити константни (иако се суштински вредности у дрвету не мењају, његова интерна репрезентација, тј. расподела података између низова `drvo` и `lenjo` се може мењати).

Пример 1.4.23

Прикажимо рад претходне функције на текућем примеру. Размотримо како се за дрво приказано на слици 1.24 израчунава збир елемената из сегмента $[3, 5]$.

- *Крећемо од корена дрвета које садржи суму сегмента* $[0, 7]$. *Сегмент* $[0, 7]$ *се сече са* $[3, 5]$, *па се рекурзивно обрађују деца.*
  - *У левом поддрвету сегмент* $[0, 3]$ *такође има пресек са* $[3, 5]$ *па прелазимо на наредни ниво рекурзије.*
    - *Приликом посете чвора у чијем је корену вредност* 7 *примећује се да његова вредност није ажурна, па се користи прилика да се она ажурира, тако што се увећа за* $2 \cdot 2 = 4$ *и постаје* 11, *а наследницима се лењи коефицијент поставља на* 2.

      Пошто је сегмент $[0, 1]$ *дисјунктан са* $[3, 5]$, *враћа се вредност* 0.

– *Приликом посете чвора у чијем је корену вредност* 9 *примећује се да његова вредност није ажурна, па се користи прилика да се она ажурира, тако што се увећа за* $2 \cdot 2 = 4$ *и постаје* 13*, а наследницима се лењи коефицијент увећава за* 2*, односно постаје* 5*.*

Сегмент $[2, 3]$ *се сече са* $[3, 5]$*, па се рекурзивно врши обрада поддрвета.*

- *Вредност* 1 *се прво ажурира тако што се повећа за* $1 \cdot 5 = 5$ *и постаје* 6*, а онда, пошто је* $[2, 2]$ *дисјунктно са* $[3, 5]$ *враћа се вредност* 0*.*

- *Вредност* 2 *се такође прво ажурира тако што се повећа за* $1 \cdot 5 = 5$ *и постаје* 7*, а пошто је сегмент* $[3, 3]$ *потпуно садржан у* $[3, 5]$ *враћа се вредност* 7*.*

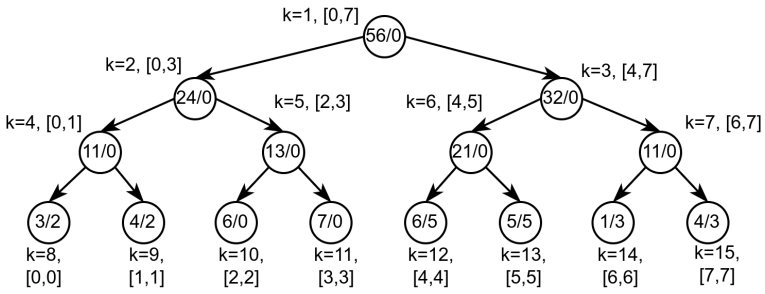
– *У десном поддрвету је чвор са вредношћу* 32 *ажуран, сегмент* $[4, 7]$ *се сече са* $[3, 5]$*, па се прелази на обраду наследника.*

– *Чвор са вредношћу* 21 *је ажуран, сегмент* $[4, 5]$ *је садржан у* $[3, 5]$*, па се враћа вредност* 21*.*

– *Чвор са вредношћу* 11 *је такође ажуран, али је сегмент* $[6, 7]$ *дисјунктан у односу на* $[3, 5]$*, па се враћа вредност* 0*.*

• *Дакле, чворови* 13 *и* 24 *враћају вредност* 7*, чвор* 32 *враћа вредност* 21*, па чвор* 56 *враћа вредност* $7 + 21 = 28$*.*

Дакле, збир сегмента $[3, 5]$ *у текућем дрвету је* 28*. Стање дрвета након извршавања упита је приказано на слици 1.25.*



Слика 1.25: Лењо сегментно дрво након рачунања збира елемената из сегмента $[3, 5]$.

Дакле, у лењом сегментном дрвету не можемо анализом појединачног чвора (нпр. листа) закључити која је тачна вредност тог чвора, међутим, када нам буде била потребна

вредност неког чвора у дрвету, ми ћемо се од корена спустити до тог чвора и, идући том путањом, све вредности на тој путањи ажурирати. На тај начин, када будемо стигли до жељеног чвора, имаћемо његову ажурну вредност, што је једино и важно.

	point query	*range query*	*point update*	*range update*
Лењо сегментно дрво	добро / $O(\log n)$	добро / $O(\log n)$	добро / $O(\log n)$	добро / $O(\log n)$

2. Графовски алгоритми

Графови су једна од најкориснијих структура података. Од давнина су коришћени за представљање мрежа путева између градова и одговарајућих мапа, које су путници носили са собом током путовања. Познат је случај копије мапе из петог века која је садржала мрежу путева Римског царства са називима градова и дужинама путева између њих, која је пружала довољно информација потребних да се пронађе најкраћи пут између два града. Још једна од класичних примена графова (специјално дрвета) је представљање генеалогија. Наиме, породична стабла су вековима коришћена у сврхе одговора на правна питања попут питања дозвољених бракова, наследства и наслеђивања власти. Наравно, постоји много других познатих примера примена графова, као што су представљање страна и ивица полиедара, комуникационе мреже, електрична кола, структурне формуле молекула, друштвене игре, тражење излаза из лавиринта, а у данашње време веома популарна примена је за моделовање односа корисника на друштвеним мрежама.

У овом поглављу ћемо проучити велики број графовских алгоритама.

- У почетном поглављу упознаћемо се са **појмом графа** и начинима на које се он може **представити у рачунару**.
- Затим ћемо се упознати и са основним графовским алгоритмима. Најпре ћемо размотрити два основна алгоритма за обилазак графа: **обилазак у дубину** и **обилазак у ширину**, као и карактеристике графова у односу на њих. Након тога бавићемо се питањем **повезаности графа**: размотрићемо алгоритме за одређивање компоненти повезаности у неусмереном, односно компоненти јаких повезаности у усмереном графу (**Тарџановим** и **Косарајуовим** алгоритмом), као и алгоритмима за одређивање **мостова** и **артикулационих тачака** (то су гране односно

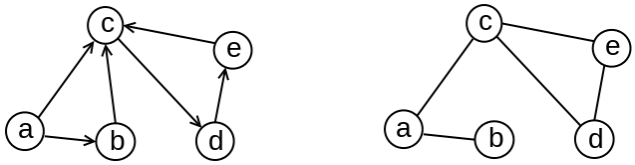
чворови чијим избацивањем повезан граф постаје неповезан).

- Биће речи и о алгоритмима за утврђивање да ли у датом графу постоји **Ојлеров пут**, односно **Хамилтонов пут** (или циклус), тј. пут (циклус) који кроз сваку грану графа, односно кроз сваки чвор графа пролази тачно једном. Детаљно ћемо описати **Хирхолцеров алгоритам** за одређивање Ојлеровог пута (или циклуса).
- Размотрићемо својства **ацикличких усмерених графова**, као и алгоритме **тополошког сортирања графа — Канов алгоритам** и алгоритам **заснован на претрази у дубину**.
- Бавићемо се, такође, различитим проблемима над **тежинским графовима**, пре свега одређивањем **најкраћих путева**. Видећемо како се познавање тополошког сортирања може искористити за ефикасно одређивање најкраћих путева од једног до свих осталих чворова у ацикличком графу. Описаћемо затим **Дајкстрин** алгоритам за одређивање најкраћих путева од једног фиксираног чвора у графу у ком су тежине свих грана ненегативне, а затим и **Белман-Фордов** алгоритам који се може применити за одређивање таквих путева у произвољном графу. На крају ћемо описати и **Флојд-Варшалов** алгоритам за рачунање најкраћих путева између свака два чвора у графу. Приказаћемо и како се проблем проналажења **транзитивног затворења** своди на проблем свих најкраћих путева.
- Коначно, приказаћемо алгоритме за конструкцију **минималног повезујућег дрвета** датог графа: **Примов** алгоритам и **Краскелов** алгоритам.

2.1 Основни појмови

Формално, *граф* $G = (V, E)$ се састоји од скупа V *чворова* и скупа E *грана* (ознаке V и E су почетна слова енглеских речи за теме тј. чвор – vertex и грану – edge). Грана најчешће одговара пару различитих чворова, мада су понекад дозвољене и *петље*, односно гране које воде од чвора ка њему самом. Граф може бити *неусмерен* тј. *неоријентисан* или *усмерен* тј. *оријентисан*. Гране усмереног графа су уређени парови чворова и код њих је редослед чворова које грана повезује битан. Ако се граф представља графички, онда се гране усмереног графа цртају као стрелице усмерене од једног чвора – почетка ка другом чвору – крају гране (слика 2.1, лево). Гране неусмереног графа су неуређени парови чворова: оне се цртају као обичне линије, без усмерења (слика 2.1, десно).

Суседом чвора u називаћемо сваки чвор v до ког постоји грана из чвора u.



Слика 2.1: Пример усмереног графа (лево) и неусмереног графа (десно).

Пример 2.1.1

Суседи чвора c у неусмереном графу са слике 2.1 (десно) су чворови a, d и e, док је у усмереном графу са слике 2.1 (лево) једини сусед чвора c чвор d. Приметимо да је у графу приказаном на слици 2.1 (лево) чвор c повезан и са чворовима a, b и e, међутим, они нису суседи чвора c јер су одговарајуће гране усмерене од чворова a, b и e ка чвору c.

Степен $d(v)$ чвора v у неусмереном графу је број грана суседних чвору v (односно број грана које чвор v повезују са неким другим чвором). У усмереном графу разликујемо *улазни степен* чвора v, који је једнак броју грана за које је чвор v крај, и *излазни степен* чвора v, који је једнак броју грана за које је чвор v почетак.

Пример 2.1.2

Степен чвора c у неусмереном графу са слике 2.1 (десно) је 3*, док је улазни степен чвора c усмереног графа са слике 2.1 (лево) једнак* 3*, а излазни степен* 1*.*

Пут од чвора v_1 до чвора v_k је низ чворова графа $(v_1, v_2, \ldots, v_k)$ повезаних гранама (v_1, v_2), (v_2, v_3), ..., (v_{k-1}, v_k). Пут је *прост* ако се сваки чвор у њему појављује само једном. За чвор v се каже да је *достижан* из чвора u ако постоји пут (усмерен, односно неусмерен, зависно од графа) од u до v. По дефиницији сваки чвор v је достижан из себе. *Циклус* је пут чији се први и последњи чвор поклапају. Циклус је *прост* ако се, сем првог и последњег чвора, ни један други чвор у њему не јавља два пута.

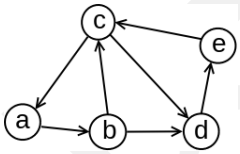
Пример 2.1.3

Низ чворова (a, b, c, d, e) у усмереном графу са слике 2.1 (лево) представља један прост пут у том графу, док пут (b, c, d, e, c) није прост. Чвор e графа са слике 2.1 (лево) достижан је из чвора a јер постоји пут (a, c, d, e) од чвора a до чвора e. Низ чворова (c, d, e, c) представља прост циклус и у датом усмереном и у датом неусмереном графу.

Усмерен граф у коме нема циклуса назива се *усмерен ациклички граф* (енгл. directed acyclic graph, DAG). *Неусмерени облик* усмереног графа $G = (V, E)$ је исти граф, без смерова на гранама (тако да су парови чворова у E неуређени). За неусмерен граф се каже да је *повезан* ако постоји пут између произвољна два чвора у графу. За усмерене графове разликујемо појам слабе и јаке повезаности: усмерени граф је *слабо повезан* ако у његовом неусмереном облику постоји пут између свака два чвора у графу (тј. у полазном графу постоји пут који може да пролази гране и у погрешном смеру), а *јако повезан* ако за свака два чвора u и v у графу постоји усмерен пут од чвора u до чвора v и усмерен пут од чвора v до чвора u.

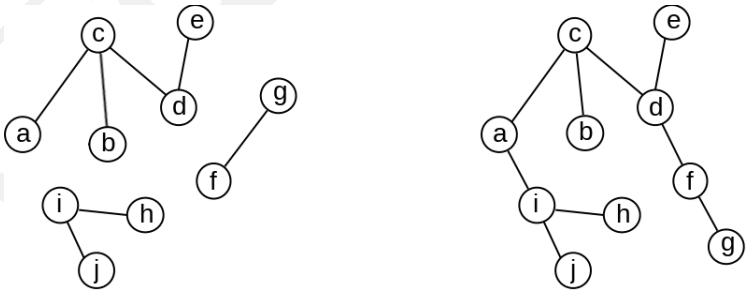
Пример 2.1.4

Неусмерен граф са слике 2.1 (десно) је повезан. Усмерен граф са слике 2.1 (лево) је слабо повезан, али није јако повезан: од чвора b, на пример, није могуће стићи до чвора a. С друге стране, граф са слике 2.2 је јако повезан.



Слика 2.2: Пример усмереног графа који је јако повезан.

Неусмерени граф је *шума* ако не садржи циклусе (слика 2.3, лево). *Дрво* је повезана шума (слика 2.3, десно). Шума садржи једно или више дрвета.



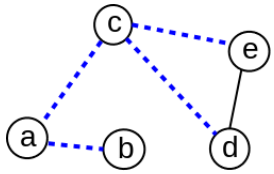
Слика 2.3: Пример графа који је шума (лево) и графа који је дрво (десно)

Граф $G' = (V', E')$, $E' \subseteq V' \times V'$, је *подграф* графа $G = (V, E)$, $E \subseteq V \times V$ ако истовремено важи $V' \subseteq V$ и $E' \subseteq E$. На пример, граф са скупом чворова $\{a, b, c, d\}$ и скупом грана $\{(a, b), (a, c), (c, d)\}$ је један подграф усмереног графа са слике 2.1 (лево). *Повезујуће дрво* неусмереног графа G је његов подграф који је дрво и садржи

све чворове графа G. *Повезујућа шума* неусмереног графа G је његов подграф који је шума и садржи све чворове графа G.

Пример 2.1.5

Гране једног повезујућег дрвета неусмереног графа са слике 2.1 (десно) су приказане плавом бојом на слици 2.4: оно садржи све чворове графа и скуп грана $\{(a,b),(a,c),(c,d),(c,e)\}$.

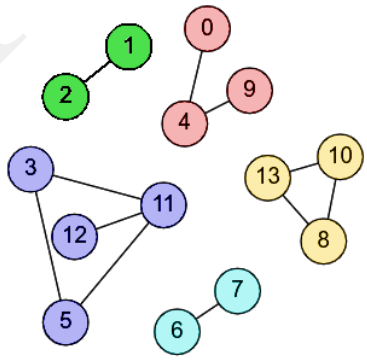


Слика 2.4: Неусмерен граф и једно његово повезујуће дрво (чије су гране приказане испрекидано).

Неусмерени граф $G = (V, E)$ који није повезан се може на јединствен начин разложити у скуп повезаних подграфова, чији скупови чворова представљају класе еквиваленције за релацију достижности и који се називају *компоненте повезаности* графа G.

Пример 2.1.6

На слици 2.5 приказан је граф са 5 *компонената повезаности (чворови сваке компоненте су приказани засебном бојом).*



Слика 2.5: Компоненте повезаности графа.

2.2 Представљање графа

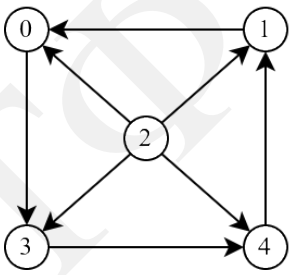
Уобичајена су два начина представљања графова: матрицом повезаности и листама повезаности.

2.2.1 Матрица повезаности

Граф се може представити *матрицом повезаности*, односно *матрицом суседства* графа (енгл. adjacency matrix). Нека је $|V| = n$ и $V = \{v_0, v_1, \ldots, v_{n-1}\}$. Матрица повезаности графа G је квадратна матрица $A = (a_{ij})$ реда n, са елементима a_{ij} који су једнаки 1 (тј. $\top$) ако и само ако $(v_i, v_j) \in E$, односно ако постоји грана од чвора v_i до чвора v_j; остали елементи матрице A имају вредност 0 (тј. $\bot$). Врста i ове матрице је, дакле, низ дужине n чија је j-та координата једнака 1 ако из чвора v_i води грана ка чвору v_j, односно 0 у противном.

Пример 2.2.1

Пример репрезентације једног усмереног графа матрицом повезаности је приказан на слици 2.6. Чворови су нумерисани бројевима од 0 до 4. На пример, на позицији $(1, 0)$ у матрици повезаности налази се вредност 1 јер постоји грана из чвора 1 ка чвору 0, док се на позицији $(0, 1)$ налази 0 јер у графу не постоји супротно усмерена грана.



	0	1	2	3	4
0	0	0	0	1	0
1	1	0	0	0	0
2	1	1	0	1	1
3	0	0	0	0	1
4	0	1	0	0	0

Слика 2.6: Представљање графа матрицом повезаности.

Ако је граф неусмерен, матрица A је симетрична.

Недостатак представљања графа матрицом повезаности је то што она увек заузима простор величине n^2 тј. $\Theta(|V|^2)$, независно од тога колико грана има граф. Ако је број грана у графу мали, већина елемената матрице повезаности је једнака нула. Сложеност операције додавања гране у граф, односно операције уклањања гране из графа је $O(1)$. Такође, и испитивање да ли су два чвора у графу повезана граном је сложености $O(1)$. Пролазак кроз све чворове суседне датом чвору је сложености $\Theta(|V|)$.

У језику C++ матрицу повезаности можемо декларисати на следећи начин (константа `MAX` представља максимални број чворова графа):

```
bool matricaPov[MAX][MAX];
```

Ако број чворова сазнајемо тек у време извршавања програма, можемо употребити динамичке структуре података (на пример, `vector`) и употребити следећу репрезентацију:

```
vector<vector<bool>> matricaPov(n);
for (int i = 0; i < n; i++)
   matricaPov[i].resize(n);
```

2.2.2 Листе повезаности

Уместо да се и све непостојеће гране експлицитно представљају, што је случај са матрицом повезаности графа, могу се формирати повезане листе од редних бројева колона јединица из i-те врсте, $i = 0, 1, \ldots, n-1$. Овај начин представљања графа назива се *листе повезаности*, односно *листе суседства* (енгл. adjacency list). Сваком чвору придружује се повезана листа која садржи све чворове до којих постоји грана из тог чвора. Листа може бити уређена према редним бројевима чворова на крајевима њених грана. Граф је представљен низом листа. Сваки елемент низа садржи име (индекс) чвора и показивач на његову листу суседа.

Пример 2.2.2

Пример представљања графа листама повезаности је приказан на слици 2.7. Листа придружена чвору 1 је дужине 1 јер из чвора 1 полази тачно једна грана. Слично, листа повезаности придружена чвору 2 је дужине 4 јер из чвора 2 полазе четири гране.

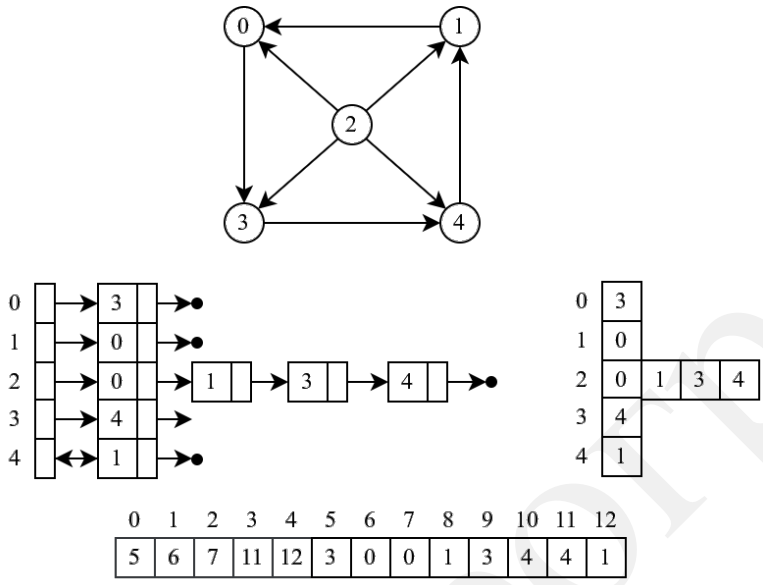
Иако назив тако сугерише, имплементација овакве репрезентације графа не мора бити заснована на повезаним листама, већ се уместо повезаних листи може користити динамички проширив низ (вектор), или нека репрезентација скупа (балансирано бинарно дрво или хеш табела).

У језику C++ се граф представљен листама повезаности, при чему се листе повезаности имплементирају коришћењем вектора, декларише на следећи начин:

```
vector<vector<int>> listaSuseda(n);
```

На пример, усмерени граф са слике 2.7 представићемо на следећи начин:

```
vector<vector<int>> listaSuseda {{3}, {0}, {0, 1, 3, 4}, {4}, {1}};
```



Слика 2.7: Представљање графа листама повезаности (лево су приказане повезане листе, десно низови, а доле статичка имплементација уз помоћ једног низа).

Број чворова графа можемо добити као број елемената у спољашњем вектору:

```
int brCvorova = listaSuseda.size();
```

Нову грану (`cvorOd`,`cvorDo`) можемо додати у усмерен граф на следећи начин:

```
listaSuseda[cvorOd].push_back(cvorDo);
```

Нову грану (`cvorA, cvorB`) можемо додати у неусмерен граф на следећи начин:

```
listaSuseda[cvorA].push_back(cvorB);
listaSuseda[cvorB].push_back(cvorA);
```

Кроз све суседе чвора можемо итерирати на следећи начин:

```
for (int cvorDo : listaSuseda[cvorOd])
    ...
```

Ако је граф *статички*, односно ако нису дозвољена уметања и брисања чворова и грана, онда се листе повезаности могу представити статичким низом a дужине $|V| + |E|$ у случају усмереног графа (односно дужине $|V| + 2|E|$ у случају неусмереног графа). Првих $|V|$ чланова низа су придружени чворовима. Елемент низа a на позицији i, $i <$

$|V|$ је придружен чвору v_i и садржи индекс почетка списка чворова суседних чвору v_i, $i = 0, 1, \ldots, n-1$, док се на позицијама i, $|V| \leq i < |V| + |E|$ налазе информације о суседима чворова. Наиме, суседи чвора v_i за $0 \leq i < n-1$ налазе се у низу a на позицијама $[a_i, a_{i+1})$, док се суседи чвора v_{n-1} налазе на позицијама од $a[n-1]$ до краја низа (слика 2.7, доле). Приметимо да се на позицији 0 у низу a налази вредност која одговара броју чворова у графу.

Пример 2.2.3

Пример представљања графа са слике 2.7, горе, статичким низом је приказан на слици 2.7, доле. На позицији 0 *налази се вредност* 5*, а на позицији* 1 *вредност* 6*, што указује на то да се на позицијама* $[5, 6)$ *у овом низу налазе суседи чвора* 0 *– у овом случају то је само чвор* 3*. Суседи чвора* 2 *налазе се на позицијама* $[7, 11)$ *и то су редом* $0, 1, 3$ *и* 4*.*

Са матрицама повезаности је једноставније радити. С друге стране, листе повезаности су просторно ефикасније за графове са малим бројем грана: њихова меморијска сложеност је $O(|V| + |E|)$, за разлику од матрица повезаности чија је меморијска сложеност $O(|V|^2)$. У пракси се често ради са *ретким* графовима који имају знатно мање грана од максималног могућег броја, што је $n(n-1)/2$ за неусмерени, односно $n(n-1)$ за усмерени граф ако искључимо петље (односно $n(n-1)/2 + n = n(n+1)/2$ грана за неусмерени, односно $n(n-1) + n = n^2$ за усмерени граф ако дозволимо петље) и тада је ефикасније користити листе повезаности. У случају када се за имплементацију користе повезане листе, испитивање да ли су два чвора у графу повезана, као и уклањање гране из графа је у најгорем случају сложености $O(|V|)$. У случају када се за имплементацију листа повезаности користе хеш табеле, очекивано време извршавања ових операција је $O(1)$. Пролазак кроз све чворове суседне чвору v је сложености $O(d(v))$, где је $d(v)$ степен чвора v у случају неусмереног графа, односно излазни степен чвора v ако је граф усмерен. Додавање новог чвора у граф је једноставније него у случају репрезентације графа матрицом повезаности.

Треба поменути да постоје и други начини за представљање графа: граф се, на пример, може чувати и као низ грана, где се за сваку грану чува информација са којим чворовима је инцидентна.

У наредним алгоритмима сматраћемо да је граф са којим радимо динамички и да је, ако није другачије наведено, задат листама повезаности.

2.3 Обилазак графа

Први проблем на који се наилази при конструкцији произвољног алгоритма за обраду графа је како прегледати граф, тј. његове чворове и гране. За разлику од, на пример,

низова где је тај проблем тривијалан због једнодимензионалности улаза (низови се могу лако прегледати секвенцијалним проласком кроз елементе), прегледање графа, односно његов *обилазак*, није тривијалан проблем.

Проблем

Дефинисати алгоритам који кренувши од задатог чвора графа посећује (на пример, исписује, означава или на неки други начин обрађује) све чворове достижне из тог чвора. Сваки чвор треба да буде посећен тачно једном, а чворови могу да буду посећени у произвољном редоследу.

Размотримо случај лавиринта у облику правоугаоне мреже поља, таквих да се са сваког поља може прећи на неко од четири суседна поља: десно, лево, доле и горе. Међутим, нека од поља лавиринта садрже зидове и на њих се не може прећи. Лавиринт садржи два посебна поља која називамо улаз и излаз. Потребно је пронаћи пут којим се од улазног поља може стићи до излазног поља у лавиринту.

Овај проблем можемо представити као графовски проблем: сваком пољу лавиринта које не садржи зид придружујемо један чвор графа, док грана између два чвора постоји ако су одговарајућа поља правоугаоне мреже суседна. На овај начин се проблем проналаска пута кроз лавиринт своди на тражење пута кроз граф од улазног чвора до излазног чвора.

Постоје два основна алгоритма за обилазак графа: *обилазак у дубину* и *обилазак у ширину*. Уместо термина обилазак често се употребљава и термин *претрага*.

У општем случају, алгоритам обиласка тј. претраге повезаног графа из чвора r може имати следећу структуру:

Алгоритам 1 Обилазак графа

```
1: procedure ObilazakGrafa(početni čvor r)
2:     dodaj čvor r u kolekciju K
3:     while K nije prazna do
4:         uzmi čvor u iz K
5:         označi čvor u
6:         po potrebi izvrši obradu čvora u
7:         for all grana (u, v) do
8:             if čvor v nije označen then
9:                 ubaci čvor v u K
```

Механизам означавања чворова нам даје гаранције да ће се алгоритам зауставити – сваки чвор биће означен (тј. посећен и обрађен) највише једном. Претпоставља се да

су у почетку сви чворови неозначени. У зависности од тога која се колекција K одабере, добијају се различити алгоритми обиласка који се могу применити за решавање различитих проблема (за неке проблеме је битно који се обилазак примењује, док за неке није).

- Уколико за колекцију K одаберемо стек, добијамо неку врсту алгоритма *обиласка у дубину*[1].
- Ако је колекција K ред, добијамо алгоритам *обиласка у ширину*.
- Ако је колекција K ред са приоритетом добијамо *обилазак према приоритету*. Ако приоритет чвора представља тежина гране која га спаја са чворовима које смо претходно обишли добијамо минимално повезујуће дрво датог графа, а ако приоритет чвора представља најкраће до тада одређено његово растојање од полазног чвора, добијамо најкраће путеве од полазног чвора до свих осталих чворова у датом графу. О тежинским графовима и овим алгоритмима биће више речи у поглављу 2.8.

2.3.1 Обилазак у дубину

Размотримо претходно разматрани проблем тражења пута кроз лавиринт. Овај проблем можемо решити тако што изаберемо неку путању у лавиринту (по неком унапред дефинисаном правилу) и пратимо је све док не стигнемо до излаза или док не дођемо до поља из кога не можемо даље – у том случају се враћамо уназад и на првој претходној раскрсници идемо неким алтернативним путем. Ово је идеја тзв. *обиласка у дубину*, односно *претраге у дубину* (енгл. depth–first–search, DFS). Обилазак у дубину је практично исти за неусмерене и усмерене графове. Међутим, пошто желимо да испитамо неке особине графова које нису исте за неусмерене и усмерене графове, разматрање обиласка у дубину је подељено на два дела: на обилазак неусмерених и обилазак усмерених графова.

2.3.1.1 Неусмерени графови

Нека је дат граф $G = (V, E)$. Желимо да извршимо обилазак графа тако да увек када је то могуће идемо даље (у дубину) пре него што се вратимо у чвор у ком смо већ били. Овај приступ зове се *обилазак у дубину* (DFS). Основни разлог корисности обиласка у дубину лежи у његовој једноставности и лаком рекурзивном опису. Имплементација може такође бити итеративна, а уместо системског стека који се користи при реализацији рекурзије, може се елиминисати рекурзија тако што се користи посебан стек – таква имплементација се уклапа у општи опис алгоритама обиласка тј. обилазак графова дат на почетку овог поглавља.

[1] Како ће у наставку бити показано, мали детаљи у имплементацији могу утицати на промену редоследа обиласка чворова и/или меморијских захтева алгоритма, тако да овако имплементиран алгоритам не мора да се поклопи са основном алгоритмом претраге у дубину која се дефинише рекурзивно.

Рекурзивна имплементација

Претрага у дубину се једноставно дефинише рекурзивно. Размотримо проблем обиласка графа у дубину када је граф задат листама повезаности и када је дат чвор r графа из кога се започиње обилазак. Иницијално су сви чворови *неозначени*. Чворове ћемо *означавати* у тренутку када их по први пут посетимо. Обично се приликом означавања чвора врши нека његова обрада (мада ћемо видети да обрада може да се врши и на другим местима). Алгоритам функционише тако што означи почетни чвор, а затим се рекурзивно примени на све његове неозначене суседе.

Алгоритам 2 Обилазак у дубину – рекурзивна формулација

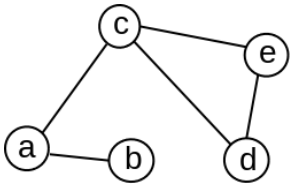
```
1: procedure DFS(čvor u)
2:     označi čvor u
3:     izvrši ulaznu obradu čvora u
4:     for all sused v čvora u do
5:         if čvor v nije označen then
6:             DFS(v)
7:     izvrši izlaznu obradu čvora u
8:
9: DFS(početni čvor r)
```

Дакле, сваки чвор u за кога се позива функција DFS се означава као посећен. Затим се међу суседима чвора u проналази први неозначени сусед v_1, па се из чвора v_1 рекурзивно покреће обилазак у дубину. Затим се прелази на следећег неозначеног суседа v_2 и из њега се рекурзивно покреће обилазак у дубину. Поступак (тренутни рекурзивни позив тј. обрада чвора u) се завршава када су означени сви суседи чвора u.

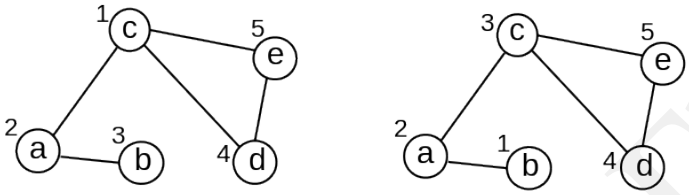
Пример 2.3.1

Размотримо неусмерени граф приказан на слици 2.8: нека је он представљен листама суседства тако да су чворови у свакој листи наведени лексикографски растуће. Ако се на том графу покрене обилазак у дубину из чвора c редом се обилазе чворови c, a, b, d, e (слика 2.9 лево), тако што се редом пролази гранама (c, a), (a, b), (c, d) и (d, e). Ако се обилазак у дубину покрене из чвора b, обилазе се редом чворови b, a, c, d, e (слика 2.9 десно), тако што се редом пролази гранама (b, a), (a, c), (c, d) и (d, e).

Обилазак графа се увек врши са неким циљем. Како би се различите примене уклопиле у обилазак у дубину, посети чвора или гране придружују се две врсте обраде, *улазна обрада* и *излазна обрада*. Улазна обрада врши се у тренутку означавања чвора. Излазна обрада врши се на крају, када су обрађени сви суседи датог чвора. Улазна и излазна



Слика 2.8: Пример неусмереног графа.



Слика 2.9: Илустрација редоследа обиласка чворова приликом обиласка графа у дубину уколико се обилазак покреће из чворова c и b, редом. Уз чворове су приказани бројеви који одговарају редоследу којим се чворови по први пут посећују.

обрада зависе од конкретне примене алгоритма DFS. На тај начин могуће је решавање различитих проблема једноставним дефинисањем улазне и излазне обраде.

Имплементација алгоритма обиласка графа у дубину дата је у наставку. У главној функцији обилазак се покреће из чвора 0. Једноставности ради, граф ће у наредним кодовима бити декларисан као глобална променљива (представљен је граф са слике 2.8 при чему су чворови од a до e нумерисани редом бројевима од 0 до 4).

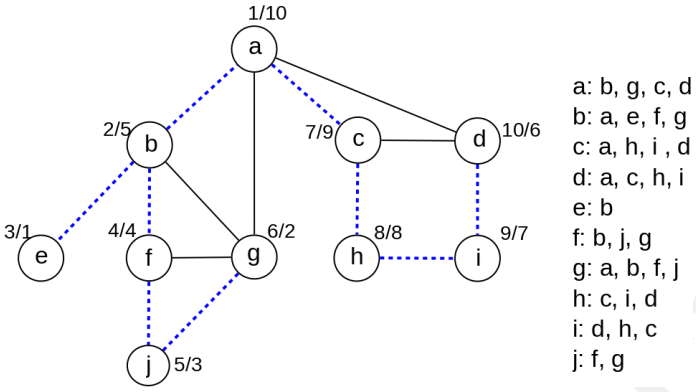
```
// reprezentacija grafa listama povezanosti
// graf je neusmeren, pa se svaka grana dva puta javlja u listi
vector<vector<Cvor>> listaSuseda;

// pomocna rekurzivna funkcija koja vrši DFS obilazak grafa iz datog cvora
void dfs(int cvor, vector<bool> &posecen) {
  posecen[cvor] = true;
  // ovde ide ulazna obrada

  // rekurzivno prolazimo kroz sve njegove susede
  // koje ranije nismo obisli
  for (int sused : listaSuseda[cvor])
    if (!posecen[sused])
```

```
      dfs(sused,posecen);

  // ovde ide izlazna obrada
}

// funkcija koja vrsi DFS obilazak datog grafa iz datog cvora
void dfs(int cvor){
  int brojCvorova = listaSuseda.size();
  vector<bool> posecen(brojCvorova, false);
  dfs(cvor, posecen);
}

int main() {
  // ucitavamo neusmeren graf
  int brojCvorova;
  cin >> brojCvorova;
  listaSuseda.resize(brojCvorova);
  int brojGrana;
  cin >> brojGrana;
  for (int i = 0; i < brojGrana; i++) {
     int cvorOd, cvorDo;
     cin >> cvorOd >> cvorDo;
     listaSuseda[cvorOd].push_back(cvorDo);
     listaSuseda[cvorDo].push_back(cvorOd);
  }

  // vrsimo obilazak od ucitanog pocetnog cvora
  int pocetniCvor;
  cin >> pocetniCvor;
  dfs(pocetniCvor);

  return 0;
}
```

Пример 2.3.2

Пример обиласка графа у дубину приказан је на слици 2.10. Уз сваки чвор су приказани његови редни бројеви у долазној, односно одлазној DFS нумерацији (долазна нумерација означава редослед којим се покретала улазна обрада чворова, а одлазна редослед у ком се завршавала обрада чворова тј. покретала излазна обрада чворова, о чему ће више

речи бити у наставку).



Слика 2.10: Пример обиласка графа у дубину. Граф је задат листама повезаности приказаним десно, а уз сваки чвор је приказан његов редни број у долазној и одлазној DFS нумерацији.

Теорема 2.3.1

Ако је граф G повезан, онда су по завршетку обиласка у дубину сви чворови графа G означени, а све гране графа G су прегледане бар по једном (тј. функција је позвана за сваки чвор графа, а у петљи `for` су анализирани сви његови суседи).

Доказ. Означимо са U скуп неозначених чворова након завршетка извршавања алгоритма. Претпоставимо супротно тј. претпоставимо да је U непразан. Пошто је граф G повезан, а скуп означених чворова $V \setminus U$ је такође непразан (јер садржи бар полазни чвор r који се означава на почетку алгоритма), бар један чвор u из $V \setminus U$ мора бити повезан граном са бар једним неозначеним чвором v из U. Међутим, то је немогуће, јер кад се посети чвор u, морају бити посећени (па дакле и означени) сви његови неозначени суседи, па и чвор v. Дакле, сви чворови графа морају бити посећени и означени. Пошто су сви чворови графа посећени, а кад се чвор посети, онда се прегледају све гране које воде из њега, закључујемо да су и све гране графа прегледане. □

Приликом извршавања алгоритма DFS на неусмереном графу $G = (V, E)$ који је задат листама повезаности, свака грана се прегледа тачно два пута, по једном са сваког краја. Према томе, укупан број извршавања тела петље `for` у свим рекурзивним позивима алгоритма DFS је $O(|E|)$. С друге стране, број рекурзивних позива је $|V|$, па се временска сложеност алгоритма DFS може описати изразом $O(|V| + |E|)$. Приметимо да сложеност обиласка графа у дубину зависи од репрезентације графа: наиме, ако би

граф био задат матрицом повезаности, онда би пролазак кроз суседе једног фиксираног чвора подразумевао пролаз кроз одговарајућу врсту матрице, што је сложености $\Theta(|V|)$. Одатле закључујемо да је пролазак кроз суседе сваког од чворова сложености $\Theta(|V|^2)$. Дакле, у случају алгоритма обиласка у дубину, обично се ефикаснија имплементација добија коришћењем репрезентације графа листама повезаности (нарочито када је граф редак, тј. када сваки чвор има релативно мали број суседа).

Просторна сложеност алгоритма DFS зависи од максималне дубине рекурзије. У најгорем случају граф може имати облик дугог пута и у том случају је дубина рекурзије пропорционална броју чворова у графу. Дакле, просторна сложеност алгоритма DFS је у најгорем случају $O(|V|)$.

Компоненте повезаности

Алгоритам DFS се мора прилагодити да би коректно радио и у случају неповезаних графова. Наиме, ако су после првог покретања описаног алгоритма сви чворови означени, онда је граф повезан, и обилазак је завршен. У противном, може се покренути нови обилазак у дубину полазећи од произвољног неозначеног чвора у графу, итд. Према томе, алгоритам обиласка у дубину се може искористити како би се установило да ли је граф повезан, односно за проналажење свих његових компоненти повезаности.

Проблем

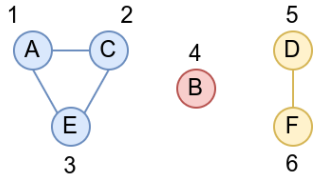
Дефинисати алгоритам који одређује све компоненте повезаности неусмереног графа тј. који свим компонентама повезаности додељује јединствени идентификатор тако да су сви чворови унутар компоненте обележени идентификатором те компоненте.

Алгоритам се лако имплементира ако се покуша покретање обиласка у дубину редом из сваког чвора графа: за сваки чвор се проверава да ли је посећен током обиласка ранијих чворова (чиме је већ придружен некој од ранијих компонената) и ако јесте, прескаче се покретање обиласка из тог чвора.

Пример 2.3.3

Размотримо граф са слике 2.11.

- *Након покретања обиласка у дубину из чвора A обилазе се чворови C и E који припадају његовој компоненти повезаности. Редни број компоненте та три чвора се може поставити на 1, док редни бројеви компоненти осталих чворова и даље остају недефинисани.*
- *Наредни чвор је B и пошто он још није посећен (тј. редни број његове компоненте и даље није дефинисан), покреће се обилазак графа из њега. Он је једини достижан*



Слика 2.11: Одређивање компоненти повезаности неусмереног графа. Приказан је и редослед обиласка чворова.

и једино њему се додељује редни број компоненте 2.

- *Наредни чвор је C. Пошто је он већ посећен (припада компоненти 1), не покреће се обилазак из њега.*
- *Наредни чвор је D и пошто он још није посећен, покреће се обилазак графа из њега. Том приликом се посећују чворови D и F и редни бројеви њихове компоненте се постављају на 3.*
- *Наредни чвор је E. Пошто је он већ посећен (припада компоненти 1), не покреће се обилазак из њега.*
- *Наредни чвор је F. Пошто је он већ посећен (припада компоненти 3), не покреће се обилазак из њега.*

У овом и у наредним кодовима, једноставности ради, претпостављамо да је граф представљен глобалном променљивом `listaSuseda`.

```
// obilazak u dubinu iz cvora cvor
void dfs(int cvor, int brojKomponente, vector<int>& komponente) {
  // tekuci cvor pridruzujemo tekucoj komponenti
  komponente[cvor] = brojKomponente;

  // rekurzivno prolazimo kroz sve njegove susede
  // koje ranije nismo obisli
  for (int sused : listaSuseda[cvor])
    if (komponente[sused] == -1)
      dfs(sused, brojKomponente, komponente);
}

// za svaki cvor grafa se u vektor komponente upisuje
// redni broj komponente kojoj on pripada;
```

```
// funkcija vraca ukupan broj komponenata povezanosti
int komponentePovezanosti(vector<int>& komponente) {
  int brojCvorova = listaSuseda.size();
  komponente.resize(brojCvorova, -1);
  int brojKomponente = 0;
  // pokrecemo novi DFS obilazak iz prvog neposecenog cvora
  for (int cvor = 0; cvor < brojCvorova; cvor++)
    if (komponente[cvor] == -1) {
      dfs(cvor, brojKomponente, komponente);
      brojKomponente++;
    }
  return brojKomponente;
}
```

Иако се претрага у дубину може позвати и више пута, и овај алгоритам је временске сложености $O(|V| + |E|)$. Наиме, обилазак компоненте број k биће сложености $O(|V_k| + |E_k|)$, где је V_k број чворова, а E_k број грана унутар те компоненте. Зато се укупно, током свих обилазака заједно обиђе $O(|V|)$ чворова и $O(|E|)$ грана.

Ми ћемо најчешће разматрати случај када је граф повезан, јер се у општем случају проблем своди на посебну обраду сваке компоненте повезаности.

Итеративне имплементације помоћу стека

Иако је једноставна за разумевање, мана рекурзивне имплементације је то што код неких графова може доћи до прекорачења системске стек-меморије приликом извршавања програма (то се може десити ако постоји дугачак низ чворова којим се пролази без враћања уназад). Наиме, рекурзивни позиви користе системски стек рачунара, на који се смештају параметри актуелних рекурзивних позива (аргументи прослеђени приликом сваког од рекурзивних позива и вредности локалних променљивих). Уместо системског стека који и на савременим системима представља веома ограничен део меморије (обично је у питању тек неколико мегабајта) можемо у нашем програму одржавати посебан стек, који може да заузме много више меморијског простора, јер се алоцира у зонама којима је придружено много више меморије (на пример, у зони хипа). Стога је понекад потребно направити имплементацију обиласка у дубину која није рекурзивна, а која, као што је то и иначе често случај код ослобађања од рекурзије, захтева коришћење структуре података стек.

Верна варијанта: ослобађање од рекурзије

Можемо приметити да се током рекурзивних позива на системском стеку у сваком тренутку налазе сви чворови на тренутном путу од почетног до текућег чвора. Те информације су нам потребне због враћања унатраг. Наиме, када из текућег чвора не можемо

прећи даље у неки непосећен чвор, чвор који се на стеку налази испод текућег је онај из којег смо дошли у текући чвор и у који треба да се вратимо (то је његов родитељски чвор). Приликом извршавања рекурзивне функције, у сваком чвору се извршава петља у којој се пролази кроз непосећене суседе тог чвора. Приликом повратка у претходни чвор потребно је извршити наредну итерацију те петље. Стога се у системском стек оквиру који одговара чвору чува и тренутна вредност бројачке промењљиве у склопу те петље (како би се након повратка у чвор та промењљива могла увећати и тако прећи на наредни корак итерације, тј. на наредног суседа). Наравно, све се ово дешава аутоматски (на системском стеку се чувају вредности свих локалних промењљивих), па програмер не мора да води рачуна о томе. У склопу нерекурзивне имплементације можемо имитирати овај поступак, тако што на стеку који ручно одржавамо чувамо уређен пар који садржи ознаку родитељског чвора и бројач у петљи у том родитељском чвору тј. ознаку детета тог родитеља које је тренутно обрађено.

```
// obilazak grafa u dubinu iz datog pocetnog cvora
void dfs(int pocetniCvor) {
   int brojCvorova = listaSuseda.size();
   vector<bool> posecen(brojCvorova, false);
   // na steku pamtimo roditeljski cvor u koji treba da se vratimo
   // i broj do sada obrađene dece u tom roditeljskom čvoru
   stack<pair<int, int>> stek;
   // prvi roditelj je pocetni cvor i za sada nije obrađeno
   // ni jedno dete
   stek.emplace(pocetniCvor, 0);
   // ovo je prvi put da smo otkrili pocetni cvor, pa vrsimo njegovu
   // ulaznu obradu
   posecen[pocetniCvor] = true;
   cout << pocetniCvor << endl; // ulazna obrada
   // dok se stek ne isprazni
   while (!stek.empty()) {
       // skidamo roditeljski cvor sa steka
       auto [roditelj, i] = stek.top();
       stek.pop();
       // ako nisu još obrađena sva njegova deca
       if (i < listaSuseda[roditelj].size()) {
          // obradjujemo naredno dete

          // pamtimo na steku da je broj obradjene dece ovog roditelja
          // uvećan za 1, tj. da kada se sledeci put u povratku vratimo
```

```
            // do ovog roditelja, da tada preskocimo trenutno dete
            stek.emplace(roditelj, i+1);
            // ako trenutno dete nije vec poseceno, sada ga posecujemo i
            // vrsimo njegovu ulaznu obradu
            int dete = listaSuseda[roditelj][i];
            if (!posecen[dete]) {
               posecen[dete] = true;
               cout << dete << endl; // ulazna obrada
               stek.emplace(dete, 0);
            }
         }
      }
   }
```

Имплементација добијена на овај начин је једина која верно осликава рекурзивни обилазак у дубину. Могуће су и другачије имплементације у којима је кôд мало једноставнији, међутим, видећемо да се свака од њих по нечему разликује од основне рекурзивне формулације.

Псеудо-DFS

Размотримо сада једну могућу модификацију обиласка графа имплементираног уз помоћ стека. Да не бисмо морали да прекидамо петљу и настављамо њено извршавање после прекида, можемо програм организовати тако да на стек одмах ставимо све суседе текућег чвора и да их одмах означимо. Ознака сада значи да је чвор стављен на стек, тј. да је заказано да ће чвор у неком наредном тренутку бити посећен, а не да је већ посећен и обрађен. Дакле, на стеку чувамо чворове које у будућности треба обрадити (када их ставимо на стек, сматрамо да смо заказали њихову обраду). Чвор обично обрађујемо у тренутку када га скидамо са стека. Након тога на стек стављамо његове суседе за које још није заказан обилазак. Описани поступак изложен је у алгоритму 3.

То што се чвор означава чим се први пут стави на стек, а не тек када се скида са стека нам гарантује да ће сваки чвор највише једном бити уписан на стек. Ипак ово може утицати на промену редоследа обиласка чворова у односу на рекурзивну имплементацију, тако да овај алгоритам није у стриктном смислу обилазак графа у дубину (зато се понекад назива *псеудо-DFS*). Ипак, у већини примена довољно је само да се гарантује да ће се сваки чвор обићи тачно једном, па овај алгоритам често задовољава све наше потребе.

Чак иако се у овој варијанти сваки чвор ставља само једном на стек, дубина стека (па самим тим и потрошња меморије) потребна за извршавање алгоритма може бити знатно већа него што је случај са претходном, верном варијантом обиласка у дубину. Наиме, у верној варијанти обиласка у дубину је број елемената на стеку асимптотски једнак

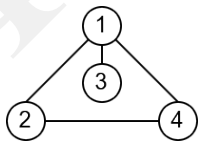
Алгоритам 3 Обилазак у дубину – формулација са стеком

```
1: procedure DFS(čvor r)
2:     stavi početni čvor r na stek
3:     while stek nije prazan do
4:         skini čvor u sa vrha steka
5:         izvrši ulaznu obradu čvora u
6:         for all sused v čvora u do
7:             if čvor v nije označen then          ▷ tj. nije mu zakazan obilazak
8:                 stavi čvor v na stek
9:                 označi čvor v                    ▷ tj. zakazuje mu se obilazak
```

дужини најдужег пута од почетног до неког удаљеног чвора из ког не можемо даље да се крећемо, док се у псеудо-DFS обиласку на стеку истовремено налазе и сви суседи тих чворова. На пример, ако размотримо граф облика звезде у ком је један централни чвор повезан са N суседа, у верној варијанти би се на стек ставио централни чвор, а затим би се на стек стављао, па скидао један по један његов сусед, тако да је за обилазак довољан стек дубине 2. У псеудо-DFS алгоритму би, међутим, након централног чвора на стек одмах било додато свих N суседа централног чвора, те је за обилазак потребан стек дубине N (наравно, у најгорем случају оба алгоритма захтевају простор $O(N)$).

Пример 2.3.4

Размотримо граф на слици 2.12. Нека је задат листама повезаности и нека су листе повезаности сортиране у растућем редоследу суседа.



Слика 2.12: Граф код којег се разликује редослед обиласка чворова приликом рекурзивног обиласка у дубину и обиласка алгоритмом псеудо-DFS.

Приликом рекурзивног обиласка у дубину кренувши од чвора 1*, рекурзивно се прелази на чвор* 2*, па затим на чвор* 4*. Пошто чвор* 4 *нема даљих непосећених суседа враћамо се назад, све до чвора* 1 *и након тога се посећује чвор* 3*. Редослед обиласка и обраде чворова је, дакле,* $1, 2, 4, 3$.

Приликом псеудо-DFS обиласка истог графа из чвора 1 *на стек се најпре ставља чвор* 1*. Затим се тај чвор скида са стека, обрађује се и на стек се стављају његови суседи* 2*,* 3 *и* 4*. Затим се скида и обрађује чвор* 2*. Он нема непосећених суседа. Скида се затим и*

обрађује чвор 3*, који такође нема непосећених суседа. На крају се скида и обрађује чвор* 4*, који такође нема непосећених суседа. Редослед обиласка и обраде је, дакле,* $1, 2, 3, 4$*, што је различито од стварног редоследа обиласка у дубину који захтева да се из чвора* 2 *пређе у до тада непосећени чвор* 4*.*

Теорема 2.3.2

Ако је граф повезан, онда су по завршетку алгоритма 3 сви чворови обрађени (за сваки чвор је извршена улазна обрада).

Доказ. Доказ тече слично доказу теореме 2.3.1. Означимо са U скуп чворова који нису обрађени након завршетка извршавања алгоритма. Претпоставимо супротно претпоставци да постоји неки необрађени чвор тј. претпоставимо да је скуп U непразан. Пошто је граф G повезан, а скуп обрађених чворова $V \setminus U$ је такође непразан (јер садржи бар полазни чвор r који се обрађује у првој итерацији петље), бар један обрађени чвор u из $V \setminus U$ мора бити повезан граном са бар једним необрађеним чвором v из U. Међутим, овако нешто је немогуће, јер након што се обради чвор u, означавају се сви његови до тада неозначени суседи, па и чвор v. Чвор v је, дакле, морао бити означен и стављен на стек, а сви чворови који су стављени на стек се у неком тренутку обрађују, па је и чвор v морао бити обрађен, што је контрадикција. Дакле, сви чворови графа морају бити обрађени. □

На основу алгоритма 3 веома је једноставно направити имплементацију у језику C++ (стога је нећемо приказивати).

Вишеструко стављање чворова на стек

Алтернатива претходном алгоритму је да се означавање чворова врши у тренутку када се заиста посете тј. скину са стека, као што је приказано у општем алгоритму за обилазак графова на почетку овог поглавља. Овај приступ описан је у алгоритму 4.

Под претпоставком да се суседи на стек постављају у обратном редоследу од оног у ком су наведени у листи повезаности, тиме се добија исти редослед обиласка као у рекурзивној имплементацији тј. у питању је заиста обилазак у дубину. Мана је то што се исти елементи могу више пута наћи на стеку, тако да је понекад потребна значајно већа дубина стека да се цео граф обиђе. Наиме, иако се на стек стављају само неозначени чворови, они се не означавају приликом стављања већ тек приликом скидања са стека. Веома је важно да се суседи означених чворова не анализирају када се ти чворови скину са стека, јер би се у супротном за исти чвор више пута пролазило кроз листе суседа.

И ову имплементацију је веома једноставно направити на основу датог алгоритма, па је нећемо приказивати.

Алгоритам 4 Обилазак у дубину – формулација са стеком

```
1: procedure DFS(čvor r)
2:     stavi početni čvor r na stek
3:     while stek nije prazan do
4:         skini čvor u sa vrha steka
5:         if čvor u nije označen then
6:             označi čvor u
7:             izvrši ulaznu obradu čvora u
8:             for all sused v čvora u do
9:                 stavi čvor v na stek
```

Приметимо да је излазну обраду чвора теже имплементирати у (свим) нерекурзивним имплементацијама. Наиме, излазну обраду текућег чвора је потребно извршити тек када са стека буде скинут и последњи његов сусед. Један начин да се тај тренутак уочи је да се на стек пре суседа текућег чвора стави специјална ознака чије скидање са стека указује да је потребно извршити излазну обраду текућег чвора.

Приметимо да је просторна сложеност нерекурзивних верзија алгоритма DFS једнака максималном броју чворова који се током алгоритма нађу у стеку, што је опет у најгорем случају $O(|V|)$, као и у рекурзивној имплементацији.

Конструкција DFS дрвета и DFS нумерација

Приказаћемо сада две једноставне а важне примене алгоритма DFS — формирање специјалног повезујућег дрвета, такозваног *DFS дрвета* и нумерацију чворова графа *DFS бројевима*. Текст у наставку се односи на рекурзивни обилазак.

Приликом обиласка графа G у дубину, у петљи којом се пролазе сви суседи чвора u могу се издвојити све гране ка новоозначеним чворовима v. Преко издвојених грана достижни су сви чворови повезаног неусмереног графа, па је подграф кога чине издвојене гране повезан. Тај подграф нема циклусе, јер се од свих грана које воде у неки чвор, издваја само једна. Према томе, издвојене гране су гране подграфа графа G који је у случају повезаног графа повезујуће дрво, које називамо *DFS дрво* графа G. Чвор из кога се покреће обилазак у дубину је корен DFS дрвета. Чак и ако се дрво не формира експлицитно, многе алгоритме је лакше разумети разматрајући DFS дрво графа G.

Проблем

Дефинисати алгоритам који у неусмереном графу конструише DFS дрво тј. који одређује све гране тог дрвета.

У наставку је дат програм који конструише DFS дрво датог графа представљеног ли-

стама суседства. DFS дрво биће представљено у виду вектора грана графа.

```
// rekurzivna funkcija koja vrsi DFS obilazak grafa od datog cvora  i
// uz to konstruise DFS drvo
void konstruisi_dfs_drvo(int cvor, vector<bool> &posecen,
                         vector<vector<int>> &dfs_drvo) {
  posecen[cvor] = true;

  // rekurzivno prolazimo kroz sve njegove susede
  // koje ranije nismo obisli
  for (int sused : listaSuseda[cvor]) {
    if (!posecen[sused]) {
      // u DFS drvo dodajemo granu iz tekuceg ka novom cvoru
      // i njoj suprotno usmerenu granu (jer je graf neusmeren)
      dfs_drvo[cvor].push_back(sused);
      dfs_drvo[sused].push_back(cvor);
      konstruisi_dfs_drvo(sused, posecen, dfs_drvo);
    }
  }
}

// omotac funkcija koja vrsi DFS obilazak datog grafa iz datog cvora
// i konstruise DFS drvo
vector<vector<int>> konstruisi_dfs_drvo(int cvor) {
  int brojCvorova = listaSuseda.size();
  vector<bool> posecen(brojCvorova, false);
  vector<vector<int>> dfs_drvo(brojCvorova);
  konstruisi_dfs_drvo(cvor, posecen, dfs_drvo);
  return dfs_drvo;
}
```

Постоје две варијанте DFS нумерације чворова:

- *долазна DFS нумерација*, односно *preOrder* нумерација, којом се чворови нумеришу према редоследу означавања и
- *одлазна DFS нумерација*, односно *postOrder* нумерација, код које се чворови нумеришу према редоследу напуштања.

Долазни број чвора v обележаваћемо са $v.Pre$, а одлазни са $v.Post$. Долазна и одлазна нумерација чворова имају примену приликом решавања разних проблема над графовима.

Пример 2.3.5

Пример графа са чворовима нумерисаним на два начина приказан је на слици 2.10.

За пример са слике 2.10 редослед чворова у растућем редоследу долазне нумерације гласи $a, b, e, f, j, g, c, h, i, d$*, док је редослед чворова у растућем редоследу одлазне нумерације* $e, g, j, f, b, d, i, h, c, a$*.*

Наредна функција исписује чворове у растућем редоследу њихове долазне DFS нумерације.

```
void dfs_preorder(int cvor, vector<bool> &posecen) {
  posecen[cvor] = true;
  // stampamo naredni cvor u preOrder numeraciji
  cout << cvor << " ";

  // rekurzivno prolazimo kroz sve susede koje nismo obisli
  for (int sused : listaSuseda[cvor])
    if (!posecen[sused])
      dfs_preorder(sused,posecen);
}
```

Наредна функција исписује чворове у растућем редоследу њихове одлазне DFS нумерације.

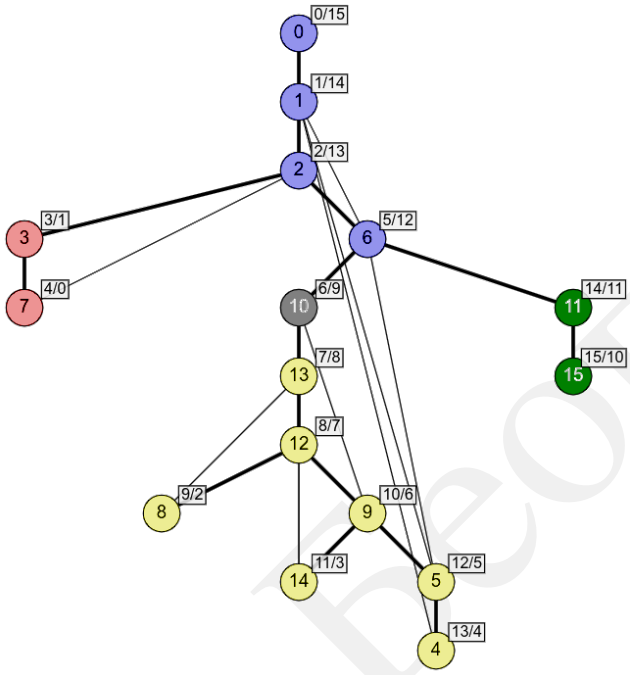
```
void dfs_postorder(int cvor, vector<bool> &posecen) {
  posecen[cvor] = true;

  // rekurzivno prolazimo kroz sve susede koje nismo obisli
  for (int sused : listaSuseda[cvor])
    if (!posecen[sused])
      dfs_postorder(sused,posecen);

  // stampamo naredni cvor u postOrder numeraciji
  cout << cvor << " ";
}
```

Алтернативно, можемо имплементирати функције којима се за све чворове у графу памте, а на крају исписују редни бројеви у долазној и одлазној DFS нумерацији.

```
// za svaki cvor se pamti dolazni i odlazni redni broj
vector<int> dolazna;
vector<int> odlazna;
// naredni slobodan redni broj (odlazni i dolazni)
int rbr_dolazna = 0;
int rbr_odlazna = 0;

void dfs(int cvor, vector<bool> &posecen) {
  posecen[cvor] = true;

  // cvor dobija naredni dolazni redni broj
  dolazna[cvor] = rbr_dolazna;
  rbr_dolazna++;

  // rekurzivno prolazimo kroz sve susede koje do sada nismo obisli
  for (int sused : listaSuseda[cvor]) {
    if (!posecen[sused]) {
      dfs(sused,posecen);
    }
  }
  // cvor dobija naredni odlazni redni broj
  odlazna[cvor] = rbr_odlazna;
  rbr_odlazna++;
}

void dfs(int cvor) {
  int brojCvorova = listaSuseda.size();
  dolazna.resize(brojCvorova, -1);
  odlazna.resize(brojCvorova, -1);
  vector<bool> posecen(brojCvorova, false);

  dfs(cvor,posecen);

  cout << "Dolazna i odlazna numeracija cvorova:" << endl;
  for (int i = 0; i < brojCvorova; i++)
    cout << "Cvor " << i << ": " << dolazna[i]
         << "/" << odlazna[i] << endl;
}
```

У односу на произвољни чвор у дрвету, остале чворове можемо поделити у четири групе:

чворови који су његови преци, чворове који су његови потомци, чворове који су лево од њега и чворове који су десно од њега (видети пример на слици 2.13).



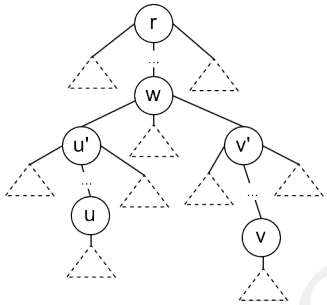
Слика 2.13: У односу на чвор број 10, црвени чворови су лево, зелени десно, плави су преци, а жути су потомци. Чворови лево имају мање и долазне и одлазне редне бројеве, преци имају мање долазне, а веће одлазне редне бројеве, потомци имају веће долазне, а мање одлазне редне бројеве, док чворови десно имају веће и долазне и одлазне редне бројеве.

Дефинишимо прецизно ове односе.

Чвор w називамо *прет̄ком* чвора v у DFS дрвету T са кореном r (слика 2.14) ако је w на јединственом путу од r до v у T. Ако је чвор w предак чвора v, онда је чвор v *пот̄омак* чвора w.

DFS дрво обухвата све чворове повезаног графа G. Редослед деце сваког чвора у дрвету одређен је листама повезаности којима се задаје граф G, па се за свака два детета истог чвора може рећи који је од њих *леви* (први по том редоследу), а који *десни*. Релација леви – десни се преноси на произвољна два чвора u и v који нису у релацији предак – потомак (слика 2.14). За чворове u и v тада постоји „најмлађи" заједнички предак w у DFS дрвету (то је онај заједнички предак чији ниједан потомак није заједнички предак чворова u и v), као и деца u' и v' чвора w такви да је чвор u' предак чвора u и

чвор v' предак чвора v. Кажемо да је *чвор u лево од чвора v* ако и само ако је чвор u' лево од чвора v'. Геометријска интерпретација ове релације је јасна: DFS дрво ћемо исцртавати наниже приликом преласка у нове – неозначене чворове (кораци у дубину), односно слева удесно приликом додавања нових грана после повратка у већ означене чворове.



Слика 2.14: Илустрација релације *лево – десно* на скупу чворова DFS дрвета.

Релације предак–потомак и леви–десни се могу окарактерисати и коришћењем долазне и одлазне нумерације чворова.

Редослед обраде чворова је одозго-наниже, слева-надесно, па се пре било ког чвора обележавају чворови лево од њега и чворови изнад њега (његови преци), а после њега се обележавају чворови испод њега (његови потомци) као и чворови десно од њега. Дакле, ако су дата два чвора u и v таква да је $u.Pre < v.Pre$, тада је u или лево од v или је предак чвора v. Како бисмо знали који од та два случаја важи, потребно је да упоредимо одлазне редне бројеве: мањи одлазни бројеви указују на чворове лево, а већи на претке. Следећа лема нам даје потпуну карактеризацију односа положаја чворова на основу њихове DFS нумерације (видети пример на слици 2.13).

Лема 2.3.1 **[DFS нумерација и однос положаја чворова]**

Ако за два чвора u и v важи $u.Pre < v.Pre$, тада важи:

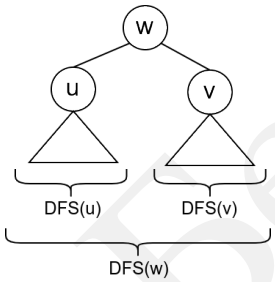
- *ако је $u.Post < v.Post$, тада је u лево од v (обрада чворова лево раније почиње и раније се завршава);*
- *ако је $u.Post > v.Post$, тада је u предак од v (обрада предака раније почиње, али се касније завршава).*

Из овога директно следе и следећи (дуални) услови.

Ако за два чвора u и v важи $u.Pre > v.Pre$, тада важи:

- *ако је u.Post $<$ v.Post, тада је u потомак чвора v (обрада потомака касније почиње, али се раније завршава);*
- *ако је u.Post $>$ v.Post, тада је u десно од v (обрада чворова десно касније почиње и касније се завршава).*

Доказ. На слици 2.15 приказана су три чвора графа u, v и w у оквиру DFS дрвета графа. Чворови u и v су деца чвора w, а чвор u је лево од чвора v. На слици су приказани и временски интервали трајања рекурзивних позива алгоритма DFS за сваки од ових чворова. Прво се покреће DFS обилазак чвора w, након тога се покреће DFS обилазак чвора u, затим се тај обилазак завршава, па се позива DFS обилазак чвора v, након тога се тај обилазак завршава и на крају се завршава DFS обилазак чвора w.



Слика 2.15: Однос између положаја чворова у DFS дрвету и трајања рекурзивних позива покренутих из ових чворова.

Било који потомак чвора w (рецимо w') се налази унутар поддрвета са кореном чији је корен неко дете чвора w (то су u, v, ...), па мора да има већи долазни DFS број од долазног броја чвора w тј. важи $w'.Pre > w.Pre$. Запажамо да је DFS алгоритам покренут из чвора било ког потомка чвора w, активан само у подинтервалу времена за које је активан DFS алгоритам из чвора w. DFS претрага из било ког чвора w' који је потомак чвора w завршава се пре завршетка DFS претраге из чвора w. Према томе, из чињенице да је неки чвор w' потомак чвора w следи да је $w'.Post < w.Post$.

Ако је неки чвор u' лево од неког чвора v', онда u' мора бити у поддрвету чији је корен u, а v' у поддрвету чији је корен v (за неко u, v који имају заједничког родитеља w и u је лево од v). DFS обилазак за сваки чвор у поддрвету са кореном u и почиње и завршава се пре DFS обиласка било ког чвора у поддрвету са кореном v. Зато је $u'.Pre < v'.Pre$ и $u'.Post < v'.Post$. □

Приметимо да претходна лема даје потпуну карактеризацију односа положаја чворова у дрвету (постоји четири могућа односа положаја и четири односа парова редних бројева

у долазној и одлазној нумерацији) – у сва четири случаја важе еквиваленције (однос положаја важи ако и само ако важи наведени однос редних бројева).

На сликама 2.10 и 2.13 се види да све гране неусмереног графа спајају претке и потомке тј. да гране не могу бити *попречне* у односу на DFS дрво, односно не повезују чворове на раздвојеним путевима од корена (тј. два чвора која су у односу лево – десно). Заиста, ако би таква грана постојала, DFS алгоритам би „прошао" њоме и укључио је у DFS дрво. Наредна лема прецизно изражава ову карактеризацију грана графа у односу на DFS дрво.

Лема 2.3.2 [Карактеризација грана неусмереног графа у односу на DFS дрво]

Нека је $G = (V, E)$ повезан неусмерен граф и нека је $T = (V, F)$ DFS дрво графа G. Свака грана графа $e \in E$ припада дрвету T (тј. $e \in F$) или спаја два чвора графа G, од којих је један предак другог у дрвету T.

Доказ. Нека је (u, v) грана неусмереног графа G, и претпоставимо да је у току DFS претраге чвор u посећен пре чвора v тј. да је $u.Pre < v.Pre$. На основу леме 2.3.1 важи да је v или потомак чвора u или је десно од u. После означавања чвора u, у петљи се рекурзивно покреће DFS претрага из сваког неозначеног суседа чвора u. У тренутку кад дође ред на чвор v, ако v није означен, из v се започиње нови рекурзивни позив DFS, па је v дете чвора u у дрвету T и грана (u, v) припада DFS дрвету T. Ако је v означен, онда је v потомак чвора u у дрвету T. Заиста, пошто је v означен, његова обрада мора бити започета и завршена током тренутног рекурзивног позива у ком се обрађује чвор u, па важи $v.Pre > u.Pre$ и $v.Post < u.Post$. На основу леме 2.3.1 важи да чвор u мора бити предак чвора v. □

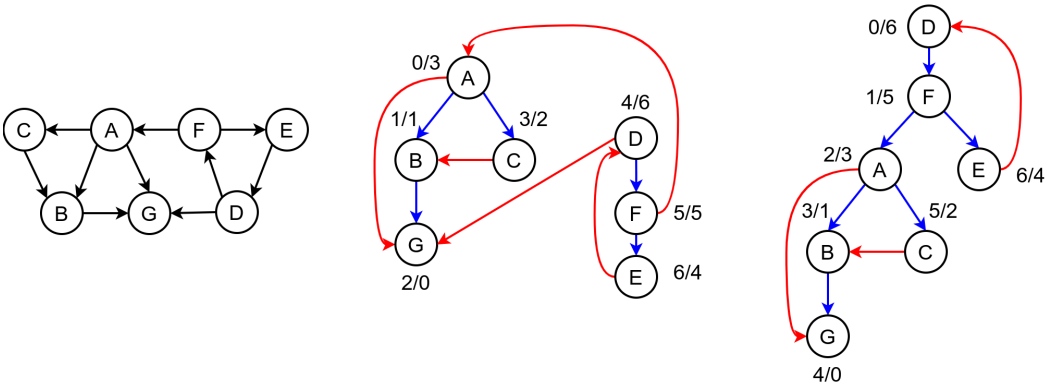
2.3.1.2 Усмерени графови

Процедура претраге у дубину усмерених графова иста је као за неусмерене графове.

Алгоритам DFS за повезан неусмерени граф, покренут из произвољног чвора, обилази цео граф. Аналогно тврђење не мора да важи за усмерене графове. Прецизније, оно ће важити само ако је граф јако повезан (о чему ће бити више речи у поглављу 2.6).

Пример 2.3.6

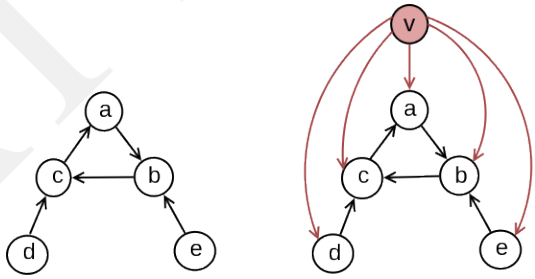
Посматрајмо усмерени граф на слици 2.16. Ако се DFS претрага започне из чвора D, биће посећени сви чворови графа. Са друге стране, ако се DFS претрага започне из чвора A, онда ће бити достигнути само чворови A, B, C и G. Да би се обишли сви чворови, потребно је покренути нов обилазак (на пример из чвора D). На тај начин се уместо DFS дрвета добија DFS шума. Приметимо да могу да постоје и гране које повезују два дрвета те шуме.



Слика 2.16: Покретањем обиласка графа са леве стране из чвора A, нису посећени сви чворови, па је обилазак покренут поново из чвора D. Тиме је добијена шума приказана на средини. Ако би се обилазак покренуо из чвора D, добило би се јединствено дрво, приказано десно.

Према томе, увек кад говоримо о DFS обиласку усмереног графа, сматраћемо да је алгоритам DFS покренут онолико пута колико је потребно да би сви чворови били означени и све гране биле размотрене. Дакле, у општем случају усмерени граф уместо DFS дрвета има *DFS шуму*.

Јединствено DFS дрво се може обезбедити тако што се у граф дода нови чвор v улазног степена 0, који се повеже гранама са свим чворовима графа G (слика 2.17) – тада DFS обилазак покренут из чвора v сигурно обилази све чворове графа G и постоји јединствено DFS дрво.



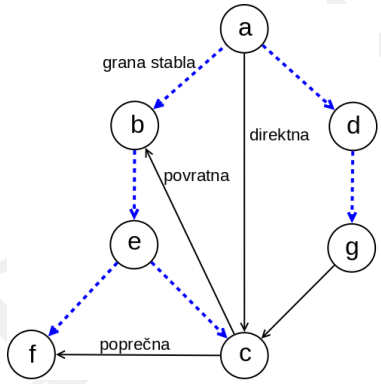
Слика 2.17: Пример додавања новог чвора v у граф тако да се из чвора v граф може у потпуности обићи.

Чак и када су јединствена за цео граф, усмерена DFS дрвета имају нешто другачије особине од DFS дрвета за неусмерене графове.

И у DFS шуми код усмерених графова важе исте врсте односа између чворова као и код неусмерених графова (односи предак-потомак и лево-десно) и лема 2.3.1 која даје карактеризацију ових односа на основу долазне и одлазне DFS нумерације остаје на снази. Наиме, однос трајања рекурзивних позива приказан на слици 2.15 и даље важи. Иако то није показано на слици 2.15, исти закључак је тачан и ако су чворови v и w у различитим дрветима DFS шуме, при чему је дрво чвора v лево од дрвета чвора w.

Међутим, карактеризација грана је другачија. На пример, није више тачно да граф не може имати попречне гране, што се може видети из примера на слици 2.18. У односу на DFS дрво гране усмереног графа припадају једној од четири категорије:

- *гране DFS дрвета*,
- *повратне*,
- *директне* и
- *попречне* гране.



Слика 2.18: DFS дрво усмереног графа. Класификација грана у односу на DFS дрво.

Прве три врсте грана повезују два чвора од којих је један потомак другог у DFS дрвету (слика 2.18):

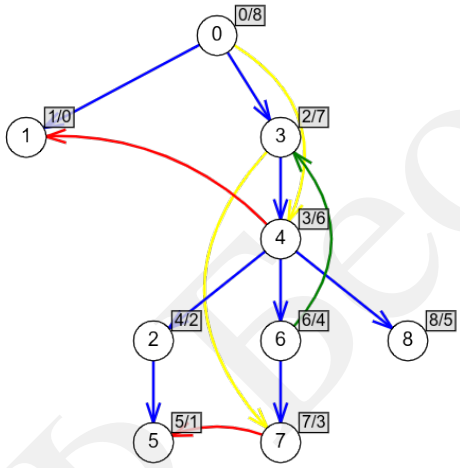
- грана DFS дрвета повезује родитеља са дететом,
- повратна грана повезује потомка са претком,
- директна грана повезује претка са потомком.

Ове врсте грана постоје и код неусмерених графова, једино што се не прави разлика између директних и повратних грана (оне спајају претке и потомке, али нису усмерене, па их не можемо разликовати).

Новина су попречне гране, које повезују чворове који нису „сродници” у дрвету. Попречне гране, међутим, морају бити усмерене „здесна улево” (од чворова са већим ка

чворова са мањим долазним бројевима), као што показује следећа лема. Наиме, код неусмереног графа DFS алгоритам увек може „прећи" преко попречне гране, док код усмереног графа може прећи преко грана усмерених слева надесно (зато оне постају гране графа и никад нису попречне).

И карактеризацију грана усмереног графа у односу на DFS дрво могуће је извршити на основу долазне и одлазне нумерације чворова графа. У наставку ћемо доказати низ лема које доприносе коначној класификацији. На слици 2.19 илустрована је веза вредности долазних и одлазних редних бројева крајева гране и њеног типа.



Слика 2.19: Класификација грана усмереног графа помоћу DFS нумерације. Гране од предака ка потомцима су или гране графа (плаве) или директне гране (жуте) и оне су усмерене од чворова са мањим ка чворовима са већим долазним редним бројевима. Повратне гране (зелене) су усмерене ка чворовима са мањим долазним, али већим одлазним редним бројевима. Попречне гране (црвене) су усмерене налево, ка чворовима са мањим и долазним и одлазним редним бројевима.

Лема 2.3.3 **[Карактеризација грана од предака ка потомцима]**

Нека је $G = (V, E)$ усмерени граф и нека је $T = (V, T)$ DFS дрво графа G. Ако је $(u, v) \in E$ усмерена грана графа G од u до v за коју важи $u.Pre < v.Pre$, онда је чвор u предак чвора v у дрвету T и грана од u до v је или грана дрвета или директна грана.

Доказ. Пошто према долазној DFS нумерацији чвор u претходи чвору v, чвор v је озна-

чен после чвора u. Грана графа (u, v) мора бити разматрана у току рекурзивног позива алгоритма DFS из чвора u. Ако у том тренутку чвор v није означен, онда се грана (u, v) мора укључити у DFS дрво, тј. $(u, v) \in T$, па је тврђење леме тачно. У противном, чвор v је означен у току извођења неког рекурзивног позива DFS из u, па је v потомак чвора u у DFS дрвету T и грана је директна. □

Непостојање попречних грана у односу на DFS дрво које иду слева удесно помаже да се нумерација чворова графа употреби за класификацију четири врсте грана у односу на DFS дрво.

Докажимо следећу карактеризацију повратних грана.

Лема 2.3.4 **[Карактеризација повратних грана]**

Грана (u, v) усмереног графа $G = (V, E)$ је повратна у односу на DFS дрво графа ако и само ако према одлазној нумерацији чвор u претходи чвору v, односно $u.Post < v.Post$ (специјално, ако је грана (u, v) петља, тј. ако је $u = v$, тада важи $u.Post = v.Post$).

Доказ. Размотримо за произвољну грану (u, v) графа G однос редног броја у одлазној DFS нумерацији чворова u и v.

- Ако је (u, v) грана дрвета или директна грана, онда је чвор v потомак чвора u, па важи $v.Post < u.Post$.
- Ако је (u, v) попречна грана, онда због тога што је чвор v лево од чвора u, поново важи $v.Post < u.Post$.
- Ако је (u, v) повратна грана и $v \neq u$, онда је v прави предак чвора u и $v.Post > u.Post$. Међутим, пошто је специјално и петља једна врста повратне гране, за повратну грану може да важи и $v = u$, па самим тим и $v.Post = u.Post$. Према томе, у општем случају за повратну грану (u, v) важи $u.Post \leq v.Post$. □

На основу свега претходног следи следећа лема.

Лема 2.3.5 **[Карактеризација грана DFS дрвета усмереног графа]**

За усмерену грану $(u, v) \in E$ важи:

- *ако је $u.Post \leq v.Post$, онда је грана (u, v) повратна,*
- *ако је $u.Post > v.Post$ и $u.Pre > v.Pre$, онда је грана (u, v) попречна,*
- *ако је $u.Post > v.Post$ и $u.Pre < v.Pre$, онда ако је чвор u родитељ чвора v у DFS дрвету грана (u, v) је грана DFS дрвета, а иначе је (u, v) директна грана.*

У наставку је приказан програм којим се за сваку грану усмереног графа одређује њен тип у односу на DFS дрво, на основу долазне и одлазне нумерације чворова.

```
// vrednosti dolazne i odlazne numeracije
vector<int> dolazna;
vector<int> odlazna;
// naredni slobodni redni broj (dolazni i odlazni)
int rbr_dolazna = 0;
int rbr_odlazna = 0;
// roditelj svakog cvora u DFS drvetu
vector<int> roditelj;

void dfs(int cvor) {
  // cvor dobija naredni dolazni redni broj
  dolazna[cvor] = rbr_dolazna;
  rbr_dolazna++;
  // rekurzivno prolazimo kroz sve susede koje do sada nismo obisli
  for (int sused : listaSuseda[cvor]) {
    if (dolazna[sused] == -1) { // !posecen[sused]
      // 'cvor' je roditelj cvora 'sused' u DFS drvetu
      roditelj[sused] = cvor;
      dfs(sused);
    }
  }
  // cvor dobija naredni odlazni redni broj
  odlazna[cvor] = rbr_odlazna;
  rbr_odlazna++;
}

// klasifikuju se grane DFS drveta dobijenog obilaskom grafa
// krenuvši od datog čvora
void klasifikuj_grane(int cvor) {
  int brojCvorova = listaSuseda.size();
  dolazna.resize(brojCvorova, -1);
  roditelj.resize(brojCvorova, -1);
  odlazna.resize(brojCvorova, -1);

  dfs(cvor);
```

```
    cout << "Dolazna i odlazna numeracija cvorova:" << endl;
    for (int i = 0; i < brojCvorova; i++)
      cout << "Cvor " << i << ": " << dolazna[i]
           << "/" << odlazna[i] << endl;

    cout << "Tipovi grana datog usmerenog grafa: " << endl;
    for (int i = 0; i < listaSuseda.size(); i++)
      for (int j = 0; j < listaSuseda[i].size(); j++) {
        int k = listaSuseda[i][j];

        if (i == roditelj[k])
           cout << i << "-" << k << " je grana DFS drveta" << endl;
        else if (odlazna[i] <= odlazna[k])
           cout << i << "-" << k << " je povratna grana" << endl;
        else // odlazna[i] > odlazna[j]
           if (dolazna[i] < dolazna[k])
              cout << i << "-" << k << " je direktna grana" << endl;
           else
              cout << i << "-" << k << " je poprecna grana" << endl;
      }
}
```

2.3.1.3 Провера постојања циклуса

Показаћемо сада како се алгоритам DFS претраге може искористити за утврђивање да ли је дати граф ациклички.

Проблем

За дати усмерени граф $G = (V, E)$ установити да ли садржи усмерени циклус.

Лема 2.3.6 **[Карактеризација постојања усмереног циклуса у графу]**

Нека је $G = (V, E)$ усмерени граф, и нека је T DFS дрво графа G. Тада граф G садржи усмерени циклус ако и само ако граф G садржи повратну грану у односу на DFS дрво T.

Доказ. Претпоставимо да граф G садржи повратну грану (u, v). Она заједно са гранама DFS дрвета на путу од v до u чини циклус, те импликација у једном смеру датог тврђења директно важи. Покажимо да важи и супротна импликација, односно да ако у графу постоји циклус, тада је нужно једна од његових грана повратна. Заиста, претпоставимо да у графу G постоји циклус који чине гране $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k), (v_k, v_1)$,

од којих нити једна није повратна у односу на DFS дрво T (чворове у циклусу смо означили редом бројевима од 1 до k, кренувши од произвољног чвора тог циклуса – ове ознаке не морају одговарати ознакама чворова графа нити DFS нумерацији чворова). Ако је $k = 1$, односно циклус је петља, онда је сама грана (v_1, v_1) повратна. Ако је пак $k > 1$, претпоставимо да ниједна од грана $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$ није повратна. Према леми 2.3.5 важе неједнакости $v_1.Post > v_2.Post > \cdots > v_k.Post$, из којих следи да је $v_k.Post < v_1.Post$, па је грана (v_k, v_1) према леми 2.3.5 повратна — супротно претпоставци. Тиме је доказано да у сваком циклусу постоји повратна грана у односу на DFS дрво. □

На основу претходне леме закључује се да се алгоритам за проверу да ли граф садржи циклус може свести на DFS претрагу графа, одређивање одлазне DFS нумерације чворова датог графа и проверу постојања повратне гране. Алтернативно, може се током извршавања DFS алгоритма у помоћном стеку одржавати низ чворова на путу од полазног чвора до текућег кроз гране DFS дрвета. Ако се приликом обиласка наиђе на чвор који је већ на стеку, можемо закључити да у графу постоји циклус и прекинути даљи обилазак графа. Сложеност ових алгоритама је $O(|V| + |E|)$.

2.3.2 Обилазак у ширину

Друга важна техника обиласка графа је *обилазак у ширину* или *претрага у ширину* (енгл. breadth–first–search, BFS). Она подразумева обилазак графа на систематичан начин, редом, по удаљености чворова од почетног (где под удаљеношћу неког чвора подразумевамо број грана на најкраћем путу од почетног до тог чвора).

Проблем

Дефинисати алгоритам који кренувши од задатог чвора графа посећује (на пример, исписује или на неки други начин обрађује) све чворове који су достижни од тог чвора, при чему се чворови обрађују у неопадајућем редоследу најкраћих растојања од задатог почетног чвора.

Обилазак у ширину се врши на практично исти начин и код неусмерених и код усмерених графова (разлике до којих може доћи ћемо посебно описати).

Овај начин обиласка је чини погодном за тражење најкраћих путева (путева са најмањих бројем грана) од чвора из кога претрага креће. Размотримо граф познанстава корисника неке друштвене мреже: претрага у ширину може се искористити за проналажење најмање удаљености (у терминима броја особа) између два корисника у мрежи. Слично, у алгоритмима рутирања у мрежама рачунара претрагом у ширину се може одредити најкраћи пут између два чвора у мрежи.

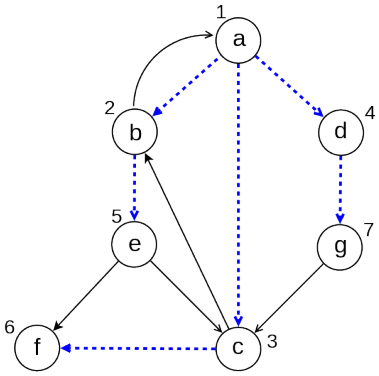
Претрага графа у ширину може бити корисна и у другим контекстима. Веб, као сложена мрежа веб страница међусобно повезаних везама (линковима), се може природно разматрати као један велики и сложен граф чији чворови одговарају појединачним веб страницама, док грана од једног чвора до другог чвора постоји ако и само ако на првој веб страници постоји линк ка другој веб страници. Интересантан проблем јесте индексирање веб страница, које користе претраживачи за веб како би ажурирали свој садржај или индексе садржаја других веб страница. Пошто је у овом проблему циљ да се крећемо систематично почев од неке веб странице, то можемо постићи претрагом графа у ширину, која у првом кораку обилази све странице ка којима са полазне странице постоји веза, у наредном кораку се посећују њихове везе, итд.

Као и у случају обиласку у дубину, обилазак у ширину имплицитно одређује дрво које се у овом случају назива *дрво претраге у ширину* (BFS дрво). Претпоставимо да је граф задат листама повезаности. Ако BFS обилазак покренемо из чвора v, онда чвор v постаје корен BFS дрвета. Након чвора v посећују се сви суседи чвора v редоследом одређеним редоследом у листи повезаности графа и они постају деца чвора v у BFS дрвету (чворови нивоа 1). Затим се долази до свих њихових непосећених суседа, односно до „унука" полазног чвора (чворови нивоа 2), и тако даље. Приликом обиласка чворови се могу нумерисати *BFS бројевима*, слично као при обиласку у дубину. Прецизније, чвор w има BFS број k ако је он k-ти по реду чвор означен у току обиласка алгоритмом BFS. BFS дрво графа формира се укључивањем само грана ка новоозначеним чворовима: лако се показује да је у случају полазног неусмереног графа добијени подграф повезан и да нема циклус, јер од свих грана које воде неком чвору укључујемо тачно једну. Ако граф није повезан, обилазак се може покренути засебно у свакој компоненти повезаности (на тај начин се добија BFS шума).

Запажа се да излазна обрада код обиласка у ширину, за разлику од обиласка у дубину, нема смисла; обилазак нема повратак „навише", већ се, полазећи од корена, креће само наниже.

Пример 2.3.7

За граф приказан на слици 2.20 приказан је поступак обиласка у ширину покренут из чвора a: након чвора a посећују се његови суседи – чворови b, c и d и они постају чворови нивоа 1. Након њих се обилазе непосећени суседи чвора b – то је једино чвор e, па чвор f као једини непосећени сусед чвора c и коначно чвор g као непосећени сусед чвора d: чворови e, f и g постају чворови нивоа 2 у BFS дрвету. Даљом анализом можемо утврдити да ниједан од чворова e, f и g нема непосећених суседа, те се обилазак у ширину завршава. Гране графа које припадају BFS дрвету истакнуте су испрекиданом линијом, а уз сваки чвор приказан је његов редни број у BFS нумерацији.



Слика 2.20: BFS дрво и BFS нумерација усмереног графа.

Како реализовати претрагу у ширину? Чворове обилазимо у жељеном редоследу коришћењем погодне структуре података: све непосећене суседе текућег чвора смештамо у дату колекцију и потребно их је обрадити у редоследу додавања у ту колекцију. У ове сврхе можемо искористити ред као структуру података. Чворови се обележавају чим се поставе у ред (тј. чим им се закаже будући обилазак). Овим се обезбеђује да се ни један чвор неће два пута ставити у ред тј. да ће ред увек садржати различите чворове. Зато је максимални број елемената који могу бити у реду једнак $|V|$, што омогућава да се ред имплементира и уз помоћ обичног низа дужине $|V|$.

Прикажимо на који начин се може реализовати претрага у ширину ако је граф повезан и ако је задат листама повезаности (ако граф није повезан свака његова компонента се обилази засебно). Приликом претраге штампамо чворове графа у редоследу одређеном BFS нумерацијом чворова и конструишемо BFS дрво. BFS дрво ћемо представити листом грана. Претпоставићемо да је граф усмерен, те да при додавању гране (u, v) у BFS дрво не треба додавати и њој симетричну грану (v, u).

```
// pretraga grafa u sirinu pokrenuta iz cvora cvor
// pretpostavljamo da je graf usmeren
void bfs(int cvor) {
  int brojCvorova = listaSuseda.size();
  vector<bool> oznacen(brojCvorova, false);
  vector<vector<int>> bfs_drvo(brojCvorova);

  // red u kom cuvamo cvorove u redosledu kojim ih treba posetiti
  queue<int> red;
  red.push(cvor);
```

```
    oznacen[cvor] = true;
    // stampamo prvi cvor u redosledu BFS numeracije
    cout << cvor << endl;
    while (!red.empty()) {
      // dohvatamo element sa pocetka reda i uklanjamo ga iz kolekcije
      int cvor = red.front();
      red.pop();
      for (int sused : listaSuseda[cvor]) {
        // neoznacene susede tekuceg cvora dodajemo u red
        if (!oznacen[sused]) {
          oznacen[sused] = true;
          // stampamo naredni cvor u skladu sa BFS numeracijom
          cout << sused << endl;
          // dodajemo odgovarajucu granu u bfs drvo
          bfs_drvo[cvor].push_back(sused);
          red.push(sused);
        }
      }
    }
    cout << "Grane BFS drveta su: ";
    for (int i = 0; i < bfs_drvo.size(); i++)
      for (int j = 0; j < bfs_drvo[i].size(); j++)
        cout << "(" << i << "," << bfs_drvo[i][j] << ")" << endl;
}
```

Лако је уверити се да се приликом BFS обиласка датог повезаног графа сваки чвор обрађује по једном и да се свака грана прегледа по једном у случају усмереног графа, односно два пута ако је граф неусмерен (грана се сматра прегледаном када се у току претраге из њеног почетка наиђе на њен крај). Стога је временска сложеност алгоритма претраге у ширину $O(|V| + |E|)$. Слично као и код претраге у дубину, ако је граф представљен матрицом повезаности, сложеност претраге у ширину једнака је $\Theta(|V|^2)$, јер је пролаз кроз све суседе неког чвора сложености $\Theta(|V|)$. Дакле, обично је претрага у ширину ефикаснија када је граф представљен листама повезаности, посебно ако је граф редак.

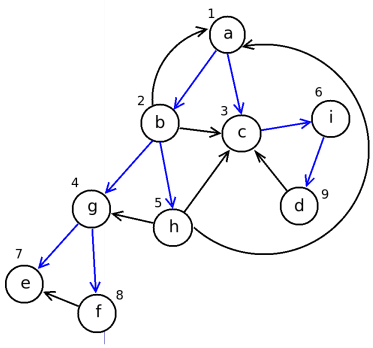
Просторна сложеност алгоритма BFS зависи од максималног броја чворова који ће ред садржати током извршавања алгоритма. У ситуацији када је полазни чвор повезан са свим осталим чворовима, након полазног чвора ће у ред бити додати сви преостали чворови. Дакле, просторна сложеност алгоритма BFS је у најгорем случају $O(|V|)$.

Пример 2.3.8

Размотримо пример извршавања алгоритма обиласка у ширину покренутог из чвора a на примеру графа са слике 2.21. Нека редослед суседа чворова у листи повезаности којом је представљен граф одговара лексикографски растућем поретку ознака чворова.

- *Најпре из реда узимамо и посећујемо чвор a, а у ред додајемо његове непосећене суседе: чвор b и чвор c.*
- *Из реда узимамо и посећујемо чвор b, а додајемо његове непосећене суседе: чворове g и h редом. Затим из реда узимамо и посећујемо чвор c, а у ред додајемо његове непосећене суседе: чвор i.*
- *Из реда узимамо и посећујемо чвор g, а у ред додајемо његове непосећене суседе: чворове e и f. Затим из реда узимамо и посећујемо чвор h који нема непосећених суседа. Затим из реда узимамо и посећујемо чвор i, а у ред додајемо његове непосећене суседе: чвор d.*
- *Након овога редом из реда узимамо и посећујемо чворове e, f и d и закључујемо да ниједан од њих нема непосећених суседа и претрага се завршава.*

Стање реда током обиласка је приказано на слици 2.21, десно. Гране BFS дрвета су на слици 2.21 лево приказане плавом бојом, а уз сваки чвор приказан је његов број у оквиру BFS нумерације.

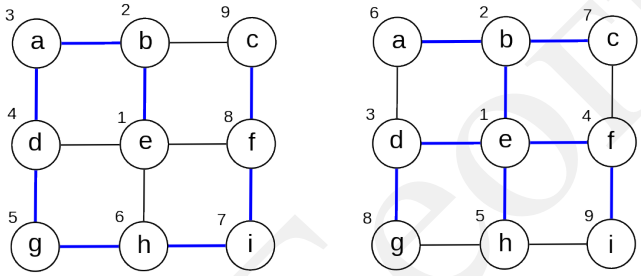


Tekući čvor	Sadržaj reda							
-	a							
a		b	c					
b			c	g	h			
c				g	h	i		
g					h	i	e	f
h						i	e	f
i							e	f	d
e								f	d
f									d
d									

Слика 2.21: Пример обиласка у ширину усмереног графа.

Пример 2.3.9

На слици 2.22 приказани су обилазак у дубину и обилазак у ширину једног неусмереног графа. Претпостављамо да је граф представљен листама повезаности и да су чворови у листама повезаности наведени у лексикографски растућем поретку. У левом делу слике приказан је поступак претраге у дубину покренут из чвора a: уз сваки чвор приказан је његов редни број у долазној DFS нумерацији, а плавом бојом су истакнуте гране које припадају DFS дрвету графа. У десном делу слике илустрован је поступак претраге у ширину покренут из чвора a: уз сваки чвор приказан је његов редни број у BFS нумерацији, а плавом бојом су истакнуте гране које припадају BFS дрвету графа.



Слика 2.22: Разлика између DFS и BFS дрвета за обилазак графа покренут из средишњег чвора. Плавом бојом означене су гране које припадају DFS и BFS дрвету, редом. Чворови су нумерисани редоследом којим се обилазе.

Приметимо да код неусмерених графова у односу на BFS дрво могу постојати једино гране BFS дрвета и попречне гране (сетимо се да у неусмереном графу у односу на DFS дрво нису могле да постоје попречне гране).

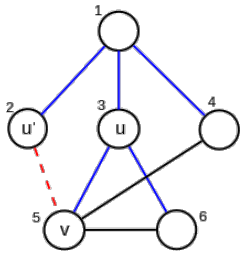
У наставку ћемо формулисати неколико тврђења која важе за обилазак у ширину, односно за BFS дрво графа.

Лема 2.3.7

Ако грана (u, v) припада BFS дрвету графа G и чвор u је родитељ чвора v, онда чвор u има најмањи BFS број међу чворовима из којих постоји грана ка v.

Доказ. Претпоставимо супротно, да у графу G постоји грана (u', v), таква да чвор u' има мањи BFS број од чвора u, при чему и u и u' оба имају мање BFS бројеве од чвора v (слика 2.23), као и да је u' чвор са најмањим BFS бројем који задовољава претходни услов. У тренутку обраде чвора u' је онда чвор v морао бити уписан у ред, па је грана (u', v) морала бити укључена у BFS дрво. Међутим, према претпоставци, грана (u, v)

је укључена у BFS дрво, па не може истовремено бити укључена и грана (u', v), те добијамо контрадикцију. □



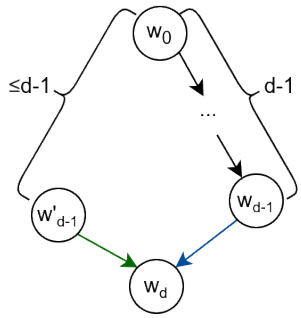
Слика 2.23: Илустрација уз доказ леме 2.3.7. Плавом бојом истакнуте су гране BFS дрвета, а уз сваки чвор приказан је редни број у BFS нумерацији чвора.

Дефинишимо *растојање* $\mathrm{d}(u, v)$ између чворова u и v као дужину најкраћег пута од чвора u до чвора v, с тим да се под дужином пута подразумева број грана које чине тај пут. Ниво чвора v у односу на BFS дрво са кореном u једнак је растојању $\mathrm{d}(u, v)$.

Лема 2.3.8

Пут од корена r BFS дрвета до произвољног чвора w кроз BFS дрво најкраћи је пут од чвора r до чвора w у графу G.

Доказ. Индукцијом по d доказаћемо да до сваког чвора w на растојању d од корена r (јединствени) пут кроз BFS дрво од r до w има дужину тачно d. За $d = 0$ тврђење је тривијално тачно: једини чвор на растојању 0 од корена r је он сâм, а пут кроз дрво од r до r такође има дужину 0. Претпоставимо да је тврђење тачно за све чворове који су на растојању мањем од d од корена r, и нека је w неки чвор на растојању d од корена; другим речима, постоји низ чворова $(w_0 = r, w_1, w_2, \ldots, w_{d-1}, w_d = w)$ који чине пут дужине d од r до w, и не постоји краћи пут од r до w. Пошто је дужина најкраћег пута од r до w_{d-1} једнака $d-1$, према индуктивној хипотези пут од r до w_{d-1} кроз BFS дрво има дужину $d-1$. У тренутку обраде чвора w_{d-1}, ако чвор w_d није означен, пошто у графу G постоји грана (w_{d-1}, w_d), та грана се укључује у BFS дрво, па до чвора w_d постоји пут дужине d кроз BFS дрво (слика 2.24). У противном, ако је у том тренутку чвор w_d већ означен, онда до чвора w_d кроз BFS дрво води грана из неког чвора w'_{d-1}, означеног пре w_{d-1}, из чега следи да је ниво чвора w'_{d-1} највише $d-1$, што значи да кроз дрво постоји пут до чвора w'_{d-1} чија је дужина највише $d-1$. Она не може бити мања од $d-1$, јер би тада у графу постојао пут од r до w_d краћи од d, што је супротно претпоставци да је w_d на растојању d од корена. Дакле, дужина пута од r до w'_{d-1} у дрвету мора бити тачно $d-1$, па пошто грана (w'_{d-1}, w_d) припада дрвету, растојање од r до w_d у дрвету је тачно d. □



Слика 2.24: Илустрација уз доказ леме 2.3.8.

Лема 2.3.9

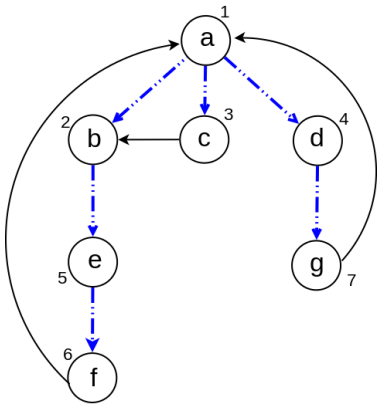
Ако је граф $G = (V, E)$ неусмерен и $(v, w) \in E$ нека његова произвољна грана, онда се нивои чворова v и w разликују највише за један.

Доказ. Претпоставимо да је од чворова v и w, чвор v први достигнут BFS претрагом и нека је његов ниво d. Тада је ниво чвора w већи или једнак од d. С друге стране, ниво чвора w није већи од $d + 1$, јер до њега води грана BFS дрвета или из чвора v, или из неког чвора који је означен пре v. Дакле, ниво чвора w је или d или $d + 1$, те тврђење леме важи. □

Приметимо да аналогно тврђење не важи нужно у случају да је граф усмерен. На слици 2.25 је приказан усмерен граф који садржи гране које повезују два чвора чији се нивои разликују за више од 1: на пример гране (f, a) и (g, a).

2.3.2.1 Покретање обиласка из више чворова истовремено

Уобичајено је да се обилазак графа покреће из једног полазног чвора, међутим, некада је обилазак у ширину могуће покренути и из више чворова истовремено. Претпоставимо, рецимо, да су у датом графу неки чворови црвени, а неки плави и да нас занима најкраће растојање (мерено бројем грана јединичне дужине) између неког црвеног и плавог чвора. Директан приступ би подразумевао да се покрене обилазак у ширину из сваког црвеног чвора, да се измери растојање од њега до њему најближег плавог чвора и да се одреди минимум тако добијених вредности. То би захтевало да се обилазак у ширину покреће изнова за сваки црвени чвор, што може бити неефикасно (сложеност може бити чак $O(|V|^3)$). Ефикаснији приступ је да се обилазак покрене истовремено из свих црвених чворова, што значи да се у почетном кораку сви истовремено додају у ред и да се сваком од њих придружи растојање 0. Након тога, обилазак у ширину би се настављао на уобичајени начин (узимањем елемента са почетка реда и додавањем у ред његових неозначених суседа уз растојање увећано за један у односу на растојање



Слика 2.25: Илустрација да лема 2.3.9 не важи за усмерене графове (гране (f, a) и (g, a) спајају чворове који нису на суседним нивоима).

елемента са почетка реда), све док се не наиђе на први плави чвор, чиме би се добило тражено најкраће растојање између црвених и плавих чворова. Пошто се врши само један обилазак, сложеност овог алгоритма је $O(|V| + |E|)$, што је у најгорем случају $O(|V|^2)$.

Задатак: Постављање рачунарске мреже

У једном рачунарском кабинету потребно је успоставити необичну рачунарску мрежу. Потребно је поставити рачунаре на n столова, при чему на неким столовима могу да стоје обични рачунари (њих имамо на располагању у неограниченом броју), а на неким посебни сервери (њих посебно морамо купити по цени од c_s динара). Неке парове столова тј. рачунара на њима је могуће повезати директно мрежним кабловима, а неке није. Цена успостављања мрежног кабла између било која два стола је c_k динара. Написати програм који одређује најмању цену коју је потребно платити тако да је сваки рачунар или сервер или је мрежним каблом повезан са бар једним сервером (не обавезно директно).

Опис улаза

Са стандардног улаза се у првом реду уносе бројеви c_s ($0 \leq c_s \leq 1000$) и c_k ($0 \leq c_k \leq 1000$), раздвојени размаком. У наредном реду се уноси број столова n ($2 \leq n \leq 50000$) и m број парова столова између којих је могуће поставити мрежни кабл $2 \leq m \leq \frac{n(n-1)}{2}$, раздвојени размаком. У наредних m редова уносе се парови бројева између 0 и $n-1$, раздвојених размаком који одређују столове између којих је могуће поставити кабл.

Опис излаза

На стандардни излаз исписати тражену најмању цену.

Пример

Улаз	*Излаз*
850 350 7 6 0 1 0 4 4 2 1 4 3 5 6 5	3450

Решење

Проблем моделујемо неусмереним графом ком су чворови столови, а гране постоје између свака два чвора (тј. стола) између којих је могуће успоставити кабл.

Ако је цена сервера мања или једнака цени једног кабла (иако та претпоставка није реална), тада је најјефтиније на сваки сто поставити по један сервер. Наиме замена сваког сервера обичним рачунаром подразумева повезивање тог рачунара са неким сервером помоћу једног кабла, што повећава укупну цену. Оптимална цена у том случају је једнака производу броја рачунара n и цене сервера c_s.

У супротном је оптимално у свакој компоненти повезаности графа. поставити по један сервер и све остале столове у тој компоненти попунити обичним рачунарима повезаним са тим сервером. Заиста, ако у некој компоненти не би постојао бар један сервер, онда рачунари у тој компоненти не би могли бити повезани ни са једним сервером, што је супротно условима задатка. Уколико би постојала бар два сервера у некој компоненти, цена не би била оптимална. Наиме један од та два сервера би се могао заменити обичним рачунаром који би се каблом повезао са другим сервером у тој компоненти, чиме би се уместо цене сервера платила цена кабла која је мања. Нека је k број компонената повезаности. Потребно је успоставити k сервера (по један у свакој компоненти), а остале рачунаре, њих $n - k$ повезати каблом са по једним рачунаром. Зато је укупна цена $k \cdot c_s + (n - k) \cdot c_k$.

Број компонената повезаности можемо једноставно одредити обиласком графа (на пример, рекурзивно имплементираним обиласком у дубину).

```
// obilaskom u dubinu krenuvsi od datog cvora se oznacavaju svi cvorovi
// koji pripadaju njegovoj komponenti povezanosti
```

```
void dfs(const vector<vector<int>>& susedi,
         vector<int>& komponente,
         int cvor, int komponenta) {
  komponente[cvor] = komponenta;
  for (int sused : susedi[cvor])
    if (komponente[sused] == 0)
      dfs(susedi, komponente, sused, komponenta);
}

// odredjuje broj komponenta povezanosti datog grafa
int broj_komponenata_povezanosti(const vector<vector<int>>& susedi,
                                 int broj_cvorova) {
  vector<int> komponente(broj_cvorova, 0);
  int komponenta = 0;
  for (int cvor = 0; cvor < broj_cvorova; cvor++)
    if (komponente[cvor] == 0)
      dfs(susedi, komponente, cvor, ++komponenta);
  return komponenta;
}

int main() {
  // ucitavamo podatke o racunarima i cenama
  int broj_racunara, broj_kablova;
  long long cena_servera, cena_kabla;
  cin >> cena_servera >> cena_kabla;
  cin >> broj_racunara >> broj_kablova;
  vector<vector<int>> susedi(broj_racunara);
  for (int i = 0; i < broj_kablova; i++) {
    int racunar1, racunar2;
    cin >> racunar1 >> racunar2;
    susedi[racunar1].push_back(racunar2);
    susedi[racunar2].push_back(racunar1);
  }

  if (cena_servera <= cena_kabla)
    // na svaki sto se postavlja server
    cout << broj_racunara * cena_servera << endl;
  else {
    // u svakoj komponenti povezanosti se kupuje po jedan server
```

```
    // a ostali racunari se povezuju kablovima sa njima
    int broj_servera = broj_komponenata_povezanosti(susedi, broj_racunara);
    int broj_obicnih = broj_racunara - broj_servera;
    cout << broj_servera * cena_servera + broj_obicnih * cena_kabla << endl;
  }

  return 0;
}
```

Задатак: Провера да ли је граф бипартитан

Група студената се окупила на летњем кампу. Свако је знао неколико других студената. Испоставило се да се сваки пар студената познаје посредно (преко низа заједничких познаника).

Потребно је да се студенти поделе у две групе, али пошто свако жели да упозна што више нових колега, потребно је да сваку групу чине међусобно непознате особе (две особе које се већ познају не могу бити у истој групи). Написати програм који одређује да ли је то могуће и ако јесте, који ће све студенти бити у групи са студентом чији је редни број 0.

Опис улаза

Са стандардног улаза се учитава број студената n ($1 \leq n \leq 10^5$), број парова m студената који се од раније познају ($0 \leq m \leq \frac{n(n-1)}{2}$), а затим и низ парова бројева од 0 до $n-1$ који представљају њихова познанства.

Опис излаза

На стандардни излаз исписати редне бројеве студената који су у групи са студентом који носи редни број 0, у растућем поретку, или симбол - ако тражене две групе није могуће формирати.

Пример 1

Улаз	*Излаз*	*Објашњење*
6 6 0 1 1 2 2 3 3 4 4 5 5 0	0 2 4	Ако су у једној групи студенти са бројевима 0, 2 и 4, у другој су студенти са бројевима 1, 3 и 5 и тада се ни у једној групи не налазе студенти који се међусобно познају.

Пример 2

Улаз	*Излаз*	*Објашњење*
5 5 0 1 1 2 2 3 3 4 4 0	-	Студент 0 не сме да буде у групи са студентом 1, који не сме да буде у групи са студентом 2, што значи да 0 и 2 морају да буду у истој групи. Студенти 2 и 3 не могу да буду у истој групи, па су 1 и 3 у истој групи. Студент 4 не сме да буде у групи са студентом 3, па он мора бити у групи са студентима 0 и 2, међутим, то није допуштено, јер се студенти 4 и 0 познају.

Решење

Студенте и њихова познанства можемо представити неусмереним графом. Поставља се питање да ли је тај граф *бипартитан* тј. да ли му се чворови могу поделити у две групе тако да све гране спајају чворове из две различите групе.

Ако неки чвор припада левој половини, тада сви његови суседи припадају десној половини, њихови суседи левој половини, њихови суседи десној и тако даље. Зато се задатак може решити обиласком графа (на пример у дубину) обележавајући чворове наизменично (за сваки непосећени чвор се обележава да ли припада левој или десној половини). Ако се приликом обиласка наиђе на чвор чији је сусед већ обележен тако да припада истој половини као текући, тада граф није бипартитан. Ако се на такав чвор не наиђе, тада граф јесте бипартитан.

По условима задатка сви студенти се познају посредно, што значи да је граф повезан. Ако граф не би био повезан, поступак претраге у дубину и означавања чворова би требало поновити за сваку компоненту повезаности засебно.

```
bool moze = true;
// svakom cvoru dodeljujemo jednu od dve boje (tj. oznake grupa)
vector<int> boje(n, -1);
int boja = 0;
```

```
stack<int> stek;
stek.push(0);
// prvi cvor bojimo prvom bojom
boje[0] = 0;
while (!stek.empty() && moze) {
  int x = stek.top(); stek.pop();
  // susedi trenutnog cvora x treba da budu obojeni suprotnom bojom od x
  boja = 1 - boje[x];
  for (int sused : susedi[x]) {
    // sused je vec obojen pogresnom bojom, pa graf nije bipartitan
    if (boje[sused] != -1 && boje[sused] != boja) {
      moze = false;
      break;
    }
    // sused nije obojen, pa ga bojimo suprotnom bojom od tekuceg cvora x
    if (boje[sused] == -1) {
      boje[sused] = boja;
      stek.push(sused);
    }
  }
}
```

Задатак: Авионска преседања

Једна авио-компанија заједно са својим партнерима изводи летове између познатих светских аеродрома. Напиши програм који одређује да ли је могуће да се коришћењем тих летова стигне са једног на други дати аеродром и ако јесте, колико је најмање летова потребно.

Опис улаза

Са стандардног улаза се задаје број m ($1 \leq m \leq 100$) летова које компанија изводи, а затим у наредних m редова опис тих летова (шифра полазног и шифра долазног аеродрома, раздвојени размаком). Након тога се уноси број k ($1 \leq k \leq 100$) путника који су заинтересовани за летове које пружа та компанија, и у наредних k линија описи релација на којима они путују (шифра полазног и шифра долазног аеродрома, раздвојени размаком).

Опис излаза

За сваког од k путника на стандардни излаз исписати најмањи број летова помоћу којих

могу да остваре жељено путовање или реч `ne` ако такво путовање није могуће остварити помоћу летова које компанија изводи.

Пример

Улаз	*Излаз*	*Објашњење*

`7`
`BEG FRA`
`FRA MUC`
`FRA JFK`
`BEG MUC`
`MUC LAX`
`LAX JFK`
`LAX ORD`
`3`
`BEG JFK`
`MUC BEG`
`BEG ORD`
	`2`
	`ne`
	`3`
		Од Београда (`BEG`) до Њујорка (`JFK`) може се стићи преко Франкфурта (`FRA`). Од Минхена (`MUC`) до Београда (`BEG`) није могуће организовати путовање. Од Београда (`BEG`) до Чикага (`ORD`) могуће је путовати преко Минхена (`MUC`) и Лос Анђелеса (`LAX`).

Решење

Аеродроми и летови између њих се могу представити усмереним графом. Најкраће путеве у том графу можемо најједноставније пронаћи претрагом у ширину. Њу имплементирамо тако што у ред стављамо чворове у редоследу њиховог растојања од полазног чвора. На почетку стављамо полазни чвор, а затим у сваком кораку скидамо чвор са почетка реда и у ред додајемо његове суседе који раније нису посећени. Уз сваки чвор у ред постављамо и његово растојање од полазног чвора. У тренутку када у ред треба ставити долазни чвор, знамо његово најкраће растојање. Ако се ред испразни пре него што се долазни чвор постави у њега, онда полазни и долазни чвор нису повезани.

```
// pretragom u sirinu odredjujemo najkrace rastojanje izmedju dva aerodroma
// graf je predstavljen listama povezanosti, pri čemu su oznake
// cvorova niske, a ne redni brojevi
int brojPresedanjaBFS(const string& aerodromOd, const string& aerodromDo,
                      const map<string, vector<string>>& letovi) {
  // skup posecenih aerodroma
  set<string> posecen;
  // red potreban za implementaciju obilaska u sirinu
  queue<pair<string, int>> red;
  // krecemo od polaznog aerodroma
  red.emplace(aerodromOd, 0);
  while (!red.empty()) {
```

```
    // uzimamo tekuci aerodrom iz reda i njegovo rastojanje od polaznog
    string aerodrom;
    int rastojanje;
    tie(aerodrom, rastojanje) = red.front();
    red.pop();
    posecen.insert(aerodrom);

    // lista suseda tekuceg aerdroma
    auto it = letovi.find(aerodrom);
    if (it != letovi.end()) {
      // prolazimo kroz sve letove sa tekuceg aerodroma
      for (const string& sused : it->second) {
        // ako su poseceni, preskacemo ih
        if (posecen.find(sused) != posecen.end())
          continue;
        // ako smo dostigli dolanzi aerodrom, znamo najkrace
        // rastojanje do njega
        if (sused == aerodromDo)
          return rastojanje+1;
        // dodajemo susedni aerodrom u red
        red.emplace(susedni, rastojanje+1);
      }
    }
  }
  // dolazni aerodrom nije dostizan
  return -1;
}

int main() {
  // graf je predstavljen listama povezanosti, pri čemu su oznake
  // cvorova niske, a ne redni brojevi
  int m;
  cin >> m;
  map<string, vector<string>> letovi;
  for (int i = 0; i < m; i++) {
    string aerodromOd, aerodromDo;
    cin >> aerodromOd >> aerodromDo;
    letovi[aerodromOd].push_back(aerodromDo);
  }
```

```
  // odgovaramo na k upita
  int k;
  cin >> k;
  for (int i = 0; i < k; i++) {
    string aerodromOd, aerodromDo;
    cin >> aerodromOd >> aerodromDo;
    int broj = brojPresedanjaBFS(aerodromOd, aerodromDo, letovi);
    if (broj == -1)
      cout << "ne" << endl;
    else
      cout << broj << endl;
  }
  return 0;
}
```

Задатак: Косе црте

Испред дворца, краљ има врт правоугаоног облика који је подељен у мрежу $m \times n$ квадрата. По обиму правоугаоника, као и дуж дијагонала неких од тих квадрата засадио је живу ограду и тако је направио један необичан лавиринт. Написати програм који одређује на колико области је подељен тај лавиринт (из једне области се не може доћи у другу ако се не прескочи жива ограда).

Опис улаза

Са стандардног улаза се учитавају димензије правоугаоника m и n ($1 \leq m, n \leq 50$), а затим матрица карактера димензије $m \times n$ која описује појединачне квадрате. Карактер `\` означава да је ограда постављена дуж главне, карактер `/` да је ограда постављена дуж споредне дијагонале, а размак да у том квадрату нема живе ограде.

Опис излаза

На стандардни излаз исписати тражени број области.

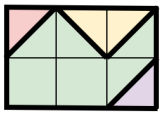
Пример 1

Улаз	*Излаз*	*Објашњење*

`2 2`
`\/`
`/\`
	`4`	Лавиринт и његове четири области су приказани на слици. 

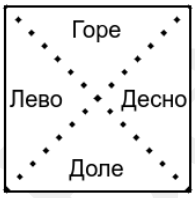
Пример 2

Улаз	*Излаз*	*Објашњење*
2 3 /\/ /	4	Лавиринт и његове четири области су приказани на слици.



Решење

Прилично је јасно да се задатак може решити тако што се преброје компоненте повезаности у графу који градимо на основу података о лавиринту, међутим, прво треба на прави начин дефинисати тај граф. Један начин је да сваки квадрат у лавиринту поделимо на четири области и да свака област представља један чвор графа (тај граф има $4mn$ чворова).



Области у једном квадрату су повезане, осим ако је постављена нека жива ограда.

- Из горње области можемо доћи у леву, осим ако је жива ограда на главној дијагонали и у десну, осим ако је жива ограда на споредној дијагонали.
- Из доње области можемо доћи у леву, осим ако је жива ограда на споредној дијагонали и у десну, осим ако је жива ограда на главној дијагонали.
- Из леве области можемо доћи у горњу, осим ако је жива ограда на главној дијагонали и у доњу, осим ако је жива ограда на споредној дијагонали.
- Из десне области можемо доћи у горњу, осим ако је жива ограда на споредној дијагонали и у доњу, осим ако је жива ограда на главној дијагонали.

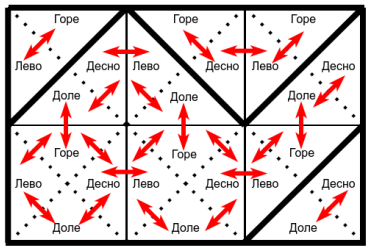
Могући су преласци и из једног у други квадрат.

- Из горње области било ког квадрата можемо доћи у доњу област квадрата изнад тог квадрата (ако такав квадрат постоји).
- Из доње области било ког квадрата можемо доћи у горњу област квадрата испод тог квадрата (ако такав квадрат постоји).
- Из леве области било ког квадрата можемо доћи у десну област квадрата лево тог квадрата (ако такав квадрат постоји).

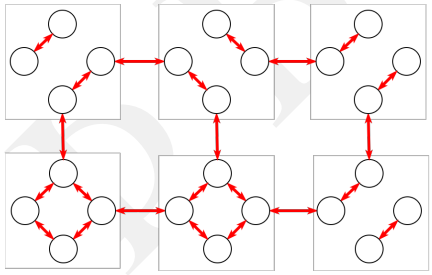
- Из десне области било ког квадрата можемо доћи у леву област квадрата десно од тог квадрата (ако такав квадрат постоји).

На наредној слици приказана је повезаност области у лавиринту који се описује карактерима:

```
/\/
  /
```



Одговарајући граф је приказан на наредној слици.



Пошто квадрата има mn, чворова графа има $4mn$. Сваки чвор је повезан са највише 3 друга чвора, па је укупна сложеност пребројавања компоненти $O(mn)$.

```
// svakom polju odgovaraju ima 4 čvora grafa
enum deo {GORE=0, DOLE, LEVO, DESNO};

// obilazak grafa zadatog nizom stringova od cvora sa koordinatama (v0, k0, d0)
// to je cvor u vrsti v0, koloni k0 i delu d0
// poseceni cvorovi su određeni visedimenzionim nizom posecen
void dfs(const vector<string>& linije, int m, int n,
```

```
        vector<vector<vector<bool>>>& posecen,
        int v0, int k0, int d0) {
  // na steku cuvamo koordinate cvorova grafa
  stack<tuple<int, int, int>> s;
  s.emplace(v0, k0, d0);
  posecen[v0][k0][d0] = true;

  while (!s.empty()) {
    // skidamo koordinate tekuceg cvora sa steka
    int v, k, d;
    tie(v, k, d) = s.top(); s.pop();

    // odredjujemo susede tekuceg cvora (ima ih najvise 3)
    vector<tuple<int, int, int>> susedi;
    // analiziramo polozaj tekuceg cvora u njegovom polju
    switch(d) {
    case GORE:
      // levi i desni cvor tekuceg polja (ako nisu zaklonjeni preprekama)
      if (linije[v][k] != '\\')
        susedi.emplace_back(v, k, LEVO);
      if (linije[v][k] != '/')
        susedi.emplace_back(v, k, DESNO);
      // donji cvor polja iznad (ako to polje postoji)
      if (v > 0)
        susedi.emplace_back(v-1, k, DOLE);
      break;
    case DOLE:
      // levi i desni cvor tekuceg polja (ako nisu zaklonjeni preprekama)
      if (linije[v][k] != '/')
        susedi.emplace_back(v, k, LEVO);
      if (linije[v][k] != '\\')
        susedi.emplace_back(v, k, DESNO);
      // gornji cvor polja ispod (ako to polje postoji)
      if (v < m-1)
        susedi.emplace_back(v+1, k, GORE);
      break;
      // ...
    }
```

```
    // prolazimo kroz niz suseda tekuceg cvora
    for (const auto& t : susedi) {
      int vv, kk, dd;
      tie(vv, kk, dd) = t;
      // neposecene susede stavljamo na stek
      if (!posecen[vv][kk][dd]) {
        posecen[vv][kk][dd] = true;
        s.push(t);
      }
    }
  }
}

int main() {
  // ucitavamo crtez lavirinta
  int m, n;
  cin >> m >> n;
  string s;
  getline(cin, s);
  vector<string> linije(m);
  string linija;
  for (int i = 0; i < m; i++)
    getline(cin, linije[i]);

  // alociramo vektor posecenosti cvorova
  // na pocetku nije posecen ni jedan cvor
  vector<vector<vector<bool>>> posecen;
  posecen.resize(m);
  for (int i = 0; i < m; i++) {
    posecen[i].resize(n);
    for (int j = 0; j < n; j++)
      posecen[i][j].resize(4, false);
  }

  // odredjujemo komponente povezanosti grafa i ispisujemo njihov broj
  int brojOblasti = 0;
  for (int v = 0; v < m; v++)
    for (int k = 0; k < n; k++)
      for (int d = GORE; d <= DESNO; d++)
```

```
        if (!posecen[v][k][d]) {
          dfs(linije, m, n, posecen, v, k, d);
          brojOblasti++;
        }

  cout << brojOblasti << endl;
  return 0;
}
```

2.4 Тополошко сортирање

Претпоставимо да је задат скуп послова у вези са чијим редоследом извршавања постоје нека ограничења. Неки послови зависе од других, односно не могу се започети пре него што се ти други послови заврше. Све зависности су познате, а циљ је направити такав редослед извршавања послова који задовољава сва задата ограничења; другим речима, тражи се редослед извршавања за који важи да сваки посао започиње тек кад су завршени сви послови од којих он зависи. На пример, желимо да саградимо кућу. Да бисмо то успели потребно је да поставимо темељ, да сазидамо зидове, да ставимо кров, да уведемо струју и воду. Притом, наравно, није могуће, на пример, сазидати зидове док се не постави темељ, нити увести воду док се не стави кров на кућу. Задатак је одредити неки исправан редослед извршавања ових послова.

Дати проблем има примену у разним доменима: на пример, за одређивање редоследа у ком је потребно извршити поновно израчунавање вредности формула у програмима за табеларна израчунавања, за утврђивање редоследа у ком треба извршити задатке у мејкфајлу, за унапређење паралелизма инструкција и слично. Задатак је осмислити ефикасан алгоритам за формирање таквог редоследа.

Описани проблем може се формулисати у терминима графова и назива се *тополошко сортирање графа* (енгл. topological sort). Наиме, задатим пословима и њиховим међузависностима може се на природан начин придружити усмерени граф $G = (V, E)$: сваком послу придружује се чвор, а усмерена грана од чвора x до чвора y постоји ако се посао y не може започети пре завршетка посла x.[2] Задатак је одредити *тополошки поредак чворова*, односно нумерацију чворова бројевима од 1 до $n = |V|$ (или од 0 до $n - 1$) тако да за сваку грану графа (u, v) важи да је полазни чвор u гране нумерисан

[2] Дакле, гране воде од посла који је независан, ка послу који је зависан. Могуће је формирати граф тако да гране воде од зависног ка независном послу. Различитим алгоритмима одговарају различите организације грана, па приликом конструкције алгоритма за тополошко сортирање треба обратити пажњу на то како су гране усмерене.

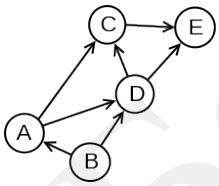
мањом вредношћу него завршни v чвор те гране. Граф над којим разматрамо овај проблем мора бити без усмерених циклуса, јер се у противном неки послови никада не би могли започети.

Проблем

У задатом усмереном ацикличком графу $G = (V, E)$ са n чворова нумерисати чворове бројевима од 1 до n, тако да ако је произвољан чвор v нумерисан бројем k, онда су сви чворови до којих постоји усмерена грана из чвора v нумерисани бројевима већим од k.

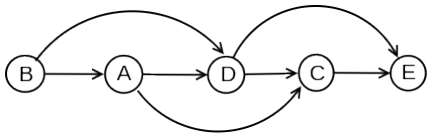
Пример 2.4.1

Размотримо граф приказан на слици 2.24.



Слика 2.24: Усмерени ациклички граф у којем постоји тачно једно тополошко уређење чворова.

У њему постоји само једно исправно тополошко уређење чворова и то је B, A, D, C, E. Чвор D, рецимо, мора да буде нумерисан већим бројем од чворова B и A јер постоје гране до D из чворова A и B. Слично чвор D мора бити нумерисан мањим бројем од чворова C и E јер постоје гране од чвора D до чворова C и E. Дакле, у овом графу редни број чвора D мора бити 3. Граф са слике 2.24 можемо представити и погодније, тако да чворови буду поређани дуж једне праве уређени у односу на тополошки поредак. Тада су све гране графа усмерене удесно (слика 2.25).

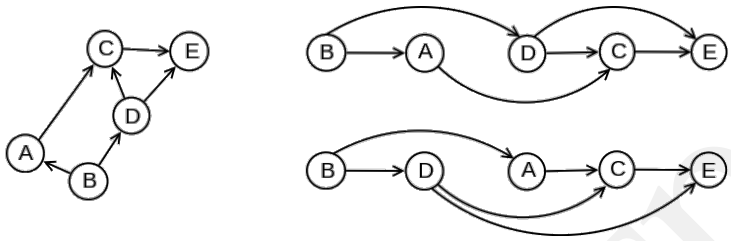


Слика 2.25: Усмерени ациклички граф код кога су чворови поређани у редоследу тополошког уређења.

У општем случају може постојати већи број исправних тополошких уређења графа.

Пример 2.4.2

Ако размотримо граф са слике 2.26, лево, у њему постоје два исправна тополошка уређења: B, A, D, C, E и B, D, A, C, E. Они су приказани на слици 2.26, десно.



Слика 2.26: Усмерени ациклички граф у којем постоје два различита тополошка уређења чворова.

Размотрићемо два различита алгоритма за одређивање тополошког уређења у ацикличком усмереном графу: *Канов*[3] *алгоритам* (енгл. Kahn's algorithm) и *алгоритам заснован на претрази графа у дубину*.

2.4.1 Канов алгоритам

Природно је започети алгоритам тако што урадимо онај посао који не зависи ни од једног другог посла. Након тога нам проблем постаје једноставнији, јер неки послови који су зависили од тог првог посла више не зависе од њега. Дакле, проблем решавамо свођењем на мањи потпроблем истог типа, тј. индуктивно-рекурзивном конструкцијом.

Природна је, дакле, наредна индуктивна хипотеза.

Индуктивна хипотеза. Умемо да нумеришемо на захтевани начин чворове свих усмерених ацикличких графова са мање од n чворова.

Базни случај је случај празног графа, који не садржи чворове и он се тривијално решава. Као и обично, посматрајмо произвољни граф са n чворова, уклонимо један чвор из њега, применимо индуктивну хипотезу на преостале чворове у графу и покушајмо да проширимо нумерацију на полазни граф. Оно што је важно приметити јесте да имамо слободу избора чвора који уклањамо па, као што смо рекли, бирамо чвор са улазним степеном нула; њему се може доделити број 1. Поставља се питање да ли у произвољном усмереном ацикличком графу увек постоји чвор са улазним степеном нула? Интуитивно се намеће потврдан одговор, јер се са означавањем негде мора започети. Следећа лема потврђује ову чињеницу.

[3] Алгоритам је 1962. први описао Артур Кан, амерички информатичар.

Лема 2.4.1

Усмерени ациклички граф увек има чвор улазног степена нула.

Доказ. Ако би сви чворови графа имали позитивне улазне степене, могли бисмо да кренемо из неког чвора „уназад” пролазећи гране у супротном смеру. Међутим, број чворова у графу је коначан, па се у том обиласку мора у неком тренутку наићи на неки чвор по други пут, што значи да у графу постоји усмерени циклус. Ово је супротно претпоставци да се ради о ацикличком графу. Дакле у усмереном ацикличком графу увек постоји чвор чији је улазни степен нула.[4] □

Претпоставимо да смо пронашли чвор чији је улазни степен нула. Нумеришимо га са 1, уклонимо све гране које воде из њега, и нумеришимо остатак графа (који је такође ациклички) бројевима од 2 до n: према индуктивној хипотези они се могу нумерисати бројевима од 1 до $n-1$, а затим се сваки редни број може повећати за један. Напоменимо још у овом тренутку да није неопходно ефективно избацивати гране из графа, јер је то операција која се не извршава ефикасно ако је граф представљен листом повезаности. Наиме, одговарајући ефекат могуће је реализовати и ефикасније, смањивањем за 1 улазног степена чвора у који дата грана улази.

Дакле, проблем се може решити сукцесивним проналажењем чворова са улазним степеном 0. При реализацији овог алгоритма треба проналазити чвор са улазним степеном нула и ажурирати улазне степене чворова после уклањања грана које полазе из датог чвора. Други проблем можемо решити тако што алоцирамо низ `ulazniStepen` димензије једнаке броју чворова у графу и иницијализујемо га на вредности улазних степена чворова. Улазне степене чворова у графу можемо једноставно одредити проласком кроз скуп свих грана у произвољном редоследу и повећавањем за један вредности `ulazniStepen[w]` сваки пут кад се наиђе на неку грану (v, w). Уколико је граф задат листом повезаности, све гране су наведене у листи повезаности којом је представљен и овај корак је линеарне сложености по броју грана у графу. Приликом уклањања чвора v, свим његовим суседима w вредност улазног степена која се налази у низу на позицији w смањујемо за 1 (што је операција сложености $O(1)$). Свака грана ће бити уклоњена тачно једном, тако да ће укупна сложеност опет бити линеарна по броју грана у графу.

Проналажење чвора са улазним степеном нула може се спровести итерацијом кроз низ улазних степена, међутим, сложеност тог корака је линеарна у односу на број чворова, тако да би укупна сложеност алгоритма била квадратна. Проблем можемо решити и ефикасније тако што током извршавања алгоритма одржавамо списак свих чворова чији је улазни степен 0 – приметимо да таквих чворова у сваком кораку може бити више. Дакле, потребно је чворове са улазним степеном нула чувати у некој колекцији у коју

[4] Аналогно се показује да у усмереном ацикличком графу увек постоји чвор излазног степена нула.

је ефикасно уметати и из које је ефикасно уклањати елементе. У ове сврхе може се користити ред (или стек, што је једнако добро). Према претходној леми у ацикличком графу постоји бар један чвор са улазним степеном нула, а пошто је граф све време извршавања алгоритма ациклички, ред ће у сваком тренутку, до самог краја, бити непразан. Сваки пут када уклонимо и нумеришемо чвор v из реда, уклањамо и све његове гране (v, w) тако што вредност `ulazniStepen[w]` смањујемо за један. Ако вредност улазног степена чвора w при томе постане нула, чвор w уписује се у ред, чиме се постиже да ће се сви чворови степена нула у смањеном графу налазити у реду (за чворове који нису били суседни са v од раније знамо да су у реду ако и само ако им је улазни степен био нула, а тај улазни степен се није променио након уклањања чвора v, док смо за чворове суседене чвору v управо ефективно проверили да ли им је степен у смањеном графу постао нула и ставили смо их у ред ако и само ако јесте). Алгоритам завршава са радом када ред који садржи чворове степена нула постане празан, јер су у том тренутку сви чворови нумерисани. Описани алгоритам зове се *Канов алгоритам*.

Пример 2.4.3

У табели 2.1 илустровано је извршавање Кановог алгоритма на примеру графа са слике 2.26. Претпостављамо да је граф задат листама повезаности, тако да су за сваки чвор његови суседи поређани у лексикографски растућем поретку. За сваки од корака алгоритма приказане су тренутне вредности улазних степена чворова графа, садржај реда који садржи чворове улазног степена нула који још увек нису нумерисани и последњи нумерисани чвор у графу. Приметимо да се у другом кораку могло десити да се у ред најпре дода чвор D, а затим чвор A и у том случају било би добијено другачије тополошко уређење: B, D, A, C, E.

Табела 2.1: Пример извршавања Кановог алгоритма за граф са слике 2.26. Приказани су улазни степени свих чворова, тренутни садржај реда и одређени редни бројеви чворова.

$d(A)$	$d(B)$	$d(C)$	$d(D)$	$d(E)$	*Ред*	*Наредни нумерисани чвор*
1	0	2	1	2	B	
0		2	0	2	A, D	$B:1$
		1		2	D	$A:2$
		0		1	C	$D:3$
				0	E	$C:4$
						$E:5$

Ако након завршетка рада алгоритма за неке чворове важи да нису били додати у ред,

то значи да постоји подскуп скупа чворова такав да у одговарајућем индукованом подграфу сви чворови имају улазни степен већи од нула. Стога индуковани подграф (а тиме и полазни граф) садржи усмерени циклус, супротно претпоставци да је граф ациклички.

```
void topolosko_sortiranje() {
  int brojCvorova = listaSuseda.size();
  // niz koji cuva ulazne stepene cvorova
  vector<int> ulazniStepen(brojCvorova,0);
  // niz koji cuva redne brojeve cvorova u topoloskom uredjenju
  vector<int> topoloskoUredjenje;
  // broj posecenih cvorova
  int brojPosecenih = 0;

  // inicijalizujemo niz ulaznih stepena cvorova
  for (int i = 0; i < listaSuseda.size(); i++)
    for (int j = 0; j < listaSuseda[i].size(); j++)
      ulazniStepen[listaSuseda[i][j]]++;

  // red koji cuva cvorove ulaznog stepena nula
  queue<int> cvoroviStepenaNula;

  // cvorove koji su ulaznog stepena 0 dodajemo u red
  for (int i = 0; i < brojCvorova; i++)
    if (ulazniStepen[i] == 0)
       cvoroviStepenaNula.push(i);

  while(!cvoroviStepenaNula.empty()) {
    // cvor sa pocetka reda numerisemo narednim brojem
    int cvor = cvoroviStepenaNula.front();
    cvoroviStepenaNula.pop();
    topoloskoUredjenje.push_back(cvor);

    brojPosecenih++;

    // za sve susede tog cvora azuriramo ulazne stepene
    for (int i = 0; i < listaSuseda[cvor].size(); i++) {
      int sused = listaSuseda[cvor][i];
      ulazniStepen[sused]--;
```

```
        // ako je ulazni stepen suseda postao 0, dodajemo ga u red
        if (ulazniStepen[sused] == 0)
            cvoroviStepenaNula.push(sused);
      }
   }

   // ako smo numerisali sve cvorove u grafu
   if (brojPosecenih == brojCvorova) {
      // stampamo dobijeno topolosko uredjenje
      cout << "Redosled cvorova u topoloskom uredjenju je:" << endl;
      for (int i = 0; i < brojCvorova; i++)
         cout << topoloskoUredjenje[i] << ": " << i+1 << endl;
   }
   else
      // zakljucujemo da graf sadrzi usmereni ciklus
      cout << "Graf nije aciklicki" << endl;
}
```

Током извршавања Кановог алгоритма смо успутно проверили да ли је дати граф ациклички, тако што смо бројали чворове које смо посетили, тј. који су доспели у ред и проверили да ли је тај број једнак укупном броју чворова. Ако на крају извршавања петље `while` постоји подскуп чворова који су степена већег од 1, онда они припадају неком циклусу, тј. граф садржи циклус.

У случају када је граф задат листама повезаности временска сложеност иницијализације низа `ulazniStepen` је $O(|V| + |E|)$. У петљи `while` (кроз коју се пролази $|V|$ пута) за проналажење чвора са улазним степеном нула потребно је константно време (приступ реду). Свака грана (v, w) разматра се тачно једном, у петљи кроз коју се пролази након уклањања чвора v из реда. Према томе, укупан број промена вредности елемената низа `ulazniStepen` у свим извршавањима спољашње петље `while` једнак је броју грана у графу. Временска сложеност Кановог алгоритма је дакле $O(|V| + |E|)$, односно линеарна је функција од величине графа.

2.4.2 Алгоритам заснован на претрази у дубину

Замислимо на тренутак да је граф задат тако да су му гране окренуте од послова који зависе ка пословима од којих зависе и дефинишимо рекурзивну процедуру која обрађује посао тј. додељује му редни број у тополошком редоследу. Да би посао за који је функција позвана могао да буде урађен, потребно је да се обезбеди да су сви послови од којих он зависи завршени пре њега, што значи да је потребно да се обиђе граф кренувши од тог посла (чвора) и да сеураде сви послови који одговарају његовим суседима

(тј. прескоче ако су већ раније урађени). Када се то уради, тек тада текућем чвору може бити додељен наредни слободан број. Примећујемо, дакле, да овај алгоритам може да врши класичан обилазак графа у дубину, додељујући тражени број чвору у склопу одлазне нумерације.

Докажимо да се на овај начин добија исправан тополошки редослед.

Лема 2.4.2

За сваку грану (u, v) ацикличног графа важи $u.Post > v.Post$.

Доказ. Као што смо раније закључили у графу $G = (V, E)$ важи:

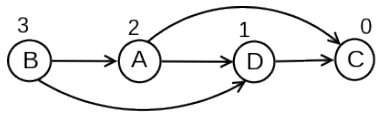
- ако је грана $(u, v) \in E$ грана DFS дрвета, директна или попречна грана, за њу важи $u.Post > v.Post$,
- ако је грана $(u, v) \in E$ повратна грана у односу на DFS дрво, за њу важи $u.Post \leq v.Post$.

Приметимо да циклус у усмереном графу постоји ако и само ако у DFS дрвету постоје повратне гране. У усмереном ацикличком графу не постоји циклус, па не постоје повратне гране у односу на DFS дрво. Дакле, за сваку грану (u, v) ацикличког графа важи услов $u.Post > v.Post$. □

Дакле, ако су гране усмерене од зависног ка независном чвору, одлазна нумерација даје исправан тополошки редослед. Имплементација је сасвим једноставна: у класичном DFS обиласку приликом излазне обраде чвору треба додати редни број у тополошком редоследу тј. додати га на крај низа у ком се чворови чувају у складу са тополошким редоследом. У главној функцији покрећемо DFS обилазак сваког чвора (у произвољном редоследу), прескачући чворове који су већ раније обрађени (слично као код одређивања компонената повезаности).

Вратимо се на стандардну поставку проблема у ком су гране графа оријентисане од независног ка зависном чвору. Ако са $t(x)$ означимо редни број чвора x у тополошком поретку графа G, за сваку грану (u, v) потребно је да важи $t(u) < t(v)$. Пошто за сваку грану (u, v) важи $u.Post > v.Post$, ако чворове графа уредимо у опадајућем редоследу у односу на одлазну нумерацију чворова, добићемо једно тополошко уређење графа. Имплементацију је потребно изменити само тако што се у одлазној обради чворовима додељују бројеви уназад, од n до 1. Ако чворове не нумеришемо, већ их стављамо у низ у складу са тополошким редоследом, они ће у низу бити сложени наопако и низ је након обраде потребно обрнути тј. чворове обрађивати од краја ка почетку тог низа. Алтернативно, чворове можемо постављати на наменски стек и обрађивати их у тополошком редоследу тако што ћемо их један по један скидати са тог стека. Могуће је користити и

неку структуру података која допушта ефикасно додавање елемената на почетак (нпр. листу или ред са два краја).



Слика 2.27: Усмерени ациклички граф: уз сваки чвор приказана је вредност његове одлазне нумерације у односу на DFS претрагу покренуту из чвора B.

Пример 2.4.4

Размотримо граф са слике 2.27: он је усмерен и ациклички. Ако покренемо DFS претрагу из чвора B редослед чворова у којима напуштамо чворове је C, D, A, B. Дакле, тополошко уређење графа добијамо обртањем овог редоследа, односно редослед чворова у тополошком поретку биће B, A, D, C. Приметимо да то одговара и редоследу чворова слева надесно у приказу графа код кога су све гране усмерене слева удесно.

```
void dfs(int cvor, vector<bool> &posecen, vector<int> &odlazna) {
  posecen[cvor] = true;

  // rekurzivno prolazimo kroz sve susede koje nismo obisli
  for (int sused : listaSuseda[cvor]) {
    if (!posecen[sused])
       dfs(sused,posecen,odlazna);
  }

  // u vektor odlazna dodajemo na kraj naredni cvor
  // koji napustamo pri DFS obilasku
  odlazna.push_back(cvor);
}

void topolosko_sortiranje() {
  int brojCvorova = listaSuseda.size();
  vector<bool> posecen(brojCvorova);
  // niz koji sadrzi redom cvorove prema redosledu napustanja
  vector<int> odlazna;

  for (int cvor = 0; cvor < brojCvorova; cvor++)
```

```
    if (!posecen[cvor])
       dfs(cvor,posecen,odlazna);

  // cvorove ispisujemo u opadajućem redosledu odlazne numeracije
  cout << "Redosled cvorova u topoloskom uredjenju je:" << endl;
  for (int i = brojCvorova - 1; i >= 0; i--)
    cout << odlazna[i] << ": " << brojCvorova - i << endl;

}
```

С обзиром на то да се приказани алгоритам своди на DFS претрагу и одређивање одлазне нумерације чворова, његова временска сложеност износи $O(|E| + |V|)$.

Задатак: Организација пројектних активности

Техника процене пројеката (енгл. Project evaulation and review technique, PERT) омогућава одређивање распореда активности тако да се пројекат што пре заврши.

У пројекту се врши више активности и свака активност захтева неко време. Претпоставићемо да је унапред познато колико времена свака активност захтева. Извршавање активности обично захтева да су завршене неке претходне активности. Активности се обично моделују помоћу грана усмереног ацикличког графа. Чворови графа (почеци и крајеви грана) су такозвани догађаји. Да би се могла извршити активност која полази из неког догађаја, потребно је да су претходно завршене све активности које долазе у тај догађај. Написати програм који одређује најмање време потребно за завршетак пројекта и за сваку активност одређује најраније и најкасније време почетка и завршетка, као и максимално допуштено кашњење почетка те активности тако да то не утиче на кашњење целог пројекта.

Опис улаза

Са стандардног улаза се уноси број догађаја $n \leq 10^3$ и број активности $m \leq 10^3$. Након тога се уноси списак активности. За сваку активности је дат редни број догађаја из ког почиње, догађаја у ком се завршава и трајање у данима (цео број од 1 до 100).

Опис излаза

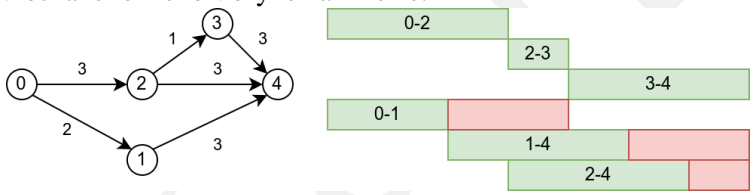
У првом реду стандардног излаза исписати најмањи број дана потребан за завршетак пројекта. У наредних m редова за сваку активност (у редоследу у ком су учитане) исписати најраније време почетка, најраније време завршетка, најкасније време почетка, најкасније време завршетка и допуштено кашњење (претпоставља се да почетни догађај креће у дану 0).

Пример

Улаз	*Излаз*
5	7
6	0 3 0 3 0
0 2 3	0 2 2 4 2
0 1 2	3 4 3 4 0
2 3 1	3 6 4 7 1
2 4 3	2 5 4 7 2
1 4 3	4 7 4 7 0
3 4 3	

Објашњење

Активности $(0, 2)$, $(2, 3)$ и $(3, 4)$ су тзв. критичне активности и њихово кашњење би проузроковало кашњење целог пројекта. Са друге стране, на пример, активност $(0, 1)$ слободно може да закасни 2 дана, јер чак иако догађај 1 буде 4 дана након почетка пројекта, активност $(1, 4)$ траје 3 дана и пројекат ће и даље бити завршен у року од 7 дана. Активност $(2, 4)$ може да касни 1 дан, јер иако крене 4 дана након почетка пројекта, пошто траје 3 дана, завршиће се на време. Граф догађаја и активности и распоред активности су приказани на слици. За сваку активност је зеленом бојом означен њен најранији могући почетак, док је црвеном бојом означено њено могуће кашњење.



Решење

У првој фази треба за сваки догађај одредити најраније време да се он деси тј. најраније време потребно да се заврше све активности које воде до њега (енгл. earliest start time, est) и најкасније време да се он деси тј. најкасније време да се започну све активности које воде из њега, тако да не дође до кашњења пројекта (енгл. latest start time, lst). Сва времена ћемо изражавати у данима (који се броје од нуле).

Најраније време почетног догађаја је нула. Најраније време каснијих догађаја се може одредити анализом свих активности које воде до њега тј. анализом свих претходника чвора. Најраније време завршетка сваке такве активности је одређеном збиром њеног најранијег почетка (тј. најранијег времена догађаја у ком та активност почиње) и њеног трајања. Догађај не може да почне док се не заврши последња активност која води до њега, тако да је потребно пронаћи максимум описаних збирова. Важи, дакле, следећа рекурентна формула (са est_i обележено је најмање време почетка догађаја i, а са w_{ji} трајање активности између догађаја j и i):

$$est_0 = 0, \qquad est_i = \max\{est_j + w_{ji}, j \in Prethodnici(i)\}.$$

Рекурзивна имплементација ове везе би била јако неефикасна услед поновљених рекурзивних позива, па је пожељно применити динамичко програмирање. Динамичко програмирање на усмереном ацикличком графу реализујемо тако што чворове сортирамо у

тополошком редоследу и затим израчунавамо статистику сваког чвора у том редоследу (јер смо сигурни да је приликом израчунавање статистике било ког чвора већ исправно израчунате статистике његових претходника).

Да пројекат не би каснио, последњи догађај мора да се деси што је раније могуће, па је стога најкасније време последњег догађаја једнако његовом најранијем времену. Најкасније време почетка било ког другог догађаја одређено је његовим следбеницима тј. активностима које од њега почињу. Да би свака таква активности могла да се заврши на време, потребно је да почне у дану чији је број једнак разлици најкаснијег времена њеног завршног догађаја и њеног трајања. Пошто све активности треба да буду завршене на време, одређујемо минимум ових разлика. Ако са lst_i обележимо најкасније време догађаја и са c обележимо последњи чвор у тополошком редоследу добијамо следећу рекурентну везу:

$$lst_c = est_c, \qquad lst_i = \min\{lst_j - w_{ij}, j \in Sledbenici(i)\}.$$

Пошто вредност статистика чворова зависи од њихових следбеника, ову статистику можемо израчунавати динамичким програмирањем, обиласком чворова у обратном тополошком редоследу.

Најраније време почетка активности одређено је најранијим временом догађаја којим почиње, док је њено најкасније време почетка одређено разликом између најкаснијег времена догађаја у ком се завршава и њеног трајања. Времена завршетка се добијају увећавањем ових времена за трајање активности. Допуштено кашњење активности је одређено разликом између њеног најкаснијег и најранијег времена почетка.

```
// topoloski sortiramo graf
vector<int> topoloskiRedosled = topoloskoSortiranje(sledbenici);

// za svaki dogadjaj odredjujemo najmanje vreme da se on desi tj.
// najmanje vreme potrebno da se zavrse sve aktivnosti koje dovode do njega
vector<int> najranijeVremeCvora(brojCvorova, 0);
najranijeVremeCvora[topoloskiRedosled[0]] = 0;
for (int i = 1; i < topoloskiRedosled.size(); i++) {
  int cvor = topoloskiRedosled[i];
  for (const auto& [prethodnik, tezina] : prethodnici[cvor])
    najranijeVremeCvora[cvor] =
      max(najranijeVremeCvora[cvor],
          najranijeVremeCvora[prethodnik] + tezina);
}
```

```
// za svaki dogadjaj odredjujemo najkasnije vreme da se on desi tj.
// najkasnije vreme potrebno da pocnu sve aktivnosti koje vode iz njega
const int infty = numeric_limits<int>::max();
vector<int> najkasnijeVremeCvora(brojCvorova, infty);
najkasnijeVremeCvora[topoloskiRedosled.back()] =
  najranijeVremeCvora[topoloskiRedosled.back()];
for (int i = topoloskiRedosled.size() - 2; i >= 0; i--) {
  int cvor = topoloskiRedosled[i];
  for (const auto& [sledbenik, tezina] : sledbenici[cvor])
    najkasnijeVremeCvora[cvor] =
      min(najkasnijeVremeCvora[cvor],
          najkasnijeVremeCvora[sledbenik] - tezina);
}

// trajanje projekta je najmanje vreme poslednjeg dogajaja
cout << najranijeVremeCvora[topoloskiRedosled.back()] << endl;

// analiziramo sve grane i odrejujemo trazena vremena i kasnjenja
for (const auto& [cvorOd, cvorDo, tezina] : grane) {
  cout << najranijeVremeCvora[cvorOd] << " "
       << najranijeVremeCvora[cvorOd] + tezina << " "
       << najkasnijeVremeCvora[cvorDo] - tezina << " "
       << najkasnijeVremeCvora[cvorDo] << " "
       << najkasnijeVremeCvora[cvorDo] - tezina - najranijeVremeCvora[cvorOd]
       << endl;
}
```

2.5 Мостови и артикулационе тачке у неусмереном графу

У овом поглављу бавићемо се питањем колико је дати неусмерени граф добро повезан. Другим речима интересује нас да ли у графу постоји слаба тачка, односно уско грло, чијим би уклањањем граф престао да буде повезан. У разним практичним применама овај проблем је веома значајан и потребно је омогућити да се уска грла у графу брзо детектују. Слична питања се могу поставити и за усмерене графове, међутим, ми ћемо се у овом поглављу ограничити на случај неусмерених графова.

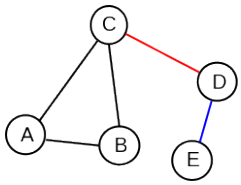
Грану неусмереног графа $G = (V, E)$ чијим се уклањањем из графа број компоненти повезаности графа повећава називамо *мост* (енгл. bridge, cut edge). Специјално, пове-

зани граф након уклањања моста престаје да буде повезан.

У рачунарским мрежама проналажење мостова може помоћи у идентификовању критичних веза чији квар може довести до распада мреже. У транспортним системима мостови представљају кључне руте које повезују различите области, а чијим би уклањањем транспорт био онемогућен.

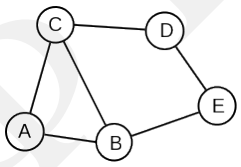
Пример 2.5.1

Уколико у графу приказаном на слици 2.28 уклонимо грану (C, D) или грану (D, E) граф престаје да буде повезан, те ове две гране, свака за себе, чине мост у датом графу.



Слика 2.28: Пример графа који садржи два моста: један је означен црвеном, а други плавом бојом.

Постоје графови у којима нема мостова (слика 2.29).



Слика 2.29: Пример графа који не садржи ни мост ни артикулациону тачку.

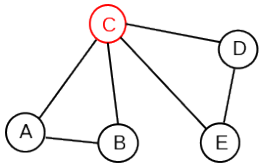
Уколико у неусмереном графу $G = (V, E)$ постоји чвор $v \in V$ такав да се његовим уклањањем из графа (заједно са гранама које су му суседне) број компоненти повезаности графа повећава, онда такав чвор називамо *артикулационом тачком* (енгл. articulation point, cut vertex). Специјално, повезани граф након уклањања артикулационе тачке престаје да буде повезан.

У транспортним системима артикулационе тачке указују на критичне раскрснице чија би евентуална недоступност (нпр. у случају неке несреће) имала значајан утицај на ток саобраћаја. На овај начин могу се планирати приоритетне инвестиције у саобраћајну инфраструктуру. Слично, артикулационе тачке у електричним колима представљају компоненте чијим би отказивањем рада дошло до губитка везе између осталих

компоненти. Идентификовање ових компоненти може побољшати поузданост дизајна електричног кола. У контексту друштвених мрежа артикулационе тачке представљају особе које повезују две или више различитих заједница људи и чије би уклањање разбило мрежу на неповезане групе.

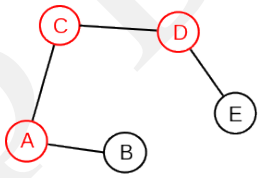
Пример 2.5.2

Уколико у повезаном графу приказаном на слици 2.30 уклонимо чвор C, граф престаје да буде повезан, те је чвор C једна артикулациона тачка овог графа. Штавише, непосредно се проверава да је чвор C једина артикулациона тачка у овом графу.



Слика 2.30: Граф који садржи једну артикулациону тачку: чвор C.

Граф може да не садржи артикулационе тачке (слика 2.29), а може и да садржи већи број артикулационих тачака (слика 2.31).



Слика 2.31: Граф који садржи већи број артикулационих тачака: то су чворови A, C и D. Свака грана овог графа је мост.

У наставку ћемо размотрити проблеме проналажења свих мостова и артикулационих тачака у датом неусмереном графу. Без смањења општости може се претпоставити да је дати граф повезан. Ако граф није повезан, онда се мостови и артикулационе тачке могу тражити независно у свакој компоненти повезаности графа.

2.5.1 Одређивање мостова

Позабавимо се најпре проблемом одређивања мостова, за који се показује да је донекле једноставнији.

Проблем

Дефинисати алгоритам који у датом неусмереном повезаном графу одређује све мостове.

Директан начин да се у датом неусмереном повезаном графу $G = (V, E)$ пронађу сви мостови подразумева да за сваку грану $e \in E$ графа G проверимо да ли је граф без гране e повезан (нпр. коришћењем алгоритма DFS). Сложеност овог алгоритма износи $O(|E| \cdot (|V| + |E|))$.[5]

Постоји ефикаснији алгоритам за одређивање мостова у графу. Ми ћемо у наставку размотрити алгоритам који је осмислио Роберт Тарџан[6] и који је линеарне временске сложености у функцији величине графа.

2.5.1.1 Тарџанов алгоритам за одређивање мостова

Мостови не припадају циклусима. Заиста, важи наредно тривијално тврђење (које наводимо без доказа).

Лема 2.5.1 **[Веза мостова и циклуса]**

Грана (u, v) је мост у графу G ако и само ако не припада ниједном циклусу у G.

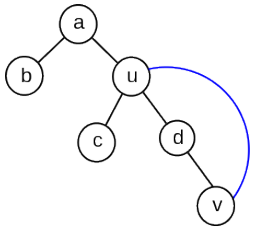
Специјално, пошто у дрвету не постоје циклуси, ако је граф дрво, онда је свака грана у том графу мост (на пример, такав је граф на слици 2.31).

Размотримо DFS дрво добијено DFS обиласком датог графа $G = (V, E)$. С обзиром на то да је полазни граф неусмерен, постоје две врсте грана графа у односу на DFS дрво: гране DFS дрвета и гране које повезују потомка са претком у односу на DFS дрво (у наставку ћемо их, једноставности ради, звати повратне гране, мада оне нису усмерене, па се могу истовремено сматрати и за директне и за повратне гране). Ако је грана (u, v) повратна, она не може бити мост у графу, јер је део циклуса који та грана чини са гранама DFS дрвета (видети пример на слици 2.32).

Дакле, мостови могу бити само гране DFS дрвета, те је довољно да алгоритам разматра само њих као кандидате, чиме се може значајно смањити број грана које је потребно проверавати. Међутим, ни ове гране не морамо проверавати техником грубе силе. Нека је (u, v) грана DFS дрвета. Претпоставимо да је DFS претрага најпре посетила чвор u па чвор v, тј. да је чвор u родитељ чвора v у DFS дрвету графа.

[5] Приметимо да је уклањање гране из графа у случају када је граф задат листама повезаности неефикасно, али пошто се након уклањања сваке гране извршава алгоритам претраге у дубину, ова операција не утиче на сложеност комплетног алгоритма.

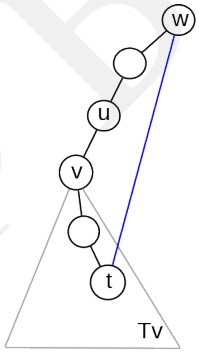
[6] Роберт Тарџан (енгл. Robert Tarjan), амерички информатичар рођен 1948. године.



Слика 2.32: Грана (u, v) која повезује потомка и претка у DFS дрвету не може бити мост.

За грану (u, v) DFS дрвета важи да је мост ако и само ако не постоји ни једна друга грана између неког претка чвора u (укључујући и чвор u) и неког потомка чвора v (укључујући и чвор v).

Дакле, грана (u, v) је мост ако и само ако поддрво DFS дрвета чији је корен чвор v (које је „изнад”[7] чвора v) остаје неповезано са делом графа „испод” ове гране тј. чвора u. Другим речима тада не постоји начин да се стигне из поддрвета T_v чији је корен чвор v до чвора u или неког претка чвора u.



Слика 2.33: Илустрација дефиниције вредности $L(v)$.

Потребно је за сваки чвор $v \in V$ израчунати колико се „ниско” можемо вратити неком повратном граном из произвољног чвора поддрвета T_v чији је корен чвор v. То можемо квантификовати на основу долазних DFS бројева чворова до којих воде повратне гране

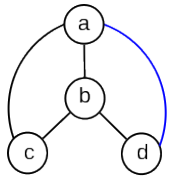
[7] Пошто се дрво у овом тексту црта наопако, тако да му је корен на врху цртежа, а листови на дну, део графа „изнад” неког чвора је нацртан испод тог чвора. У наставку ће бити коришћена терминологија у којој су термини изнад/испод, нижи/вижи, горе/доле дефинисани у складу са положајем корена и листова, а не у складу са наопаким цртањем дрвета. Ово је у складу и са тим да је висина корена 0 и да висина чворова расте како се удаљавамо од корена. Да бисмо смањили могућност забуне, ове одреднице ће бити писане под наводницима.

из T_v („нижи" чворови тј. чворови ближи корену имају и ниже DFS бројеве). Као што је раније описано, за сваки чвор $v \in V$ може се одредити вредност $v.Pre$ која означава редни број чвора v при долазној DFS нумерацији – ту вредност ћемо у наставку краће звати *редним бројем чвора* v. Циљ нам је да за сваки чвор v одредимо и редни број *„најнижег" претка достижног повратном граном* (енгл. lowlink) до ког се можемо попети неком (највише једном) повратном граном из поддрвета T_v (слика 2.33). Нагласимо да ово не мора бити „најнижи" достижни предак, јер се уз коришћење више повратних грана можда може стићи до неког „нижег" претка тј. претка који има мањи DFS број и ближе је корену дрвета.

Дефинишимо сада прецизно појам „најнижег" претка достижног повратном граном[8]. Означимо са $L(v)$ (енгл. lowlink) мању од вредности редног броја чвора v и најмање међу вредностима редних бројева чворова до којих се може стићи повратном граном из произвољног чвора поддрвета T_v (које укључује и чвор v), која није грана DFS дрвета која води од v ка његовом родитељском чвору у DFS дрвету. Могуће је да из поддрвета T_v не постоји ниједна грана ка претку чвора v и тада је вредност $L(v)$ једнака редном броју чвора v. Дакле важи следеће:

$$L(v) = \min\{v.Pre, \min_{\substack{w \text{ је предак } v \\ \text{постоји грана } (t,w),\ t \in T_v, \\ \text{која не спаја } v \text{ са његовим родитељем}}} w.Pre\}.$$

Вредност $L(v)$ је дефинисана као редни број чвора, међутим, пошто постоји јасна бијекција између чворова и њихових редних бројева можемо слободно рећи да је „најнижи" предак чвора v достижан повратном граном онај чвор чији је редни број $L(v)$. Путања од чвора v до његовог „најнижег" претка достижног повратном граном се увек састоји од низа грана DFS дрвета за којима следи највише једна повратна грана. „Најнижи" предак достижан повратном граном је увек јединствено одређен, али путања до њега не мора бити (пример је дат на слици 2.34).

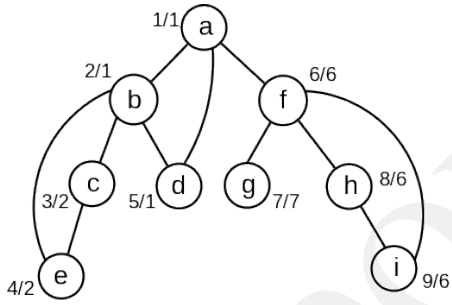


Слика 2.34: Најнижи достижни чвор повратном граном чвора b је чвор a. До њега се може стићи путевима (b, c, a) и (b, d, a).

[8]Нагласимо да се претпоставља да се ради о неусмереном графу. У усмереним графовима је могуће дефинисати сродан појам, али је дефиниција компликованија и биће изложена у поглављу 2.6.

Пример 2.5.3

На слици 2.35 дат је пример графа где је уз сваки чвор v приказан његов редни број $v.Pre$ и вредност $L(v)$. На пример, важи $L(c) = 2$ и $L(e) = 2$ јер из чвора e граном (e, b) можемо стићи до чвора b чија је редни број једнак 2. Слично, важи $L(b) = 1$ јер из чвора d који је потомак чвора b можемо стићи граном (d, a) до чвора a чији је редни број 1.



Слика 2.35: Пример графа и одговарајућег DFS дрвета. Уз сваки чвор v приказан је његов редни број и вредност $L(v)$.

Сада можемо дати карактеризацију мостова помоћу функције L. Поддрво T_v DFS дрвета чији је корен у чвору v остаће након избацивања гране (u, v) неповезано са делом графа „испод" ове гране ако и само ако важи $L(v) > u.Pre$. Дакле, важи наредна теорема.

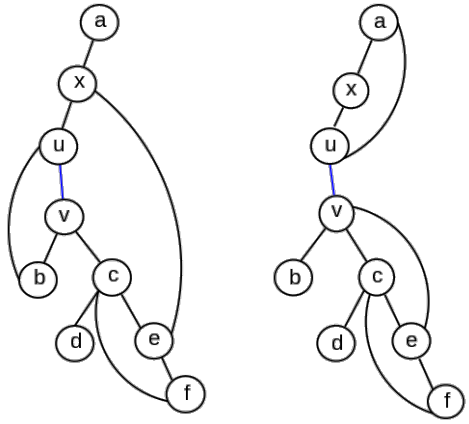
Теорема 2.5.1 **[Карактеризација моста у графу помоћу функције L]**

Грана (u, v) је мост у графу G ако и само ако важи $L(v) > u.Pre$.

Пример 2.5.4

Размотримо граф приказан на слици 2.36, лево.

Грана (u, v) није мост у графу јер се из поддрвета T_v можемо вратити у део DFS дрвета „испод" ове гране. Прецизније, из чвора b можемо се вратити у чвор u, а из чвора e у чвор x. Пошто је чвор x „испод" чвора u, важи $x.Pre < u.Pre$ и вредност $L(v)$ једнака је редном броју $x.Pre$ чвора x. Стога није задовољен услов $L(v) > u.Pre$, па грана (u, v) није мост. Чак и кад граф не би садржао грану (x, e), након избацивања гране (u, v) могли бисмо се граном (b, u) из поддрвета T_v вратити до чвора u, а тиме и до произвољног чвора „испод" њега у DFS дрвету, те и у том случају грана (u, v)



Слика 2.36: Лево: пример графа у коме грана (u, v) није мост, десно: пример графа у коме грана (u, v) јесте мост.

не би била мост у графу (важило би да је $L(v) = u.Pre$*, па ни тада не би важило* $L(v) > u.Pre$*).*

Размотримо сада граф са слике 2.36, десно. Након избацивања гране (u, v) *из графа, из поддрвета* T_v *можемо се вратити „најниже" до чвора* v*, тј. важи услов* $L(v) = v.Pre > u.Pre$ *и у овом графу грана* (u, v) *јесте мост.*

Приметимо да, такође, важи да је грана (u, v) мост ако и само ако је $L(v) = v$.

Остаје питање како ефикасно израчунати вредности $L(v)$ за све чворове v у графу. Важи следећа лема.

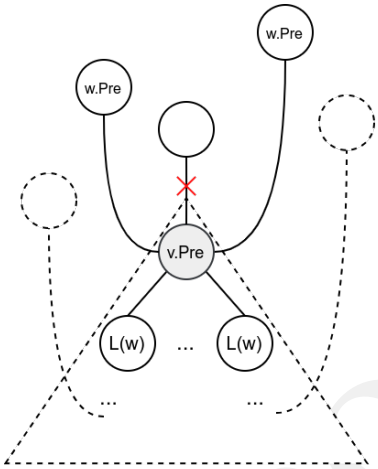
Лема 2.5.2 **[Рекурзивна веза за L]**

За сваки чвор $v \in V$ *неусмереног графа* $G = (V, E)$ *важи:*

$$L(v) = \min\{v.Pre, \min_{\substack{(v,w)\in E \\ w \text{ је предак чвора } v \\ w \text{ није родитељ чвора } v}} w.Pre, \min_{w \text{ је дете од } v} L(w)\} \tag{2.1}$$

Доказ. Разматрајмо произвољни чвор v и путању од њега до „најнижег" достижног чвора повратном граном. Та путања је или празна или се састоји од тачно једне повратне гране или се састоји од неколико грана DFS дрвета за којима следи тачно једна

повратна грана. У једнакости (2.1) се анализирају све могуће такве путање и тражи се она најбоља.



Слика 2.37: Илустрација једнакости (2.1).

Заиста, празна путања је обухваћена изразом $v.Pre$, а путање које се састоје тачно од једне повратне гране изразом

$$\min_{\substack{(v,w)\in E \\ w \text{ је предак од } v \\ w \text{ није родитељ од } v}} w.Pre.$$

Ако путања садржи гране DFS дрвета, након преласка прве гране те путање стиже се до неког чвора w (који није родитељ чвора v), а затим се наставља путањом од чвора w до „најнижег" достижног чвора повратном граном кренувши од чвора w. Заиста, до сваког чвора достижног повратном граном из w, може се стићи путањом и из чвора v која се завршава повратном граном, а ако би се од чвора w могло доћи до неког „нижег" чвора путањом са повратном граном, до њега би се могло стићи путањом са повратном граном и из v (слика 2.37). Путање од v које садрже бар једну грану дрвета се, дакле, анализирају изразом:

$$\min_{w \text{ је дете од } v} L(w). \qquad \square$$

На основу овога, вредности функције L се могу одредити током DFS обиласка графа. Дефинишемо рекурзивну функцију која врши DFS обилазак такву да ће након завршетка њеног рекурзивног позива за произвољни чвор v графа бити одређене вредности

$v.Pre$ и $L(v)$. Приликом улазне обраде чвора v додељује му се улазни број $v.Pre$ и вредност $L(v)$ се иницијализује на ту вредност. Након тога се обрађују све гране (v, w). Приликом обраде гране (v, w), ради се следеће:

- Ако чвор w није посећен, врши се рекурзивно његов обилазак и грана (v, w) постаје грана DFS дрвета. Након рекурзивног позива (у ком се врши обрада комплетног поддрвета чији је корен w), израчуната је вредност $L(w)$, па вредност $L(v)$ ажурирамо на вредност $\min\{L(v), L(w)\}$.
- Ако је чвор w раније посећен и није родитељ чвора v, ажурирамо вредност $L(v)$ на вредност $\min\{L(v), w.Pre\}$ (овим се обрађује свака грана ка претку w чвора v, а ако је w посећени потомак чвора v, важи $w.Pre > v.Pre \geq L(v)$, па до ажурирања неће доћи).

Индукцијом, уз коришћење леме 2.5.2, лако се може доказати коректност описаног алгоритма.

Теорема 2.5.2 **[Коректност алгоритма израчунавања функције L]**

Претходним алгоритмом се исправно израчунавају вредности $L(v)$ за све чворове v.

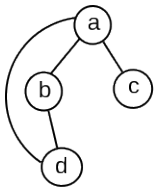
Имплементација у наставку одређује долазне редне бројеве свих чворова, редне бројеве њихових „најнижих" предака достижних повратном граном тј. функције L и све мостове у једном DFS обиласку.

```
int vreme_dolazna = 0;
vector<bool> posecen;
vector<int> dolazna;
vector<int> lowlink;
vector<int> roditelj;
// niz grana koje su mostovi u grafu
vector<pair<int,int>> mostovi;

void dfs_mostovi(int cvor) {
  // dodeljujemo redni broj cvoru 'cvor'
  posecen[cvor] = true;
  dolazna[cvor] = vreme_dolazna++;
  // inicijalizujemo vrednost funkcije L cvora 'cvor' na njegov redni broj
  lowlink[cvor] = dolazna[cvor];

  // rekurzivno prolazimo kroz sve susede koje nismo obisli
  for (int sused : listaSuseda[cvor]) {
```

```
    // ukoliko je sused vec posecen
    if (posecen[sused]) {
      // ako grana ne vodi ka roditelju datog cvora
      if (sused != roditelj[cvor])
         // po potrebi azuriramo vrednost L cvora na redni broj suseda
         if (dolazna[sused] < lowlink[cvor])
             lowlink[cvor] = dolazna[sused];
    } else {
    // ukoliko sused nije posecen

      // pamtimo granu DFS drveta
      roditelj[sused] = cvor;
      // pokrecemo pretragu iz cvora 'sused'
      dfs_mostovi(sused);

      // nakon obrade poddrveta čiji je koren cvor 'sused'
      // vrednost L cvora 'sused' je odredjena;
      // po potrebi azuriramo vrednost L za cvor 'cvor'
      if (lowlink[sused] < lowlink[cvor])
         lowlink[cvor] = lowlink[sused];

      // proveravamo da li je grana (cvor, sused) most
      // u ovom trenutku su vrednosti L cvora 'sused' i
      // rednog broja cvora 'cvor' odredjene
      if (lowlink[sused] > dolazna[cvor])
         mostovi.emplace_back(cvor, sused);
    }
  }
}

// svi mostovi u grafu se smestaju u globalni niz mostovi
void odrediMostove(int cvor) {
  int brojCvorova = listaSuseda.size();
  posecen.resize(brojCvorova, false);
  dolazna.resize(brojCvorova);
  lowlink.resize(brojCvorova);
  roditelj.resize(brojCvorova, -1);
  dfs_mostovi(cvor);
}
```

Пример 2.5.5

У наставку је илустровано извршавање описаног алгоритма на примеру једноставног неусмереног графа са слике 2.38. Претпоставља се да је граф задат листама повезаности тако да су суседи сваког чвора уређени лексикографски растуће и да DFS претрага започиње из чвора a.



Слика 2.38: Пример графа који садржи један мост: грану (a, c).

- *Покрећемо DFS из чвора a, постављамо $a.Pre = 1$ и $L(a) = 1$*
  - *Разматрамо суседа b чвора a.*
    Покрећемо DFS из чвора b, постављамо $b.Pre = 2$ и $L(b) = 2$
    - *Разматрамо суседа a чвора b.*
      То је грана ка родитељу коју даље не обрађујемо.
    - *Разматрамо суседа d чвора b.*
      Покрећемо DFS из чвора d, постављамо $d.Pre = 3$ и $L(d) = 3$
      - *Разматрамо суседа a чвора d*
        Грана (d, a) је грана од потомка ка претку, па постављамо $L(d) = a.Pre$, и добијамо $L(d) = 1$.
      - *Разматрамо суседа b чвора d.*
        То је грана ка родитељу, коју даље не обрађујемо.
    - *Враћамо се у чвор b.*
      Пошто је $L(d) < L(b)$ постављамо $L(b) = L(d)$, па важи и $L(b) = 1$
      Пошто је $L(d) < b.Pre$, грана (b, d) није мост.
  - *Враћамо се у чвор a.*
    Пошто важи $L(b) = L(a)$, не радимо ништа.
    Пошто је $L(b) = a.Pre$, грана (a, b) није мост.
  - *Разматрамо суседа c чвора a.*
    Покрећемо DFS из чвора c, постављамо $c.Pre = 4$ и $L(c) = 4$
    - *Разматрамо суседа a чвора c*
      То је грана ка родитељу коју даље не обрађујемо.
  - *Враћамо се у чвор a.*
    Пошто важи $L(c) > L(a)$, не радимо ништа.
    Пошто је $L(c) > a.Pre$, грана (a, c) јесте мост.

– *Разматрамо суседа d чвора a.*
Пошто важи $L(d) = L(a)$, не радимо ништа.

Тарџанов алгоритам за одређивање свих мостова у графу се заснива на DFS претрази, са одговарајућом долазном и одлазном обрадом која је сложености $O(1)$, па је временска сложеност овог алгоритма $O(|V| + |E|)$.

2.5.2 Одређивање артикулационих тачака

Позабавимо се сада проблемом одређивања артикулационих тачака.

Проблем

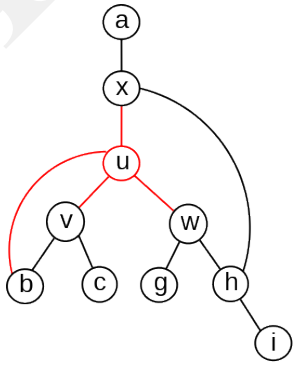
Дефинисати алгоритам који у датом неусмереном повезаном графу одређује све артикулационе тачке.

Артикулационе тачке у датом неусмереном повезаном графу можемо да одредимо тако што за сваки чвор $v \in V$ графа G извршимо проверу да ли је граф без чвора v повезан. Сложеност овог директног алгоритма је $O(|V| \cdot (|V| + |E|))$. Уместо њега, размотрићемо Тарџанов алгоритам, који је доста ефикаснији.

2.5.2.1 Тарџанов алгоритам за одређивање артикулационих тачака

Веома налик Тарџановом алгоритму за одређивање мостова је и алгоритам за тражење артикулационих тачака у графу. Покушајмо да формулишемо критеријум да је чвор артикулациона тачка кроз неколико примера.

Пример 2.5.6



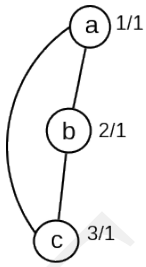
Слика 2.39: Пример графа у коме је чвор u артикулациона тачка.

Размотримо граф са слике 2.39. Чвор u има два детета у DFS дрвету: v и w. Након избацивања чвора u и њему суседних грана из графа, из поддрвета T_w можемо се вра-

тиши у део графа „испод" чвора u (јер је $L(w) = x.Pre$), међутим, из поддрвета T_v можемо се вратити „најниже" до чвора u (јер важи $L(v) = u.Pre$). С обзиром на то да чвор u има дете v тако да ниједан чвор из поддрвета T_v није повезан са неким претком чвора u, чвор u јесте артикулациона тачка у графу.

Пример 2.5.7

Приметимо да за корен a DFS дрвета графа са слике 2.40 и његово дете b важи услов $L(b) = a.Pre$, тј. чвор a има дете из којег се не можемо повратном граном вратити „ниже" од чвора a, а притом чвор a није артикулациона тачка.

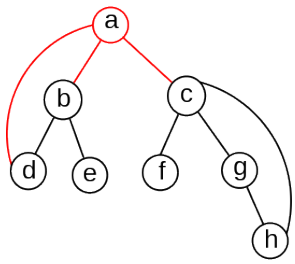


Слика 2.40: Илустрација графа у коме за корен DFS дрвета a и његово дете b важи услов $L(b) \geq a.Pre$, а корен a DFS дрвета није артикулациона тачка.

Разлог је то што је a корен DFS дрвета, па не постоји део DFS дрвета „испод" њега. Дакле, случај чвора који је корен DFS дрвета мора се засебно разматрати.

Пример 2.5.8

Размотримо граф са слике 2.41.



Слика 2.41: Пример графа у коме је чвор a као корен DFS дрвета артикулациона тачка.

Чвор a као корен DFS дрвета има два детета и након његовог избацивања граф постаје неповезан. Дакле, чвор a је артикулациона тачка у графу.

Карактеризација артикулационих тачака дата је наредном лемом.

Лема 2.5.3 **[Карактеризација артикулационих тачака]**

Чвор u је артикулациона тачка графа ако и само ако је испуњен један од наредна два услова:

- *u није корен DFS дрвета и има дете v у DFS дрвету такво да ниједан чвор у поддрвету T_v није повезан са неким претком чвора u;*
- *u је корен DFS дрвета и има бар два детета.*

Први услов одговара ситуацији када након избацивања чвора u из графа више није могуће доћи из поддрвета T_v чији је корен v дете чвора u, до неког претка чвора u. Ако је задовољен други услов, с обзиром на то да у неусмереним графовима не постоје попречне гране, избацивање корена DFS дрвета довело би до „разбијања" графа на већи број компоненти повезаности (по једну за свако дете корена DFS дрвета).

Из примера је јасно да се, као и у случају мостова, за испитивање повезаности елемената поддрвета са прецима чвора могу користити вредности $L(v)$, чиме се добија наредна теорема.

Теорема 2.5.3 **[Карактеризација артикулационих тачака помоћу вредности L]**

Чвор u је артикулациона тачка у графу ако и само ако важи један од наредна два услова:

- *u није корен DFS дрвета и за неко дете v чвора u важи услов $L(v) \geq u.Pre$;*
- *u је корен DFS дрвета и има бар два детета.*

У наставку је дата имплементација Тарјановог алгоритма којим се у датом графу одређују све артикулационе тачке.

```
int vreme_dolazna = 0;
vector<bool> posecen;
vector<int> dolazna;
vector<int> lowlink;
vector<int> roditelj;
vector<bool> artikulacioneTacke;

void dfs_artikulacione(int cvor) {
  // dodeljujemo redni broj cvoru 'cvor'
  posecen[cvor] = true;
  dolazna[cvor] = vreme_dolazna ++;
```

```
// broj dece cvora cvor
int broj_dece = 0;

// vrednost funkcije L za cvor 'cvor' inicijalizujemo na njegov redni broj
lowlink[cvor] = dolazna[cvor];

// rekurzivno prolazimo kroz sve susede koje nismo obisli
for (int sused : listaSuseda[cvor]) {
  // ako je grana (cvor, sused) povratna i ne vodi ka roditelju cvora 'cvor'
  if (posecen[sused]) {
     if (sused != roditelj[cvor])
        // po potrebi azuriramo vrednost L za cvor 'cvor'
        if (dolazna[sused] < lowlink[cvor])
           lowlink[cvor] = dolazna[sused];
  } else {
    // 'sused' je novo dete cvora 'cvor' u DFS drvetu
    broj_dece++;
    // dodajemo granu u DFS drvo
    roditelj[sused] = cvor;

    // pokrecemo pretragu iz cvora 'sused'
    dfs_artikulacione(sused);

    // nakon obrade poddrveta čiji je koren cvor 'sused'
    // vrednost funkcije L cvora 'sused' je odredjena,
    // pa po potrebi azuriramo vrednost L cvora 'cvor'
    if (lowlink[sused] < lowlink[cvor])
       lowlink[cvor] = lowlink[sused];

    // ako 'cvor' nije koren DFS drveta i ako cvor nijedan cvor u poddrvetu
    // cvora 'sused' nije povezan sa nekim pretkom cvora 'cvor',
    // onda je 'cvor' artikulaciona tacka
    if (roditelj[cvor] != -1 && lowlink[sused] >= dolazna[cvor])
       artikulacioneTacke[cvor] = true;
  }
}
// obradjena su sva deca cvora 'cvor'
// proveravamo da li je cvor 'cvor' koren DFS drveta i
```

```
  // da li ima vise od jednog deteta
  if (roditelj[cvor] == -1 && broj_dece > 1)
    // ako je uslov ispunjen, cvor je artikulaciona tacka
    artikulacioneTacke[cvor] = true;
}

void ispisi_artikulacione_tacke(int cvor) {
  int brojCvorova = listaSuseda.size();
  posecen.resize(brojCvorova, false);
  dolazna.resize(brojCvorova);
  lowlink.resize(brojCvorova);
  roditelj.resize(brojCvorova, -1);
  // inicijalno nijedan cvor nije artikulaciona tacka
  artikulacioneTacke.resize(brojCvorova, false);

  dfs_artikulacione(cvor);

  cout << "Artikulacione tacke u grafu su: ";
  for (int i = 0; i < artikulacioneTacke.size(); i++) {
    if (artikulacioneTacke[i])
      cout << i << " ";
 }
 cout << endl;
}
```

Задатак: Усмеравање путева

У једном граду су улице уске и ствара се гужва у саобраћају. Градске власти су одлучиле да све улице постану једносмерне, у нади да ће се тиме повећати проточност, међутим, нису сигурни да ли је могуће да усмере путеве тако да се и даље може стићи од било које, до било које друге тачке у граду.

Опис улаза

Са стандардног улаза се учитава број тачака у граду n ($1 \leq n \leq 10^5$), а затим број двосмерних путева између њих m ($1 \leq m \leq 10^5$). У наредних m линија налазе се по два различита броја a_i и b_i ($1 \leq a_i, b_i \leq n$, $a_i \neq b_i$), који представљају редне бројеве тачака спојених улицом.

Опис излаза

На стандардни излаз исписати 0 ако није могуће усмерити улице или m парова тачака

који представљају усмерене улице.

Пример 1

Улаз	*Излаз*
3	1 2
3	2 3
1 2	3 1
2 3	
1 3	

Пример 2

Улаз	*Излаз*
4	0
4	
1 2	
2 3	
3 1	
1 4	

Решење

Проблем се сасвим директно моделује неусмереним графом у ком су чворови тачке у граду, а гране двосмерни путеви између њих. По претпоставкама задатка полазни граф је повезан.

Кључни увид за решење задатка је да је усмеравање путева могуће ако и само ако у графу не постоји мост. Наиме, ако у графу постоји мост, како год да га усмеримо, неће постојати други пут између дела дрвета изнад и испод тог моста. Ако у графу не постоји мост, тада је могуће усмерити све гране дрвета наниже, а све повратне гране навише, чиме би се добила веза између свих чворова графа. Наиме, пошто је полазни граф повезан, од корена је могуће кроз дрво стићи до било ког чвора дрвета (тј. графа). Од сваког чвора је могуће повратном граном вратити се у неки део дрвета „испод" тог чвора, па се крећући се на тај начин може стићи и до корена. Дакле, било који чвор и корен су обострано достижни, па су и сви чворови међусобно обострано достижни.

Пошто се током одређивања мостова памте родитељи свих чворова, гране дрвета можемо, на пример, препознати коришћењем низа родитеља.

```
// određujemo sve mostove grafa
dfs_mostovi(0);

if (mostovi.size() > 0)
  // ako u grafu ima mostova, ulice se ne mogu usmeriti
  cout << 0 << endl;
else {
  // u grafu nema mostova, pa usmeravamo grane drveta "naviše", a
  // povratne grane "naniže"
  for (int cvor = 0; cvor < n; cvor++)
    for (int sused : listaSuseda[cvor]) {
      // u neusmerenom grafu je svaka grana predstavljena dva puta
```

```
      // ovim se obezbeđuje da će se svaka grana razmatrati samo jednom
      if (cvor > sused) continue;
      // (u, v) je ta grana usmerena od pretka ka potomku
      int u = cvor, v = sused;
      if (dolazna[u] > dolazna[v])
        swap(u, v);

      // usmeravamo granu na pravi način
      if (roditelj[v] == u)
        // grana drveta
        cout << u+1 << " " << v+1 << endl;
      else
        // povratna grana
        cout << v+1 << " " << u+1 << endl;
    }
}
```

2.6 Компоненте јаке повезаности графа

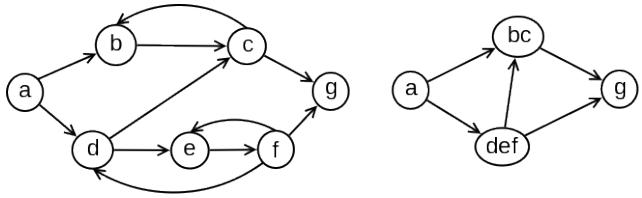
За усмерени граф кажемо да је *јако повезан* (енгл. strongly connected) ако је сваки чвор графа достижан из сваког другог чвора у графу.

На скупу чворова усмереног графа $G = (V, E)$ може се дефинисати релација $\sim$ *обостране достижности*: $u \sim v$ ако је чвор u достижан из чвора v и чвор v је достижан из чвора u. По дефиницији за сваки чвор u важи $u \sim u$ (напоменимо да то не значи да у графу постоје петље). Приметимо да су два чвора обострано достижна ако и само ако припадају неком заједничком усмереном циклусу. За релацију обостране достижности важи да је:

- рефлексивна – за сваки чвор $u \in V$ важи $u \sim u$,
- симетрична – за свака два чвора $u, v \in V$ важи $u \sim v$ ако и само ако $v \sim u$,
- транзитивна – за свака три чвора $u, v, w \in V$ из $u \sim v$ и $v \sim w$ следи и $u \sim w$.

Релација обостране достижности је стога релација еквиваленције. Она разлаже скуп чворова V у класе еквиваленције које називамо *компонентама јаке повезаности* графа G (енгл. strongly connected components). На слици 2.42 (лево) приказан је усмерени граф G који има четири компоненте јаке повезаности, које се састоје редом од чворова $\{a\}$, $\{b, c\}$, $\{d, e, f\}$ и $\{g\}$.

Компоненте јаке повезаности у графу имају разне примене. Размотримо пример дру-



Слика 2.42: Граф G и одговарајући кондезовани граф G^C чији чворови одговарају компонентама јаке повезаности графа G.

штвене мреже у којој корисници могу пратити акције других корисника. Потребно је идентификовати групе корисника који су тесно повезани једни са другима: свако од њих прати сваког другог, директно или индиректно. Овај проблем одговара проблему одређивања компоненти јаке повезаности у графу у коме су корисници чворови, а усмерене гране одговарају релацији праћења корисника. У софтверским системима модули се могу представити чворовима, а зависности међу њима усмереним гранама графа. Идентификовање компоненти јаке повезаности помаже у проналажењу циклуса или потенцијалних проблема у структури међузависности модула, који могу бити од помоћи у унапређењу дизајна софтвера.

Од графа G може се формирати *кондензовани* или *компримовани* граф G^C: то је усмерени ациклички граф који садржи информације о компонентама јаке повезаности графа G (слика 2.42, десно). Наиме, сваки чвор у графу G^C одговара једној компоненти јаке повезаности графа G, а два чвора у графу G^C су повезана граном ако и само ако у графу G постоји бар једна грана од неког чвора прве компоненте до неког чвора друге компоненте јаке повезаности. Јасно је да је граф G^C ациклички: ако би у њему постојао циклус, то би значило да се све компоненте јаке повезаности које припадају циклусу могу спојити у једну, већу компоненту јаке повезаности. У наставку текста ћемо компоненте јаке повезаности звати краће само компоненте.

Решавамо следећи проблем.

Проблем

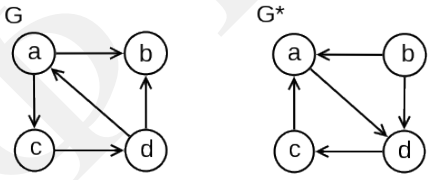
За дати усмерени граф $G = (V, E)$ одредити све компоненте јаке повезаности.

У наставку текста размотрићемо неколико варијанти алгоритма за решавање овог проблема заснованих на обиласку из сваког чвора, а затим и Тарџанов алгоритам и Косарацуов алгоритам.

2.6.1 Алгоритам заснован на обиласку из свих чворова

Компонента јаке повезаности којој припада чвор $v \in V$ може да се одреди тако што се за сваки чвор u достижан из v покрене DFS обилазак са циљем да се установи да ли је v достижан из u. Поступак се затим може поновити за чворове који не припадају претходно издвојеним компонентама (ако такви чворови постоје). Поступак је неефикасан јер захтева велики број покретања алгоритма DFS.

Описани поступак може се усавршити. Нека је G^* граф који садржи исте чворове као граф G, али супротно усмерене гране: грана (v, u) припада графу G^* ако и само ако грана (u, v) припада графу G (видети слику 2.43). Овај граф зваћемо *транспонованим графом* (енгл. transpose graph) графа G. Компоненте можемо да одредимо и на следећи начин: покренемо DFS обилазак из чвора v_0 у оба графа, G и G^*. Ако је чвор u достижан из чвора v_0 у графу G то значи да обиласком графа G из v_0 у оригиналном графу G можемо доћи до u неким усмереним путем. Обртањем усмерења грана се чворовима "мењају улоге'' и чвор v_0 је достижан из u у графу G^*. Дакле, DFS обилазак покренут из чвора v_0 у графу G проналази скуп чворова A који су достижни из чвора v_0, а DFS обилазак покренут из чвора v_0 у графу G^* скуп чворова B из којих је достижан чвор v_0. Компонента јаке повезаности којој припада чвор v_0 једнака је $A \cap B$. Овај поступак понављамо за произвољни чвор који не припада до сада одређеним компонентама повезаности, уколико такав чвор постоји. Најгори сценарио је када су сви чворови засебне компоненте и тада је сложеност $O(|V|(|V| + |E|))$.



Слика 2.43: Граф G и њему транспоновани граф G^*.

Пример 2.6.1

Размотримо графове G и G^* приказане на слици 2.43: DFS обилазак графа G покренут из чвора d обилази скуп чворова $A = \{d, a, b, c\}$, док DFS обилазак графа G^* покренут из чвора d обилази скуп чворова $B = \{d, c, a\}$. Важи $A \cap B = \{d, a, c\}$, те чворови d, a и c припадају истој компоненти јаке повезаности. Приметимо да једино чвор b не припада истој компоненти повезаности као чвор d, а пошто DFS обилазак из чвора b у графу G обилази само чвор b (јер је његов излазни степен 0), то чвор b чини сам за себе другу (преосталу) компоненту јаке повезаности графа G.

Постоји неколико различитих алгоритама линеарне временске сложености за одређи-

вање компоненти јаке повезаности у усмереном графу, а најпознатији међу њима су Тарџанов алгоритам и Косарајуов алгоритам. Оба алгоритма су заснована на DFS обиласку графа, само се код Тарџановог алгоритма све ради у једном пролазу кроз граф, док се у Косарајуовом алгоритму два пута позива алгоритам DFS.

2.6.2 Тарџанов алгоритам

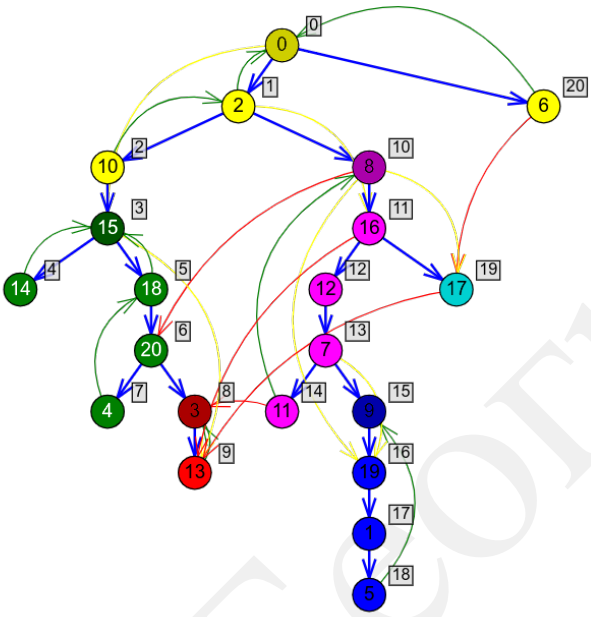
Као што је то често случај код графовских алгоритама и издвајање компонената јаке повезаности се заснива на пажљивој анализи DFS дрвета и његових особина. Приликом DFS обиласка датог усмереног графа G имплицитно се формира DFS дрво, односно DFS шума. Једноставности ради, можемо претпоставити да се DFS шума састоји од једног дрвета. Наиме, свако дрво DFS шуме се може анализирати засебно, јер је јасно да су чворови сваке компоненте јаке повезаности подскуп чворова неког појединачног дрвета у DFS шуми. Алтернативно, као што је објашњено у поглављу 2.3.1.2, граф можемо проширити новим чвором v улазног степена 0 који је повезан гранама са свим осталим чворовима. Проширени граф сем компоненти графа G има само још једну додатну једночлану компоненту $\{v\}$. Наиме, ниједан други чвор не може бити са њим у компоненти јаке повезаности, јер је улазни степен чвора v једнак 0.

У наредном разматрању ћемо у потпуности занемарити директне гране (оне гране које спајају претке са потомцима). Наиме, од сваког претка до потомка се већ може стићи гранама DFS дрвета, па се уклањањем директних грана не мења достижност чворова и компоненте јаке повезаности остају непромењене.

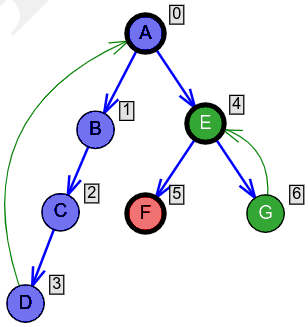
2.6.2.1 Распоред компоненти у DFS дрвету

Размотримо граф приказан на слици 2.44. Компоненте јаке повезаности су обележене разним бојама. Запажа се да су чворови сваке компоненте јаке повезаности груписани у DFS дрвету, тј. да је свака компонента део неког поддрвета DFS дрвета. У доста случајева чворови чак имају и узастопне редне бројеве, мада то не мора бити увек случај (на пример, последњи чвор жуте компоненте се открива тек након што се открију и обраде све остале компоненте). Овај однос положаја компоненти унутар DFS дрвета је кључан елемент Тарџановог алгоритма и у наставку ћемо се потрудити да га прецизније испитамо и формално докажемо његова својства.

Приликом DFS обиласка важно нам је да знамо када смо прешли из једне компоненте у другу. У сваку компоненту улазимо тако што најпре посетимо њен чвор који има најмањи редни број (долазни DFS број). Назовимо *базним чвором* b (енгл. base vertex) компоненте X онај чвор те компоненте који има најмањи редни број при долазној DFS нумерацији, тј. онај чвор $b \in X$ за који важи $b.Pre = \min_{v \in X} v.Pre$.



Слика 2.44: Компоненте јаке повезаности обележене бојама на DFS дрвету. Поред сваког чвора наведен је његов редни број у долазној DFS нумерацији. Базни чвор сваке компоненте приказан је мало тамнијом бојом.



Слика 2.45: Граф који садржи три компоненте јаке повезаности графа. Уз сваки чвор дат је његов редни број у долазној DFS нумерацији. Базни чворови сваке компоненте су подебљани.

Пример 2.6.2

Граф са слике 2.45 садржи три компоненте јаке повезаности: $\{A, B, C, D\}$, $\{E, G\}$ *и* $\{F\}$. *Базни чвор прве компоненте је чвор* A, *друге* E, *а треће* F.

Базни чворови графа са слике 2.44 су $0, 15, 3, 8, 9$ *и* 17.

Наредни пример ће нам указати на важне односе који у свакој компоненти важе између њеног базног чвора и осталих чворова. Те односе ћемо затим формализовати и доказати у леми 2.6.1.

Пример 2.6.3

У примерима на сликама 2.44 и 2.45 запажа се да су сви чворови сваке компоненте потомци њеног базног чвора у DFS дрвету. Додатно, на путу од базног чвора до било ког другог чвора у тој компоненти кроз гране DFS дрвета не може да буде прекида, тј. пут садржи само чворове те компоненте. На пример, у графу приказаном на слици 2.45 једној компоненти повезаности припадају чворови $\{A, B, C, D\}$ *и чворови* B, C *и* D *јесу потомци базног чвора те компоненте – чвора* A *у DFS дрвету. Слично важи и за чвор* G *који је потомак базног чвора* E *своје компоненте. Посебно, с обзиром да чворови* A *и* D *припадају истој, првој компоненти јаке повезаности графа, то важи и за чворове* B *и* C *који се налазе на путу од чвора* A *до чвора* D *кроз гране DFS дрвета. Наредна лема доказује да то није случајно. Истакнимо и да обратно не мора да важи, тј. јасно је да могу постојати потомци базног чвора у DFS дрвету који не припадају његовој компоненти: на пример, чворови* E, F *и* G *графа са слике 2.45 су потомци чвора* A, *али се не налазе са њим у истој компоненти.*

Наредна лема показује да запажања из примера 2.6.3 важе и у општем случају.

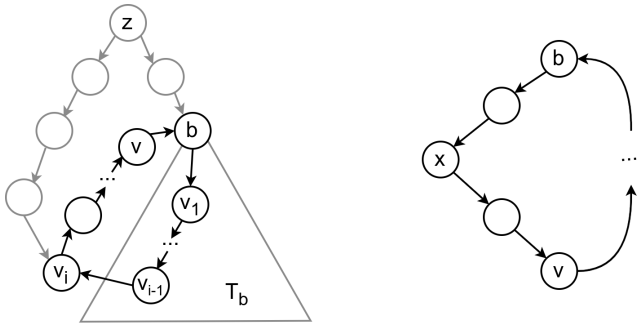
Лема 2.6.1 **[Однос базних и осталих чворова компоненте]**

Нека је b *базни чвор компоненте* X. *Тада:*

- *сваки чвор* $v \in X$ *је потомак чвора* b *у DFS дрвету,*
- *сви чворови на путу од* b *до* $v \in X$ *преко грана DFS дрвета припадају компоненти* X.

Доказ. Претпоставимо супротно, односно да у компоненти X постоје чворови који нису у поддрвету T_b DFS дрвета са кореном у b, и нека је v један од таквих чворова. Нека је $(v_0 = b, v_1, \ldots, v_k = v)$ пут од b до v кроз чворове компоненте X. Нека је v_i први чвор на том путу који није у поддрвету T_b (дакле, чвор b је предак чворова

v_j за $j = 1, \ldots, i-1$). Тада је грана (v_{i-1}, v_i) попречна у односу на DFS дрво. Наиме, пошто v_i није у поддрвету T_b, грана (v_{i-1}, v_i) није грана DFS дрвета, а не може бити ни повратна, јер би онда чвор v_i био предак чвора b и b не би био базни чвор те компоненте. Све попречне гране у односу на DFS дрво су усмерене улево, па и грана (v_{i-1}, v_i). Дакле, чвор v_i је лево од чвора v_{i-1}, па важи v_i.*Pre* $< v_{i-1}$.*Pre*. Додатно, они имају заједничког претка z у DFS дрвету (слика 2.46, лево).



Слика 2.46: Илустрације уз доказ леме 2.6.1.

Чвор z не може бити један од чворова $v_0 = b, v_1, v_2, \ldots, v_{i-1}$, јер би иначе чвор v_i био у поддрвету са кореном у чвору b. Пошто је b предак чвора v_{i-1}, онда је z и заједнички предак чворова v_i и b, те је чвор v_i лево и од чвора b, па важи v_i.*Pre* $< b$.*Pre*. Притом, чвор v_i припада компоненти у којој је базни чвор b, јер је обострано достижан са чвором b: b је достижан из чвора v_i путем од v_i до $v_k = v$ и даље путем од v_k до b који постоји јер је v, по претпоставци, у овој компоненти. Ово је у супротности са претпоставком да је b базни чвор ове компоненте. Дакле, сви чворови компоненте X се налазе у поддрвету T_b DFS дрвета.

Докажимо други део леме. Нека је x произвољни чвор на путу од b до v (слика 2.46, десно). Постоји пут од чвора b до чвора x кроз гране DFS дрвета, а такође и пут од чвора x до чвора b: тај пут добија се надовезивањем пута од x до v кроз гране DFS дрвета и пута од v до b (овај пут постоји јер чворови v и b припадају истој компоненти). Стога је чвор x у истој компоненти као и чвор b. Закључујемо да сви чворови на путу од чвора b до чвора v кроз гране DFS дрвета припадају компоненти чији је базни чвор b. □

Наредни пример сугерише чињеницу да се чворови сваке компоненте добијају тако што се из поддрвета чији је корен базни чвор те компоненте уклоне друге компоненте. Ово

ћемо формализовати и доказати у леми 2.6.2.

Пример 2.6.4

На сликама 2.44 и 2.45 запажа се да се сви чворови који припадају било којој компоненти јаке повезаности добијају тако што се из дрвета чији је корен базни чвор те компоненте уклоне поддрвета чији су коренови базни чворови других компоненти. На пример, ако на слици 2.44 из дрвета чији је корен чвор 0*, одсечемо поддрвета чији су корени чворови* 15 *и* 8 *остају само жути чворови који чине једну компоненту. Проверите да ово важи и за све остале компоненте.*

У графу приказаном на слици 2.45 чвор A је базни чвор. Чворови E и F су такође базни чворови и притом су потомци базног чвора A. Приметимо да су чворови који припадају компоненти чији је базни чвор A они који су потомци чвора A, а нису потомци ни чвора E ни чвора F: то су чворови B, C и D. Слично, чворови који припадају компоненти са базним чвором E су они који су потомци од чвора E, а нису потомци чвора F, што је само чвор G.

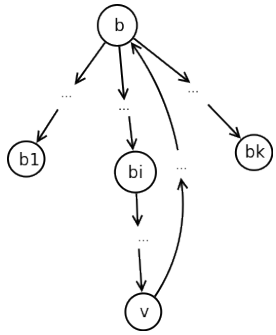
Наредна лема доказује да запажања из примера 2.6.4 важе у општем случају.

Лема 2.6.2 **[Одсецање других компоненти]**

Нека је b базни чвор неке компоненте и нека су $b_1, b_2, \ldots, b_k$ базни чворови неких других компоненти који су потомци чвора b у односу на DFS дрво. Тада се компонента којој припада чвор b састоји од свих потомака чвора b који нису потомци неког од чворова $b_1, b_2, \ldots, b_k$.

Доказ. Нека је чвор v у истој компоненти као и чвор b. На основу леме 2.6.1 он мора бити потомак у односу на DFS дрво чвора b. Претпоставимо супротно тврђењу, да је он потомак и чвора чвора b_i за неко i, $1 \leq i \leq k$ (слика 2.47).

Пошто је b_i потомак чвора b, онда постоји пут кроз гране DFS дрвета од чвора b до чвора b_i. Слично, пошто је чвор v потомак чвора b_i постоји пут од чвора b_i до чвора v, а због тога што су v и b у истој компоненти, постоји и пут од чвора v до чвора b. Надовезивањем ова два пута добија се пут од чвора b_i до b. На основу тога што је чвор b достижан из чвора b_i и чвор b_i достижан из чвора b следи да су чворови b и b_i у истој компоненти што је у супротности са претпоставком леме јер су b и b_i базни чворови различитих компоненти. Дакле, чворови који су у истој компоненти као и чвор b не могу бити потомци и неког другог базног чвора b_i, који је потомак чвора b. □



Слика 2.47: Илустрација уз доказ леме 2.6.2.

2.6.2.2 Издвајање компоненти уз помоћ стека

Под претпоставком да некако умемо да одредимо базне чворове свих компоненти јаке повезаности (што је проблем којим ћемо се позабавити у поглављу 2.6.2.3), леме 2.6.1 и 2.6.2 нам дају могућност да одредимо који су чворови заједно са неким базним чвором у истој компоненти јаке повезаности. И више од тога, овакав распоред компонената нам даје могућност да их прилично једноставно набројимо током DFS обиласка.

Потребно је да у неку погодну структуру података смештамо чворове у редоследу DFS обиласка, а да непосредно пре него што напустимо неки чвор проверимо да ли је он базни и ако јесте испишемо (и уклонимо) њега и све његове потомке који се и даље налазе у тој структури података: то ће бити чворови који су након њега посећени, и након њега додати у ту структуру. Дакле, елементе треба додавати на крај ове структуре података и уклањати их са краја ове структуре. Структура података коју је погодно користити у ове сврхе је стек. Наиме, приликом означавања чвора v у току DFS обиласка, чвор v се уписује на засебан наменски стек намењен набрајању компонената јаке повезаности. Када се заврши позив алгоритма DFS из чвора v, у склопу излазне обраде, ако је v базни чвор неке компоненте, са стека се уклања чвор v и сви чворови изнад њега на стеку (наравно, уназад: од елемента на врху стека па све до чвора v). На тај начин ако је чвор v базни, издваја се компонента која садржи чвор v и сви њени чворови уклањају се са стека. Редослед уклањања базних чворова нам гарантује да ћемо са базним чворовима неке компоненте уклонити само оне потомке који нису потомци неког другог базног чвора. Описани поступак приказан је у алгоритму 5.

Докажимо коректност овог алгоритма.

Теорема 2.6.1 **[Коректност набрајања компоненти помоћу стека]**

За сваки базни чвор b компоненте јаке повезаности графа G важи следеће:

- *При улазној обради чвора b, он се поставља на стек.*

Алгоритам 5 Издвајање компоненти јаке повезаности уз помоћ стека

```
1: procedure DFS(почетни чвор u)
2:     означи чвор u
3:     стави чвор u на стек
4:     for all чвор v који је sused чвора u do
5:         if чвор v није означен then
6:             DFS(v)
7:     if чвор u је базни чвор компоненте then
8:         скидај елементе са стека све док се не скине u
9:         скинути чворови су сви чворови компоненте чији је базни чвор u
10:
11: for all неозначен чвор u do
12:     DFS(u)
```

- *Пре његове излазне обраде исправно су обрађене све компоненте јаке повезаности у поддрвету чији је он корен. На врху стека, изнад чвора b, налазе се сви чворови његове компоненте повезаности.*
- *Након излазне обраде чвора b он се уклања са стека, заједно са свим чворовима његове компоненте, након чега је садржај стека исти као при улазу у чвор b.*

Доказ. Доказ се може извести индукцијом по броју m компоненти јаке повезаности које чине граф.

За $m = 1$, постоји само један базни чвор b и он је корен целог DFS дрвета. У првом кораку DFS обраде он се поставља на празан стек, затим се сви остали чворови, један по један стављају на стек, све док се на самом крају DFS обиласка не дође до излазне обраде чвора b. У том тренутку (пошто је он једини базни чвор), сви чворови се скидају са стека, обрађује се исправно једина компонента графа и стек на крају остаје празан, као што је и био пре уласка у чвор b.

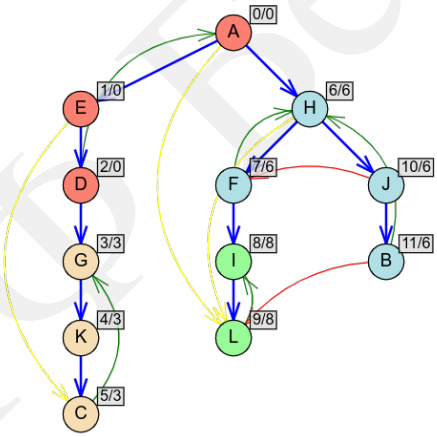
Нека је тврђење тачно за графове са мање од m компоненти и нека је G граф са m компоненти. Нека је b први базни чвор на који се наилази при DFS обиласку (то је чвор из ког покрећемо DFS обилазак), а нека су $b_1, b_2, \ldots, b_{m-1}$ његови потомци, који су базни чворови осталих компоненти. Сваки од њих ће током DFS обиласка из чвора b у неком тренутку бити стављен на стек. Према индуктивној хипотези, после завршетка DFS обиласка из чвора b_i, $1 \leq i \leq m-1$, издвојене су све компоненте достижне из чвора b_i и сви њихови чворови су уклоњени су са стека, остављајући стек сваки пут у стању као пре улаза у чвор b_i. То значи да се у тренутку излазне обраде чвора b на

стеку налазе сви чворови дрвета са кореном b (потомци чвора b), који не припадају ни једном дрвету чији је корен неки од чворова $b_1, \ldots, b_{m-1}$. Међутим, на основу леме 2.6.2 знамо да су то тачно чворови компоненте чији је базни чвор b. Приликом излазне обраде чвора b та компонента се исправно издваја, њени чворови се скидају са стека и стек остаје празан. □

2.6.2.3 Одређивање базних чворова компоненти

Да би Тарџанов алгоритам био комплетан, недостаје још само да формулишемо неки ефективни тест којим бисмо могли да за дати чвор испитамо да ли је он базни чвор своје компоненте.

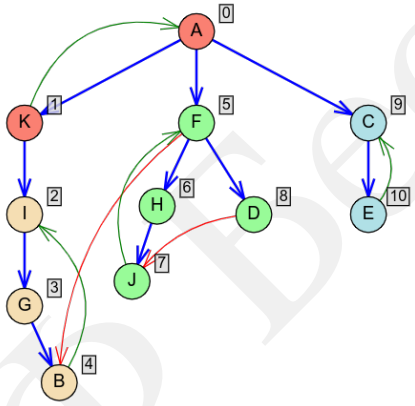
У свакој компоненти јаке повезаности могуће је стићи од било ког чвора до било ког другог чвора. Зато се од сваког чвора компоненте може стићи до њеног базног чвора, што је „најнижи" тј. чвор са најмањим редним бројем у тој компоненти. Пожељно је зато да за сваки чвор испитамо који је „најнижи" чвор до ког се може стићи у дрвету, у нади да ће то увек бити базни чвор компоненте.



Слика 2.48: Уз сваки чвор графа означен је његов редни DFS број и најмањи редни број чвора до ког је могуће вратити се из тог чвора. Ситуација изгледа идеално – из сваког чвора је могуће вратити се тачно до базног чвора његове компоненте.

Размотримо граф на слици 2.48. Видимо да је на овом графу наступила идеална ситуација, јер је „најнижи" чвор до ког је могуће вратити се из произвољног чвора графа увек базни чвор компоненте којој тај чвор припада. Базни чворови се онда лако препознају тако што су то они чворови којима се редни број поклапа са најмањим бројем чвора до ког се из тог чвора можемо вратити.

Ипак, проблем није тако једноставан, јер понекад попречне гране могу да нас одведу до чвора који има мањи редни број од редног броја базног чвора компоненте. Размотримо сада граф на слици 2.49. Због попречне гране (F, B) се из чвора F (са редним бројем 5) можемо вратити до чвора B (са редним бројем 4), па затим до чвора I (са редним бројем 2). Дакле, најмањи редни број чвора до кога можемо да се вратимо из чвора F (са редним бројем 5) је 2, па, пошто тај број није једнак редном броју чвора F, чвор F не препознајемо као базни чвор компоненте, а он то јесте. Дакле, разматрање попречне гране (F, B) показује да једноставан критеријум који смо формулисали није сасвим коректан. С друге стране, ако се попречна грана (D, J) не би разматрала, не би било могуће установити да се из чвора D (са редним бројем 8) можемо вратити до чвора J (са редним бројем 7), па затим уназад до чвора F (са редним бројем 5), и не би било јасно да чвор D припада компоненти чији је базни чвор F.



Слика 2.49: Грана (D, J) мора бити разматрана у оквиру путева ка чворовима са мањим редним бројем јер води ка чвору исте компоненте (иначе се не би могло видети да се од D можемо стићи до F), а грана (F, B) не сме бити разматрана, јер води ка чвору различите компоненте (иначе би деловало да се од F може стићи до нижег чвора).

Јасно је да морамо модификовати критеријум тако да укључи разматрање неких, а искључи разматрање неких других попречних грана. Видимо да не треба одређивати најмањи редни број *произвољног* чвора до ког се можемо „спустити" из датог чвора v, јер тај чвор може да буде такав да не припада истој компоненти којој припада и чвор v. Нас заправо занима да за сваки чвор v одредимо најмањи број чвора *у његовој компоненти* до ког је могуће вратити се. Лако се може доказати да је чвор базни чвор компоненте ако и само ако се тај број поклапа са његовим редним бројем. Директне гране, већ смо констатовали, немају никаквог утицаја на одређивање овог броја. Допуштено је „подизати се" гранама дрвета, а „спуштати се" повратним гранама (јер повратне гране увек спајају два чвора која су у истој компоненти, пошто креирају циклус са гранама дрвета

између та два чвора) и „спуштати се” попречним гранама (ка чворовима са мањим редним бројевима), али не свим попречним гранама, већ само оним које спајају чворове који су у истој компоненти (попречна грана може повезивати два чвора из исте или из различите компоненте, слика 2.49).

Иако делује да смо проблем одређивања свих чворова неке компоненте свели на проблем одређивања њеног базног чвора компоненте, а проблем одређивања базног чвора компоненте на познавање чворова компоненте, видећемо ускоро да нисмо направили циркуларну дефиницију и да смо се приближили решењу проблема.

Описани бројеви (најмањи редни број чвора унутар компоненте до ког се можемо попети) дају јасан критеријум за одређивање базних чворова, али се не могу једноставно одредити коришћењем једног DFS обиласка (што је илустровано примером 2.6.5). Уместо њих, користе се редни бројеви чворова унутар компоненте, али до којих се може стићи само кретањем низ гране дрвета, а затим повратком уз највише једну попречну или повратну грану. За сваки чвор одређујемо *„најнижи” чвор његове компоненте достижан једном повратном или попречном граном*. Редни број тог чвора поново обележавамо са $L(v)$ (енгл. lowlink), као у поглављу 2.5, али јасно је да $L(v)$ мора бити дефинисан нешто другачије него у случају неусмерених графова. Нека је X компонента која садржи чвор v. Означимо са M најмањи редни број чвора компоненте X до кога се из чвора v може стићи путем који се састоји од грана DFS дрвета и који се завршава највише једном повратном или попречном граном. Вредност $L(v)$ дефинишемо као $\min\{v.Pre, M\}$, односно:

$$L(v) = \min\{v.Pre, \min_{\substack{w \in X \\ \text{постоји грана } (t,w),\ t \in T_v \\ (t,w) \text{ је попречна или повратна}}} w.Pre\}.$$

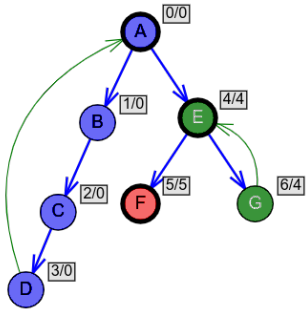
На слици 2.50 приказан је граф са слике 2.45, при чему је уз сваки чвор v, поред редног броја у долазној нумерацији, приказана и вредност $L(v)$ чвора. Базни чворови су тачно они чворови за које важи $v.Pre = L(v)$.

Пример 2.6.5

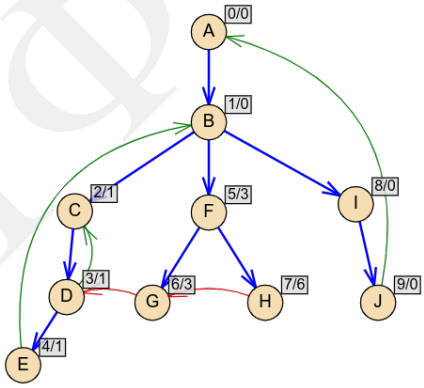
Граф на слици 2.51 има једну компоненту и на њему се може видети како се „најнижи” чвор у компоненти до кога се може доћи може одредити праћењем ланца путања одређеног вредностима функције L.

На пример,

- $L(F) = 3$, *јер се од чвора F граном дрвета стиже до G, па затим попречном граном до чвора D који има редни број* 3. *Дакле, из дрвета са кореном F се једном*



Слика 2.50: Граф код кога су уз сваки чвор v приказане вредности $v.Pre$ и $L(v)$. Чворови за које важи $L(v) = v.Pre$ су базни чворови компоненти повезаности.



Слика 2.51: Граф са једном компонентом јаке повезаности. Уз сваки чвор v је написан његов редни број $v.Pre$ и вредност $L(v)$.

попречном граном враћамо у дрво са кореном D.

- $L(D) = 1$, *јер се од чвора D граном дрвета стиже до чвора E, па затим повратном граном до чвора B који има редни број* 1. *Дакле из дрвета са кореном D се једном повратном граном враћамо у дрво са кореном B.*
- $L(B) = 0$, *јер се од чвора B гранама дрвета редом стиже до чворова I, па затим J, а онда се повратном граном од J враћамо у чвор A који има редни број* 0. *Дакле, из дрвета са кореном B се једном повратном граном можемо вратити у базни чвор A са редним бројем* 0.

Да би се за чвор F установило да се може вратити у чвор A са редним бројем 0 *потребно је, између осталог, да видимо грану (J, A), што се дешава тек пред крај DFS обиласка, након што је обилазак чвора F завршен. Ово указује на то да је за одређивање најнижег чвора у компоненти до ког се може стићи (што је увек базни чвор) потребно више пута обилазити граф. Видећемо да то није случај са вредностима $L(v)$ и да се оне могу лако одредити током јединог обиласка графа који врши Тарџанов алгоритам.*

Аналогно леми 2.5.2, могуће је формулисати лему која описује рекурзивне везе између вредности функције L између чворова DFS дрвета (на основу које следи поступак за одређивање вредности функције L током DFS обиласка).

Било који чвор који није базни има мању вредност $L(v)$ од свог редног броја $v.Pre$. Заиста, постоји пут од њега до базног чвора, који у неком тренутку мора да „побегне" из поддрвета са кореном v, било повратном граном било попречном граном ка чвору у истој компоненти и да нас доведе до чвора са мањим редним бројем. Пратећи ланац оваквих путања, доћи ћемо у једном тренутку и до базног чвора за који је $L(v) = v.Pre$. Докажимо ово и формално.

Лема 2.6.3

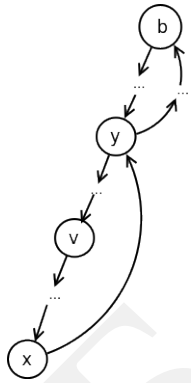
Чвор v је базни чвор ако и само ако важи $L(v) = v.Pre$.

Доказ. Покажимо први смер тврђења: ако је чвор v базни онда важи $L(v) = v.Pre$. Претпоставимо супротно, односно да за базни чвор v важи $L(v) < v.Pre$ и покажимо да онда чвор v није базни чвор. Према дефиницији вредности L, постоји чвор w у истој компоненти као и v такав да је $L(v) = w.Pre$. Стога је $w.Pre < v.Pre$ те чвор v није базни чвор.

Докажимо сада супротни смер импликације: ако за чвор v важи услов $L(v) = v.Pre$, онда је чвор v базни. Претпоставимо супротно: да важи услов $L(v) = v.Pre$, а да чвор

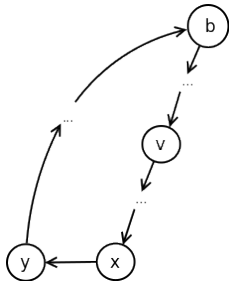
v није базни чвор. Нека је b базни чвор компоненте која садржи чвор v. Према леми 2.6.1 чвор b је предак чвора v. Пошто су b и v у истој компоненти, постоји прости пут p од v до b. Нека је y први чвор на путу p који није у поддрвету са кореном у v и нека је x чвор који му претходи на путу p:

- ако је грана (x, y) повратна (слика 2.52), онда је y предак чвора v и $L(v) \leq y.Pre < v.Pre$, супротно претпоставци (чвор y није на путу од v до x, јер је пут p по претпоставци прост);



Слика 2.52: Случај када је грана (x, y) повратна у доказу леме 2.6.3.

- ако је грана (x, y) попречна (слика 2.53), онда је она усмерена улево и $y.Pre < x.Pre$. С обзиром на то да су поддрво са кореном у y и поддрво са кореном у v дисјунктни и да постоји грана од потомка чвора v (односно x) до y, на основу својстава попречних грана можемо закључити да важи $y.Pre < v.Pre$. Одавде следи да је $L(v) \leq y.Pre$, јер је y у истој компоненти јаке повезаности као и v и достижан је преко низа грана DFS дрвета након кога следи једна повратна или попречна грана. На основу овога и чињенице да је $y.Pre < v.Pre$, добијамо да важи $L(v) < v.Pre$, супротно претпоставци.



Слика 2.53: Случај када је грана (x, y) попречна у доказу леме 2.6.3.

□

Финализујмо опис алгоритма тако што ћемо приказати како се одређују вредности $L(v)$. Приликом уласка у чвор u током DFS обиласка додељујемо му редни број $v.Pre$ и иницијализујемо $L(v) = v.Pre$. Затим обрађујемо све гране (v, w) које воде из чвора v до неког чвора w.

- Ако чвор w није посећен, рекурзивно вршимо његов обилазак и грана (v, w) постаје грана DFS дрвета. Након рекурзивног позива позната је вредност $L(w)$, па вредност $L(v)$ ажурирамо на вредност $\min\{L(v), L(w)\}$.
- Ако је чвор w посећен, а грана (v, w) је повратна или попречна, а чвор v је у истој компоненти, анализирамо и ажурирамо вредност $L(v)$ на вредност $\min\{L(v), w.Pre\}$ – ово је у складу са дефиницијом вредности L, и не користи се израз $\min\{L(v), L(w)\}$ јер је допуштен прелаз преко произвољно много грана дрвета, али само преко једне повратне или попречне гране.
- Ако је чвор w посећен, а грана (v, w) је директна или је попречна, а чвор w је у некој другој компоненти, ову грану просто прескачемо.

Остаје питање како ефикасно утврдити да ли попречна грана (v, w) грана води ка чвору исте или друге компоненте повезаности. Једно решење је да се примети да се оно може свести на питање да ли се у тренутку обраде ове гране њен крајњи чвор w налази у наменском стеку намењеном набрајању компоненти или не. Дајмо прво мало прецизнију карактеризацију елемената који се налазе на стеку.

Лема 2.6.4 **[Чворови који остају на стеку након излазне обраде]**

Чвор се налази на стеку након његове излазне обраде ако и само ако постоји пут у графу од њега до неког чвора који се тренутно налази испод њега на стеку

Доказ. Ако би се на стеку налазио чвор током чије је излазне обраде утврђено да се у графу не може од тог чвора доћи до неког чвора испод њега на стеку, тај чвор би био базни чвор компоненте и морао би приликом своје излазне обраде бити уклоњен са стека заједно са осталим чворовима његове компоненте. □

Пример 2.6.6

У графу на слици 2.49 се у тренутку улазне обраде чвора F на стеку налазе чворови A и K при чему је завршена излазна обрада само чвора K. Он се налази на стеку јер постоји пут од њега до чвора A који је испод њега на стеку.

Сада можемо доказати и карактеризацију попречних грана.

Лема 2.6.5 **[Попречне гране и стек]**

Попречна грана (v, w) води из чвора v ка чвору w који је у тренутку обраде чвора v на стеку ако и само ако су v и w у истој компоненти повезаности.

Доказ. Ако је грана (v, w) попречна, чвор w је лево од v. Он ће као такав бити посећен пре чвора v и притом додат на стек, а у тренутку посете чвора v његова излазна обрада ће већ бити завршена.

Ако чвор w није више на стеку приликом обраде чвора v, он је морао бити уклоњен са стека заједно са свим чворовима своје компоненте, што значи да није у истој компоненти са v.

Ако је чвор w и даље на стеку приликом обраде чвора v, на основу леме 2.6.4, пратећи ланац путања ка све нижим и нижим чворовима на стеку, из њега ћемо се „спустити" до неког заједничког претка чворова v и w, чиме се обезбеђује да су чворови v и w узајамно достижни и налазе се у истој компоненти. □

Пример 2.6.7

У графу на слици 2.49 попречна грана (F, B) води ка чвору B који се не налази више на стеку у тренутку обраде чвора F, па F и B нису у истој компоненти. Попречна грана (D, J) води ка чвору J који је и даље на стеку у тренутку обраде чвора D, па су чворови D и J у истој компоненти.

Дакле, важи следеће:

- попречна грана (v, w) води ка чвору w који је у тренутку обраде чвора v на стеку ако и само су v и w у истој компоненти повезаности;
- повратна грана (v, w), јасно, увек води ка чвору w који је у тренутку обраде чвора v на стеку (јер је w предак од v, па је завршена његова улазна обрада током које је је стављен на стек, али није завршена његова излазна обрада нити излазна обрада базног чвора његове компоненте, па w није још могао бити скинут са стека);
- ако директна грана (v, w) води ка чвору w на стеку, важи да је $w.Pre > v.Pre$, па та грана не може утицати на вредност $L(v)$.

Зато у коду можемо проверавати оне гране (v, w) које воде од чвора v до чвора w који се налази на стеку, не анализирајући посебно њихову врсту.

Испитивање да ли је чвор на стеку проласком кроз стек није ефикасно, па додатно користимо наменски низ у коме ћемо током извршавања алгоритма памтити за сваки чвор да ли се тренутно налази на стеку или не. Алтернативно решење је да се приликом скидања чвора v са стека његове вредности $v.Pre$ и $L(v)$ поставе на $+\infty$. Када се то уради све попречне гране могу бити посећене, али до смањења вредности $L(v)$ може доћи само код грана ка чворовима који су још на стеку и налазе се у истој компоненти.

Сада можемо дати и заокружени псеудокод Тарјановог алгоритма.

Алгоритам 6 Тарјанов алгоритам за одређивање компонената јаке повезаности

procedure DFS(čvor v)
    dodeli redni broj $v.Pre$
    $L(v) = v.Pre$
    stavi v na stek
    for all sused w čvora v **do**
        if čvor w nije označen **then**
            DFS(w)
            $L(v) = \min\{L(v), L(w)\}$
        else if w je na steku **then**
            $L(v) = \min\{L(v), w.Pre\}$
    if $L(v) = v.Pre$ **then**
        skidaj elemente sa steka sve dok se ne skine v
        skinuti čvorovi su svi čvorovi komponente sa korenom v
for all neoznačen čvor v **do**
    DFS(v)

Тарјанов алгоритам за одређивање компоненти јаке повезаности ослања се на DFS обилазак графа, те је сложености $O(|V| + |E|)$.

Алгоритам се може имплементирати у језику C++ на следећи начин.

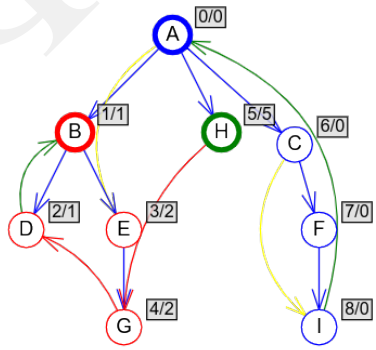
```
int vreme_dolazna = 1;

void dfsTarjan(int cvor, vector<int> &dolazna, vector<int> &lowlink,
               stack<int> &redosledUObilasku, vector<bool> &naSteku,
               vector<int>& komponente, int& komponenta) {
  dolazna[cvor] = lowlink[cvor] = vreme_dolazna;
  vreme_dolazna++;
  redosledUObilasku.push(cvor);
  naSteku[cvor] = true;
```

```
  // rekurzivno prolazimo kroz sve susede koje nismo obisli
  for (int sused : listaSuseda[cvor]) {
    // ako cvor 'sused' do sada nismo posetili
    if (dolazna[sused] == -1) {
      // pokrecemo DFS obilazak iz cvora 'sused'
      dfsTarjan(sused, dolazna, lowlink, redosledUObilasku, naSteku,
                komponente, komponenta);
      // ako je potrebno azuriramo vrednost L cvora 'cvor'
      if (lowlink[sused] < lowlink[cvor])
        lowlink[cvor] = lowlink[sused];
    }
    // ako je u pitanju povratna ili poprecna grana,
    // azuriramo vrednost L za cvor 'cvor'
    // samo ako se sused nalazi u steku
    // to znaci da sused pripada istoj komponenti povezanosti
    else if (naSteku[sused])
      if (dolazna[sused] < lowlink[cvor])
        lowlink[cvor] = dolazna[sused];
  }

  // ako je u pitanju bazni cvor komponente
  // stampamo sve cvorove te komponente
  if (dolazna[cvor] == lowlink[cvor]) {
    while (true) {
      // ispisujemo element sa vrha steka i uklanjamo ga
      int cvor_komponente = redosledUObilasku.top();
      komponente[cvor_komponente] = komponenta;
      naSteku[cvor_komponente] = false;
      redosledUObilasku.pop();
      // ako smo stigli do baznog cvora prekidamo petlju
      if (cvor_komponente == cvor)
        break;
    }
    komponenta++;
  }
}

// za svaki cvor odredjujemo redni broj komponente jake povezanosti
```

```
vector<int> tarjanSCC() {
  int brojCvorova = listaSuseda.size();
  vector<int> dolazna(brojCvorova, -1);
  vector<int> lowlink(brojCvorova);
  // stek na koji smestamo cvorove u redosledu DFS obilaska
  stack<int> redosledUObilasku;
  // vektor koji omogucava brzu proveru da li se cvor nalazi na steku
  vector<bool> naSteku(brojCvorova, false);
  // redni broj komponente svakog cvora
  vector<int> komponente(brojCvorova, -1);
  // redni broj tekuce komponente
  int komponenta = 0;

  // pokrecemo DFS iz svakog neposecenog cvora
  for (int cvor = 0; cvor < brojCvorova; cvor++)
     if (komponente[cvor] == -1)
        dfsTarjan(cvor, dolazna, lowlink, redosledUObilasku, naSteku,
                  komponente, komponenta);
  return komponente;
}
```

Пример 2.6.8

Размотримо извршавање овог алгоритма на примеру графа приказаног на слици 2.54.



Слика 2.54: Пример графа који има три компоненте јаке повезаности: $\{B, D, E, G\}$, $\{H\}$ и $\{A, C, F, I\}$. Уз сваки чвор приказан је редни број у долазној нумерацији и вредност функције L.

- *dfs*(A)
  стек: A
  иницијализујемо A.*Pre* $= L(A) = 0$
  - *грана* (A, B) *постаје грана дрвета*
    dfs(B)
    стек: A, B
    иницијализујемо B.*Pre* $= L(B) = 1$
    - *грана* (B, D) *постаје грана дрвета*
      dfs(D)
      стек: A, B, D
      иницијализујемо D.*Pre* $= L(D) = 2$
      - *грана* (D, B) *је повратна грана*
        смањујемо $L(D) = 1$ *излаз из* D: $L(D) = 1$, *а* D.*Pre* $= 2$, *па он није базни чвор компоненте*
        $L(B)$ *није веће од* $L(D)$, *па остаје* 1
    - *грана* (B, E) *постаје грана дрвета*
      dfs(E)
      стек: A, B, D, E
      иницијализујемо E.*Pre* $= L(E) = 3$
      - *грана* (E, G) *постаје грана дрвета*
        dfs(G)
        стек: A, B, D, E, G
        E.*Pre* $= L(E) = 4$
        - *грана* (G, D) *је попречна грана*
          D *је на стеку, па смањујемо* $L(G) = 2$ *излаз из* G: $L(G) = 2$, *а* G.*Pre* $= 4$, *па он није базни чвор компоненте*
          $L(E)$ *смањујемо на вредност* $L(G) = 2$ *излаз из* E: $L(E) = 2$, *а* E.*Pre* $= 3$, *па он није базни чвор компоненте*
          $L(B)$ *није веће од* $L(E)$, *па остаје* 1 *излаз из* B: $L(B) = B$.*Pre* $= 1$, *он јесте базни чвор компоненте*
          са стека се скидају G, E, D, B *који чине компоненту*
          стек: A
          $L(A)$ *није веће од* $L(B)$, *па остаје* 0
  - *директна грана* (A, E) *се прескаче јер* E *није на стеку*
  - *грана* (A, H) *постаје грана дрвета*
    dfs(H)
    стек: A, H
    иницијализујемо H.*Pre* $= L(H) = 5$
    - *попречна грана* (H, G) *се прескаче јер* G *није на стеку излаз из* H:

$L(H) = H.Pre = 5$, *па је он базни чвор компоненте*
са стека се скида: H *који чини компоненту*
стек: A
$L(A)$ *није веће од* $L(H)$, *па остаје* 0
– *грана* (A, C) *постаје грана дрвета*
$dfs(C)$
стек: A, C
иницијализујемо $C.Pre = L(C) = 6$
– *грана* (C, F) *постаје грана дрвета*
$dfs(F)$
стек: A, C, F
иницијализујемо $F.Pre = L(F) = 7$
- *грана* (F, I) *постаје грана дрвета*
$dfs(I)$
стек: A, C, F, I
иницијализујемо $I.Pre = L(I) = 8$
- *повратна грана* (I, A)
смањујемо $L(I) = 0$ *излаз из* I: $L(I) = 0$, *а* $I.Pre = 8$, *па он није базни чвор компоненте*
$L(F)$ *смањујемо на вредност* $L(I) = 0$ *излаз из* F: $L(F) = 0$, *а* $F.Pre = 7$, *па он није базни чвор компоненте*
$L(C)$ *смањујемо на вредност* $L(F) = 0$
– *директна грана* (C, I)
$L(C)$ *није веће од* $L(I)$, *па остаје* 0 *излаз из* C: $L(C) = 0$, *а* $C.Pre = 6$, *па он није базни чвор компоненте*
$L(A)$ *није веће од* $L(C)$, *па остаје* 0 *излаз из* A: $L(A) = A.Pre = 0$, *па је он базни чвор компоненте*
са стека се скидају: I, F, C, A, *који чине компоненту*
стек:

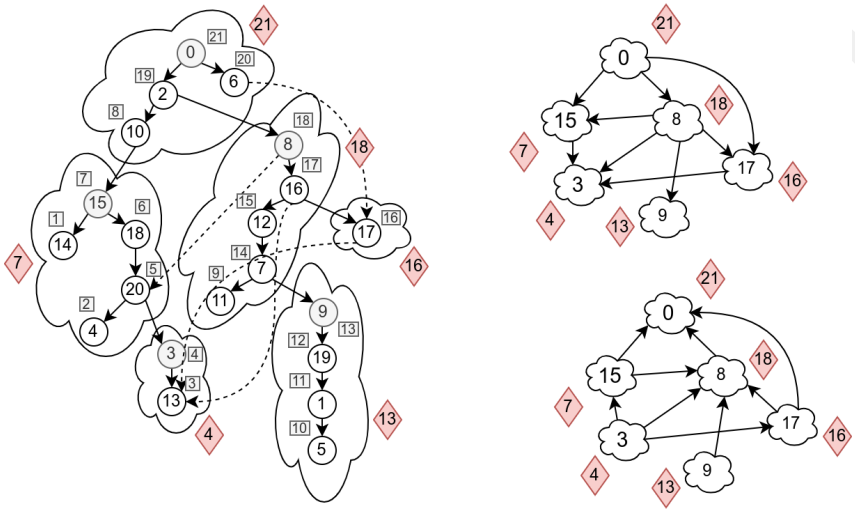
2.6.3 Косарајуов алгоритам

Још један алгоритам линеарне временске сложености за одређивање компоненти јаке повезаности у усмереном графу је Косарајуов алгоритам. Овај алгоритам је осмислио (међутим није публиковао) Косарају[9] 1978. године, а затим је до њега касније независно дошао и Шарир[10] 1981. године. Он користи два DFS обиласка, па је мало спорији од Тарјановог алгоритма, али је једноставнији за разумевање.

[9] Самбасива Рао Косарају (енгл. Sambasiva Rao Kosaraju), индијско-амерички информатичар.
[10] Миша Шарир (енгл. Micha Sharir), израелски математичар.

Приметимо да се приликом извршавања Тарјановог алгоритма базни чворови скидају са стека (заједно са осталим чворовима из њихових компонената) у растућем редоследу њихове одлазне нумерације. При томе, базни чвор има максимални редни број одлазне DFS нумерације од свих чворова своје компоненте. Обилазак базних чворова тече у обратном тополошком редоследу чворова кондензованог графа (графа добијеног тако што се свака компонента представи само њеним базним чвором), тј. обилазак тог ацикличког графа се врши од његових „листова" ка „корену".

Пример 2.6.9



Слика 2.55: На слици лево је уз сваки чвор приказан одлазни редни број у DFS обиласку графа са слике 2.44 (прегледности ради, гране унутар исте компоненте нису приказане), док је уз сваку компоненту приказан највећи редни број одлазне DFS нумерације чвора те компоненте (то је редни број базног чвора). Десно је приказан кондензовани граф оригиналног графа и кондензовани граф његовог транспонованог графа.

На слици 2.55 приказани су одлазни редни бројеви чворова у DFS обиласку графа са слике 2.44. Јасно се види да базни чворови имају највећи одлазни број од свих чворова у њиховој компоненти. Тарјанов алгоритам би редом проналазио компоненте чији су базни чворови 3, 15, 9, 17, 8, *и* 0 *(а њихови одлазни редни бројеви су редом* 4, 7, 13, 16, 18, 21*).*

Можемо приметити и да све гране у кондензованом графу воде од чворова са већим ка чворовима са мањим одлазним бројевима. Докажимо ово и формално. Означимо са

$v.Post$ одлазни редни број чвора v у неком DFS обиласку који по једном посећује све чворове графа, а са $C.Post$ максимални одлазни редни број свих чворова компоненте C, тј. $C.Post = \max_{v \in C} v.Post$.

Теорема 2.6.2

Нека су C и C' две различите компоненте јаке повезаности графа G и нека у кондензованом графу графа G постоји грана (C, C'). Тада важи $C.Post > C'.Post$.

Доказ. Разликујемо два случаја у односу на то коју од компоненти C и C' је алгоритам DFS прву посетио.

- Претпоставимо да DFS обилазак најпре посећује неки чвор v компоненте C и да у том тренутку ниједан други чвор из C и C' није посећен. Сви чворови компоненте C достижни су из чвора v. Додатно, с обзиром на то да у кондензованом графу графа G постоји грана (C, C'), из чвора v су достижни и сви чворови компоненте C'. То значи да ће DFS обилазак покренут из чвора v посетити све чворове u из $C \cup C'$, те ће они бити његови потомци у DFS дрвету. Одатле следи да је $v.Post > u.Post$ за свако $u \in C \cup C'$, $u \neq v$, одакле важи $C.Post > C'.Post$.
- Претпоставимо да DFS обилазак најпре посећује неки чвор v компоненте C' и да у том тренутку није посећен ниједан други чвор из C' и C. С обзиром на то да у кондензованом графу графа G постоји грана (C, C') и да је кондензовани граф ациклички, у кондензованом графу не сме постојати грана из неког чвора компоненте C' ка неком чвору компоненте C. Стога DFS обилазак покренут из чвора v не стиже ни до једног чвора компоненте C. Одавде следи да ће чворови компоненте C бити посећени касније, односно важи $C.Post > C'.Post$.

□

Из претходне теореме следи да ако бисмо DFS обилазак покренули из оног базног чвора који има најмањи одлазни број, сигурни бисмо били да би тај DFS обилазак обишао све чворове у његовој компоненти и ниједан други чвор (јер у кондензованом графу не постоји ниједна грана која би могла да „побегне” из те компоненте). Међутим, ми тај базни чвор не знамо и није нам лако да га одредимо. Дакле, ако бисмо вршили DFS обилазак од „листова” ка „корену” кондензованог графа, добили бисмо тачно једну по једну компоненту повезаности, међутим, проблем је што ми не знамо чворове који припадају компонентама које су „листови” у кондензованом графу и није лако одредити их. Са друге стране, сигурни смо да чвор са највећим одлазним редним бројем припада „корену”, тј. компоненти кондензованог графа из које не излази ни једна грана. Ако бисмо покренули алгоритам DFS из тог чвора, набројали бисмо и чворове ван те компоненте, међутим, не и ако *графу обрнемо гране*!

Размотримо транспоновани граф G^T графа G, добијен променом усмерења свих грана у графу: он има исте компоненте јаке повезаности као и граф G. Другим речима, два чвора су узајамно достижна у полазном графу ако и само ако су узајамно достижна у њему транспонованом графу. Зато, ако покренемо DFS обилазак из било ког чвора у транспонованом графу, сасвим смо сигурни да ће у том обиласку бити достигнути сви чворови из компоненте јаке повезаности којој припада полазни чвор (а можда и неки други). Приметимо да ће кондензовани графови графова G и G^T бити међусобно транспоновани (слика 2.55, десно). Другим речима, у транспонованом кондензованом графу неће постојати грана из корене компоненте ка другим компонентама. Стога је за одређивање корене компоненте која садржи неки чвор v, довољно покренути алгоритам DFS из чвора v у графу G^T. На овај начин се обилазе сви чворови компоненте чвора v и ништа више од тога. Алгоритам се даље наставља по истом принципу. Можемо из графа уклонити све ове чворове (заправо, означити да су посећени), пронаћи чвор са највећом вредношћу одлазне нумерације у остатку графа, покренути нови DFS обилазак у графу G^T и тако даље.

Одавде директно следи Косарацуов алгоритам, који има две фазе:

- у првој фази се покреће одговарајући DFS обилазак графа G, све док се не посете сви чворови графа G (обилазак покрећемо редом из сваког чвора, при чему већ посећене чворове прескачемо); притом се чворови графа сортирају растуће према вредности одлазне нумерације чворова;
- у другој фази се конструише транспоновани граф G^T графа G и покреће се низ DFS (или BFS) обилазака у редоследу добијеном у претходном кораку, односно у опадајућем редоследу одлазне нумерације. Сваки скуп чворова, који је достижан у наредном обиласку даје нову компоненту јаке повезаности графа G.

Описани алгоритам се састоји у покретању два обиласка графа, па му је временска сложеност $O(|V| + |E|)$.

Алгоритам се у језику C++ може имплементирати на следећи начин.

```
void odrediOdlazniRedosled(int cvor,
                           vector<bool>& posecen,
                           vector<int>& odlazniRedosled) {
  posecen[cvor] = true;
  for (int sused : listaSuseda[cvor])
    if (!posecen[sused])
      odrediOdlazniRedosled(sused, posecen, odlazniRedosled);
  odlazniRedosled.push_back(cvor);
}
```

```
// za svaki cvor odredjuje se lista suseda
// iz kojih grane vode do tog cvora
vector<vector<int>> odrediTransponovaniGraf() {
  vector<vector<int>> listaDolaznihSuseda;
  int n = listaSuseda.size();
  listaDolaznihSuseda.resize(n);
  for (int cvor = 0; cvor < n; cvor++)
     for (int sused : listaSuseda[cvor])
        listaDolaznihSuseda[sused].push_back(cvor);
  return listaDolaznihSuseda;
}

// dfs obilazkom iz datog cvora svim cvorovima dostiznim iz njega
// u transponovanom grafu se dodeljuje dati broj komponente
void odrediKomponentu(int cvor, vector<int>& komponente, int komponenta,
                      const vector<vector<int>>& listaDolaznihSuseda) {
  komponente[cvor] = komponenta;
  for (int sused : listaDolaznihSuseda[cvor])
      if (komponente[sused] == -1)
         odrediKomponentu(sused, komponente, komponenta,
                          listaDolaznihSuseda);
}

// za svaki cvor se odredjuje redni broj jake povezanosti
vector<int> kosarajuSCC() {
  // vrsimo DFS i cvorove redjamo u niz u redosledu odlazne numeracije
  int brojCvorova = listaSuseda.size();
  vector<bool> posecen(brojCvorova, false);
  vector<int> odlazniRedosled;
  for (int cvor = 0; cvor < brojCvorova; cvor++)
    if (!posecen[cvor])
      odrediOdlazniRedosled(cvor, posecen, odlazniRedosled);

  // odredjujemo transponovani graf
  vector<vector<int>> listaDolaznihSuseda = odrediTransponovaniGraf();

  // redni broj komponente svakog cvora
  vector<int> komponente(brojCvorova, -1);
```

```
  // redni broj tekuce komponente
  int komponenta = 0;
  // cvorove obilazimo u opadajucem redosledu odlazne numeracije
  for (int i = brojCvorova - 1; i >= 0; i--) {
    int cvor = odlazniRedosled[i];
    // ako cvoru jos nije dodeljena komponenta
    if (komponente[cvor] == -1)
      // dodeljujemo komponentu njemu i svim cvorovima dostiznim
      // iz njega u transponovanom grafu
      odrediKomponentu(cvor, komponente, komponenta++,
                       listaDolaznihSuseda);
  }
  return komponente;
}
```

Пример 2.6.10

У примеру на слици 2.55 покретањем DFS обиласка из чвора 0 *добијају се одлазни редни бројеви који су приказани на слици.*

- *Покренемо у транспонованом графу DFS обилазак из чвора са највећим одлазним редним бројем. То је чвор* 0 *чији је одлазни редни број* 21. *DFS обилазак редом набраја чворове* 0, 2, 10 *и* 6.
- *Чворови са одлазним бројевима* 20 *и* 19 *су већ посећени, па наредни DFS обилазак покрећемо из чвора са одлазним редним бројем* 18. *То је чвор* 8 *и DFS обилазак набраја чворове* 8, 16, 12, 7 *и* 11.
- *Чвор са одлазним редним бројем* 17 *је већ посећен, па наредни DFS обилазак покрећемо из чвора са одлазним редним бројем* 16. *То је чвор* 17 *и DFS обилазак из њега набраја само њега.*
- *Чворови са одлазним бројевима* 15 *и* 14 *су већ посећени, па наредни DFS обилазак покрећемо из чвора са одлазним редним бројем* 13. *То је чвор* 9 *и DFS обилазак набраја чворове* 9, 19, 1 *и* 5.
- *Чворови са одлазним бројевима* 12, 11, 10, 9 *и* 8 *су већ посећени, па наредни DFS обилазак покрећемо из чвора са одлазним редним бројем* 7. *То је чвор* 15 *и DFS обилазак набраја чворове* 15, 14, 18, 20 *и* 4.
- *Чворови са одлазним бројевима* 6 *и* 5 *су већ посећени, па наредни DFS обилазак покрећемо из чвора са одлазним редним бројем* 4. *То је чвор* 3 *и DFS обилазак набраја чворове* 3 *и* 13.
- *Чворови са одлазним бројевима* 3, 2 *и* 1 *су већ посећени, па се алгоритам завршава.*

Приметимо да се у првом кораку алгоритма чворови уређују у обрнутом тополошком

редоследу графа G. Додатно, алгоритам генерише компоненте јаке повезаности у опадајућем редоследу одлазне нумерације, односно чворови кондензованог графа се добијају у тополошком редоследу.

Задатак: Докажи све формуле!

Скуп логичких формула се састоји од одређеног броја исказних слова, и одређеног броја импликација (између тих исказних слова). За импликације сматрамо да су унапред доказане, док елементарни искази морају бити доказани или директно или применом правила *modus ponens*, којим се из раније доказаног изказа A, на основу импликације $A \Rightarrow B$ доказује исказ B.

$$\frac{A \qquad A \Rightarrow B}{B}$$

Потребно је одредити најмањи могући број исказа које је потребно доказати директно, тако да се затим из њих и датих импликација могу применом правила *modus ponens* доказати сви остали искази.

Опис улаза

Са стандардног улаза се уноси број елементарних исказа m ($1 \leq m \leq 10^4$), а затим број импликација n($1 \leq n \leq 10^4$). Након тога се уноси низ парова бројева i и j ($0 \leq i < m$), који представљају импликације $p_i \Rightarrow p_j$.

Опис излаза

На станадардни излаз исписати најмањи број исказа које треба доказати.

Пример

Улаз	*Излаз*	*Објашњење*

`6`
`7`
`0 1`
`1 3`
`1 2`
`3 0`
`2 4`
`4 2`
`5 2`
	`2`	Довољно је, на пример, доказати формуле 0 и 5. Није довољно доказати само једну формулу.

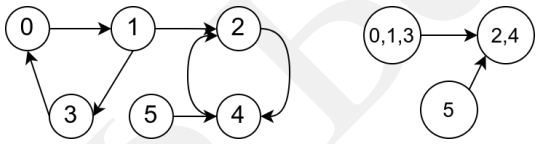
Решење

Елементарне исказе можемо представити чворовима, а импликације гранама графа. Сви чворови достижни из неког чвора који одговара доказаном тврђењу одговарају

тврђењима која могу бити доказана применом импликација.

Једноставан случај је када у графу нема циклуса, тј. када је у питању усмерени ациклички граф. Потребно је и довољно доказати само она тврђења која се не јављају на десној страни ниједне импликације, тј. само она тврђења којима одговарају чворови графа улазног степена 0. Заиста, та тврђења не могу бити доказана применом импликација (јер ниједна импликација не води до њих), док сва остала тврђења могу бити доказана применом импликација (пошто нема циклуса, пратећи гране уназад, доћи ће се до неког чвора чији је улазни степен 0).

Случај графа који садржи циклусе је донекле компликованији. Приметимо да за сваку компоненту јаке повезаности графа важи да је довољно доказати било који њен чвор и тада ће сигурно бити могуће доказати и све остале њене чворове (јер су сви чворови у компоненти узајамно достижни). Зато је могуће извршити редукцију графа, тј. направити кондензовани граф у ком је свака компонента повезаности представљена појединачним чвором. Кондензован граф је увек ациклички и на њега можемо применити решење за ациклички граф. На наредној слици је приказан граф из примера овог задатка, као и његов кондензовани граф, који за чворове има три компоненте јаке повезаности полазног графа.



Илустрације ради, у наредној имплементацији експлицитно градимо кондензовани граф (додуше, за решење овог задатка то није неопходно, већ је довољно само пребројати улазне степене његових чворова).

```
// graf implikacija
vector<vector<int>> susedi;

// redni broj trenutne komponente jake povezanosti
int brojKomponenata = 0;
// preslikavanje čvorova u redne brojeve njihovih komponenata
vector<int> komponente;
// liste suseda u kondenzovanom grafu u kome su čvorovi komponente
// jake povezanosti polaznog grafa
vector<vector<int>> susedneKomponente;

// određuju se komponente jake povezanosti i gradi se kondenzovani
```

```
// graf liste povezanosti ovog grafa se smeštaju u globalnu
// promenljivu susedneKomponente
void napraviKondenzovaniGraf() {
  // određuju se komponente jake povezanosti
  // broj komponenata je sadržan globalnoj promenljivoj brojKomponenata
  // komponenta svakog čvora je određena globalnim nizom komponente
  odrediKomponente();
  // broj čvorova kondenzovanog grafa je broj komponenata jake povezanosti
  susedneKomponente.resize(brojKomponenata);
  // grane koje su već dodate u kondenzovani graf
  set<pair<int, int>> dodateGrane;
  // prolazimo kroz sve grane originalnog grafa
  for (int cvor = 0; cvor < susedi.size(); cvor++)
    for (int sused : susedi[cvor]) {
      // u kondenzovani graf dodajemo granu između komponente čvora i
      // komponente suseda (ako nije već dodata)
      if (komponente[cvor] != komponente[sused] &&
          dodateGrane.count({komponente[cvor], komponente[sused]}) == 0) {
        susedneKomponente[komponente[cvor]].push_back(komponente[sused]);
        dodateGrane.emplace(komponente[cvor], komponente[sused]);
      }
    }
}

int main() {
  // ucitavamo podatke o iskazima i implikacijama
  // ...

  // pravimo kondenzovani graf u kome su cvorovi komponente povezanosti
  napraviKondenzovaniGraf();

  // određujemo ulazne stepene čvorova kondenzovanog grafa
  vector<int> ulazniStepen(susedneKomponente.size(), 0);
  for (int cvor = 0; cvor < susedneKomponente.size(); cvor++)
    for (int sused : susedneKomponente[cvor])
      ulazniStepen[sused]++;

  // brojimo čvorove čiji je ulazni stepen nula
  int broj = 0;
```

```
    for (int cvor = 0; cvor < susedneKomponente.size(); cvor++)
      if (ulazniStepen[cvor] == 0)
        broj++;

    cout << broj << endl;

    return 0;
}
```

2.7 Ојлерови и Хамилтонови путеви

У неким графовским проблемима потребно је пронаћи пут између два чвора који посећује сваку грану графа тачно једном или, дуално, пут који посећује сваки чвор графа тачно једном. У овом поглављу бавићемо се проблемима испитивања да ли у графу постоје овакве врсте путева. Иако ова два проблема на први поглед наликују, показаће се да се технике за њихово решавање значајно разликују.

2.7.1 Ојлерови путеви и циклуси

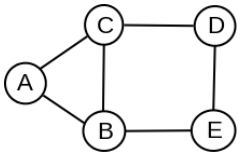
Размотримо наредни проблем: поштар треба да достави одређен број писама на различите адресе у неколико суседних улица. Поштар гледа у мапу и уочава на који начин су те улице међусобно повезане. Црта поједностављену мапу и притом сваку од раскрсница означава неким бројем. Поштар жели да достави сва писма, а да притом сваким путем прође тачно једном. Поставља се питање да ли је то могуће урадити. Овај проблем могуће је моделовати у терминима графова: свака раскрсница представља чвор графа, а пут који води од једне раскрснице до друге грану графа. Дати проблем се онда своди на питање да ли у графу постоји пут који садржи сваку грану графа тачно једном.

Ојлеров[11] *пут* (енгл. Eulerian trail, Eulerian path) је пут у графу који пролази кроз сваку грану графа тачно једном (при чему неке чворове може посетити и више пута). *Ојлеров циклус* (енгл. Eulerian circuit, Eulerian cycle) је Ојлеров пут чији се почетни и крајњи чвор поклапају.

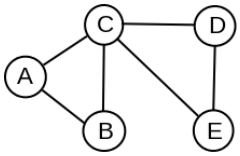
Пример 2.7.1

У неусмереном графу приказаном на слици 2.56 постоји Ојлеров пут (C, D, E, B, C, A, B)*, али не постоји Ојлеров циклус. У графу приказаном на слици 2.57 постоји Ојлеров циклус* (C, D, E, C, A, B, C)*.*

[11] Леонард Ојлер (нем. Leonhard Euler), (1707-1783), швајцарски математичар.



Слика 2.56: Неусмерен граф који садржи Ојлеров пут (на пример (C, D, E, B, C, A, B)), али не садржи Ојлеров циклус.



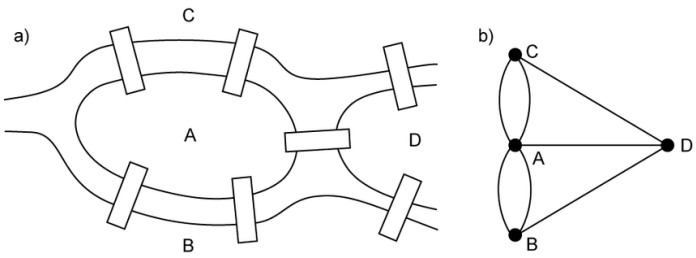
Слика 2.57: Неусмерен граф који садржи Ојлеров циклус. Један Ојлеров циклус у овом графу је (C, D, E, C, A, B, C).

У наставку ћемо се бавити одређивањем Ојлерових путева и циклуса.

Проблем

За дати граф (било усмерен, било неусмерен) утврдити да ли има Ојлеров пут или Ојлеров циклус и ако има одредити га.

Појам Ојлерових графова у вези је са, како се сматра, првим решеним проблемом теорије графова. Швајцарски математичар Леонард Ојлер наишао је 1736. године на следећи задатак. Град Кенигсберг, данас Калињинград, лежи на обалама и на два острва на реци Прегел, као што је приказано на слици 2.58, лево. Град је повезан путем седам мостова. Питање које је мучило многе тадашње грађане Кенигсберга било је да ли је могуће почети шетњу из било које тачке у граду и вратити се у полазну тачку, прелазећи при томе сваки мост тачно једном. Овај проблем се може формулисати као следећи проблем из теорије графова: да ли је могуће у неусмереном повезаном графу пронаћи циклус, који сваку грану графа садржи тачно једном, тј. Ојлеров циклус. Другим речима, да ли је могуће нацртати граф са слике 2.58, десно, не дижући оловку са папира, тако да оловка свој пут заврши на месту са кога је и кренула. Напоменимо да овај граф има вишеструке гране између парова чворова, па строго гледано по дефиницији није граф, већ мултиграф. Ојлер је доказао да је овакав обилазак могућ ако и само ако је граф повезан и сви његови чворови имају паран степен (теорема 2.7.1 у наставку). Графови који садрже Ојлеров циклус зову се *Ојлерови графови*. Пошто „граф” на слици 2.58, десно, има чворове непарног степена, закључујемо да проблем Кенигсбершких мостова нема решење.



Слика 2.58: Проблем Кенигсбершких мостова, и одговарајући мултиграф.

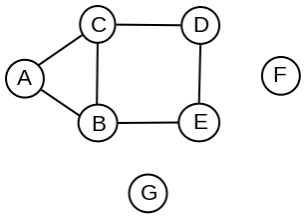
Пре него што докажемо карактеризацију Ојлерових графова преко степена чворова, докажимо једну лему.

Лема 2.7.1 **[Циклус у Ојлеровом неусмереном графу]**

Ако у неусмереном графу G сваки чвор има паран степен, онда у том графу постоји циклус.

Доказ. Претпоставимо да смо започели обилазак графа из произвољног чвора u произвољним редоследом. Сигурно је да ћемо се током обиласка вратити у чвор u, јер кад год уђемо у неки чвор, смањујемо његов степен за један, чинимо га непарним, па га увек можемо и напустити. Наравно, овакав обилазак не мора да садржи све гране графа. Дакле, у сваком графу у ком је степен свих чворова паран, за сваки чвор u мора да постоји циклус који почиње и завршава се у u. □

Приметимо да Ојлеров пут и Ојлеров циклус могу постојати и у неповезаном графу ако граф поред једне повезане компоненте садржи само изоловане чворове (слика 2.59).



Слика 2.59: Неповезани граф који садржи Ојлеров пут (на пример (C, D, E, B, C, A, B)).

Теорема 2.7.1 **[Карактеризација Ојлерових неусмерених графова]**

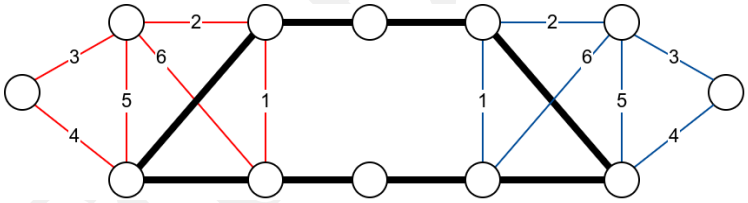
У неусмереном графу $G = (V, E)$ постоји Ојлеров циклус ако и само у графу G сви

чворови имају парни степен, и сви чворови степена различитог од нула припадају истој компоненти повезаности.

Доказ. Лако је показати да ако у графу постоји Ојлеров циклус, онда сви чворови графа морају имати паран степен. Наиме, за време обиласка циклуса, у сваки чвор се улази исто толико пута колико пута се из њега излази. Пошто се свака грана пролази тачно једном, број грана суседних произвољном чвору мора бити паран.

Да бисмо индукцијом доказали да је овај услов и довољан да би граф имао Ојлеров циклус, морамо најпре да изаберемо параметар по коме ће бити изведена индукција. Тај избор треба да омогући смањивање проблема, без његове промене. Ако уклонимо чвор или грану из графа, степени чворова у добијеном графу нису више сви парни. Треба да уклонимо такав скуп грана S, да за сваки чвор u графа G број грана из скупа S суседних са u остане паран (макар и 0). Произвољан циклус задовољава овај услов, а на основу леме 2.7.1 сваки граф чији су сви чворови парног степена садржи неки циклус. Сада можемо да формулишемо индуктивну хипотезу и докажемо теорему.

Индуктивна хипотеза. Повезани неусмерени граф са мање од m грана чији сви чворови имају паран степен садржи Ојлеров циклус, који се може ефективно пронаћи.



Слика 2.60: Пример конструкције Ојлеровог циклуса индукцијом. Пуном линијом извучене су гране помоћног циклуса. Избацивањем грана овог циклуса из графа, добија се граф са четири компоненте повезаности, при чему се две компоненте састоје од изолованих чворова.

Посматрајмо граф $G = (V, E)$ који садржи m грана у ком сви чворови имају паран степен. Нека је P неки циклус у графу G (који постоји, на основу леме 2.7.1), и нека је G' граф добијен уклањањем грана циклуса P из графа G. Степени свих чворова у графу G' су парни, јер је број уклоњених грана суседних било ком чвору паран. Ипак се индуктивна хипотеза не може применити на граф G', јер он не мора бити повезан (слика 2.60). Нека су $G'_1, G'_2, \ldots, G'_k$ компоненте повезаности графа G'. У свакој компоненти повезаности степени свих чворова су парни. Поред тога, број грана у свакој компоненти је мањи од m јер је укупан број грана у свим компонентама мањи од m. Према томе, индуктивна хипотеза се може применити на сваку од компоненти посебно:

у свакој компоненти G'_i постоји Ојлеров циклус P'_i, и ми знамо да га пронађемо. Потребно је сада све ове циклусе објединити са помоћним циклусом P у један Ојлеров циклус за граф G. Полазимо из произвољног чвора циклуса P („магистралног пута") све док не дођемо до неког чвора v_j који припада некој компоненти G'_j. Тада обилазимо компоненту G'_j циклусом P'_j („локалним путем") и враћамо се у чвор v_j. Настављамо обилазак циклуса P на тај начин, обилазећи циклусе компоненти у тренутку наиласка на њих. На крају обиласка ћемо се вратити у полазни чвор. У том тренутку све гране графа G смо прошли тачно једном, што значи да је конструисан Ојлеров циклус.

Овај доказ сугерише ефикасан рекурзивни алгоритам за конструкцију Ојлеровог циклуса у графу. □

Видели смо да неусмерени граф може имати Ојлеров пут, а да истовремено нема Ојлеров циклус (слика 2.56). Наредна лема (која се доказује веома једноставно) даје потребан и довољан услов да неусмерен граф има Ојлеров пут.

Теорема 2.7.2 **[Постојање Ојлеровог пута и циклуса у неусмереном графу]**

Неусмерени граф има Ојлеров пут ако и само ако сви чворови степена различитог од нула припадају једној компоненти повезаности и важи један од наредних услова:

- *степен сваког чвора је паран или*
- *степен тачно два чвора је непаран, а осталих чворова је паран.*

Ако важи прва претпоставка онда је сваки Ојлеров пут истовремено и Ојлеров циклус. Уколико важи друга претпоставка, чворови непарног степена су почетни и крајњи чвор Ојлеровог пута и у оваквом графу не постоји Ојлеров циклус.

У графу са слике 2.56 чворови B и C су непарног степена, док су остали чворови парног степена, те чворови B и C представљају почетни и крајњи чвор Ојлеровог пута. Додатно, у овом графу не постоји Ојлеров циклус.

Ситуација је слична и у усмереним графовима. Наредна лема даје потребан и довољан услов да у усмереном графу постоји Ојлеров пут.

Теорема 2.7.3 **[Постојање Ојлеровог пута и циклуса у усмереном графу]**

У усмереном графу постоји Ојлеров пут ако и само ако сви чворови који нису изоловани припадају истој слабо повезаној компоненти повезаности (истој компоненти повезаности одговарајућег неусмереног графа) и важи један од наредних услова:

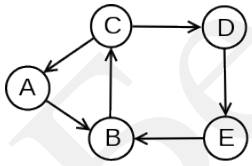
- *улазни и излазни степени свих чворова су међусобно једнаки или*

- *улазни степен већи је за један од излазног степена тачно за један чвор, излазни степен већи је за један од улазног степена тачно за један чвор, док су за све остале чворове улазни и излазни степен међусобно једнаки.*

У првом случају, сваки Ојлеров пут је и Ојлеров циклус, а у другом случају Ојлеров пут почиње у чвору чији је излазни степен већи за један, а завршава се у чвору чији је улазни степен већи за један.

Пример 2.7.2

У графу приказаном на слици 2.61 су улазни и излазни степени чворова A, E и D међусобно једнаки, улазни степен чвора B је већи за један од излазног, док је улазни степен чвора C за један мањи од излазног степена. У овом графу постоји Ојлеров пут који почиње у чвору C, а завршава се у чвору B (нпр. (C, D, E, B, C, A, B)), док Ојлеров циклус не постоји.



Слика 2.61: Усмерени граф који садржи Ојлеров пут (C, A, B, C, D, E, B), али не садржи Ојлеров циклус.

2.7.1.1 Хирхолцеров алгоритам

Иако се из доказа теореме 2.7.1 може издвојити идеја за конструкцију Ојлеровог циклуса у повезаном неусмереном графу код кога су степени свих чворова парни, потребно је прецизирати детаље имплементације да би се уместо рекурзивне добила ефикасна итеративна имплементација. Кренимо од основне идеје алгоритма, која се може применити и на неусмерене и на усмерене графове. Једноставности ради, претпоставићемо да граф има јединствену компоненту (слабе) повезаности.

Полази се из произвољног чвора u у таквом графу. У сваком кораку прелази се произвољном непосећеном граном до неког суседа тренутног чвора. Овај корак се понавља све док се не вратимо у полазни чвор u. У њега се морамо вратити у неком моменту, јер је степен сваког чвора паран. На овај начин конструише се неки циклус. Уколико он садржи све гране графа, Ојлеров циклус је конструисан. Иначе се тај циклус проширује на следећи начин: полазећи из чвора u дуж циклуса проналази се први чвор v који припада текућем циклусу и који има грану која није укључена у циклус. Том граном се креће из чвора v и открива се нови циклус (он се враћа у чвор v), који се

састоји искључиво од грана које још увек нису у претходном циклусу. Тај нови циклус додајемо у текући циклус, конструишући од два циклуса нови, дужи циклус. Поступак се наставља док пронађени циклус не обухвати све гране графа.

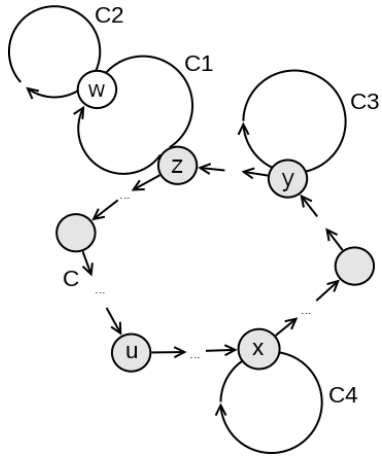
Један ефикасан алгоритам заснован на претходној идеји је *Хирхолцеров*[12] *алгоритам*. Граф се обилази кренувши од било ког чвора u, све док се у неком тренутку не вратимо у чвор u и тако пронађемо неки циклус. Након тога се редом обилазе чворови тог циклуса у потрази за чвором из ког постоје гране које још нису посећене. У зависности од структуре података која се користи у имплементацији, овај обилазак може бити унапред, у редоследу откривања чворова тог циклуса или уназад, у обратном редоследу. Претпоставимо да се обилазак врши уназад, тј. да се из чвора u враћамо уназад гранама откривеног циклуса. Гране (или, алтернативно, њихове крајње чворове) које обрадимо током овог повратка уназад додајемо у резултујући Ојлеров циклус. Ако је граф усмерен, треба обратити пажњу на редослед грана (или чворова) у конструисаном Ојлеровом циклусу. Ако циклус чувамо у низу и гране (или чворове) додајемо редом на крај низа, на крају алгоритма је тај низ потребно обрнути. Алтернативно, гране (или чворове) уместо у низу можемо чувати у некој структури података која допушта ефикасно додавање на почетак (на пример у листи или реду са два краја). Приликом повратка кроз циклус, у сваком кораку проверавамо да ли из текућег чвора (рецимо да је то чвор v) постоје гране које још нисмо обишли. Ако постоје, покрећемо нови обилазак графа из чвора v и тако налазимо нови циклус који се састоји од до тада непосећених грана. Ако се такав циклус пронађе, онда (привремено) прелазимо на његову интеграцију у резултујући Ојлеров циклус, тако што сада пролазимо кроз његове гране (уназад), али поново проверавајући да ли из неког од његових чворова постоје нови циклуси које треба на исти начин обићи и интегрисати у резултујући циклус. Након што се вратимо у чвор v и обрадимо све гране које из њега полазе, настављамо интеграцију иницијалног циклуса, враћајући се уназад грану по грану према чвору u.

Пример 2.7.3

Размотримо граф са слике 2.62.

- *Обилазак креће из чвора u и иницијално се конструише циклус C чији су чворови означени сивом бојом.*
- *Уназад се враћамо чворовима циклуса C почев од чвора u и гране редом додајемо у Ојлеров циклус (који чувамо у обратном редоследу). Када стигнемо до чвора z, примећујемо да из њега постоје непосећене гране, па обилазимо циклус C_1. Претпоставимо, на пример, да када стигнемо до чвора w настављамо са обиласком цилуса C_1 назад ка чвору z уместо да пређемо на обилазак циклуса C_2. Пошто су*

[12] Карл Хирхолцер (нем. Carl Hierholzler), (1840-1871), немачки математичар.



Слика 2.62: Обједињавање циклуса у нови циклус.

када се вратимо у чвор z све гране из чвора z посећене, враћамо се назад дуж циклуса C_1 и његове гране интегришемо у Ојлеров циклус. Међутим, када стигнемо до чвора w примећујемо да из њега постоје непосећене гране и тада обилазимо циклус C_2 и враћамо се у чвор w. Тада су посећене све гране из чвора w, па се враћамо назад гранама циклуса C_2, интегришући их у Ојлеров циклус, а затим настављамо да се враћамо назад гранама циклуса C_1 од чвора w до чвора z, интегришући тако и те гране у Ојлеров циклус.

- *Настављамо да се враћамо назад гранама циклуса C интегришући их у Ојлеров циклус. Када стигнемо до чвора y обилазимо циклус C_3 и враћамо се назад у чвор y. Пошто су тада све гране из чвора y посећене, враћамо се назад дуж циклуса C_3 и на тај начин он се интегрише у Ојлеров циклус.*
- *Настављамо да се враћамо назад гранама циклуса C интегришући их у Ојлеров циклус. Када стигнемо до чвора x обилазимо циклус C_4 и враћамо се назад у чвор x. Пошто су тада све гране из чвора x посећене, враћамо се назад гранама циклуса C_4 и он се на тај начин интегрише у Ојлеров циклус.*
- *На крају се враћамо уназад преосталим гранама циклуса C од чвора x до чвора u интегришући и те гране у Ојлеров циклус.*

Пошто смо гране додавали уназад, коначан Ојлеров циклус добијамо обртањем редоследа додатих грана.

Приметимо да циклус C има улогу главног пута, а циклуси C_1, C_3 и C_4 имају улогу споредних путева у односу на њега. Међутим, у тренутку док интегришемо циклус C_1

у резултат, он има улогу главног пута, а циклус C_2 има улогу споредног пута.

Приметимо и да циклусе C_1 и C_2 сматрамо одвојеним циклусима, само зато што смо током обиласка графа унапред у тренутку када смо дошли у чвор w одабрали да се вратимо назад гранама циклуса C_1 до чвора z, уместо да кренемо у обилазак циклуса C_2. Да смо приликом из чвора w направили обилазак циклуса C_2 пре него што се гранама циклуса C_1 вратимо у чвор z, тада би деловало да из чвора z постоји само један споредни циклус (додуше самопресецајући). Обиласком његових грана уназад добио би се потпуно исти крајњи резултат, тј. Ојлеров циклус.

Примећујемо да алгоритам има рекурзивну структуру (са главног циклуса прелазимо на споредни, са којег можемо прећи на нови споредни циклус и тако даље), па и имплементација може бити рекурзивна. Модификујемо рекурзивни алгоритам обиласка у дубину тако да пролази свим раније непосећеним излазним гранама из текућег чвора. Када се заврши рекурзвни обилазак свих суседа, текући чвор се додаје на почетак Ојлеровог циклуса који се гради. Приметимо да се, за разлику од класичног обиласка у дубину, не води рачуна о томе који су чворови раније посећени, па се рекурзивна обрада истог чвора може покренути и неколико пута. У главној функцији прво израчунавамо излазне степене свих чворова (да би Ојлеров циклус сигурно постојао, претпостављамо да су им улазни степени једнаки). Када прођемо неком граном из текућег чвора, избацићемо је из даљег разматрања. Ако је излазни степен у неком тренутку једнак k, сматраћемо да је непосећено првих k грана у листи суседа тог чвора. Посећиваћемо увек последњу од њих и избацивaћемо је из даљег разматрања смањујући излазни степен.

```
// rezultujuci Ojlerov ciklus
deque<int> ojlerovCiklus;
// izlazni stepeni cvorova racunajuci samo
vector<int> izlazniStepen;

void posetiCvor(int v) {
   // dok ima jos neposecenih grana
   while (izlazniStepen[v] > 0)
       // posecujemo suseda na kraju neposecene grane i
       // izbacujemo tu granu iz daljeg razmatranja
       posetiCvor(listaSuseda[v][--izlazniStepen[v]]);
   // sve grane su obradjene, pa dodajemo cvor na pocetak
   // Ojlerovog ciklusa
   ojlerovCiklus.push_front(v);
}
```

```
void pronadjiOjlerovCiklus() {
   // izracunavamo pocetne izlazne stepene svakog cvora
   int brojCvorova = listaSuseda.size();
   izlazniStepen.resize(brojCvorova);
   for (int v = 0; v < brojCvorova; i++)
       izlazniStepen[v] = listaSuseda[v].size();
   // pokrecemo obilazak iz proizvoljnog cvora
   posetiCvor(0);
}
```

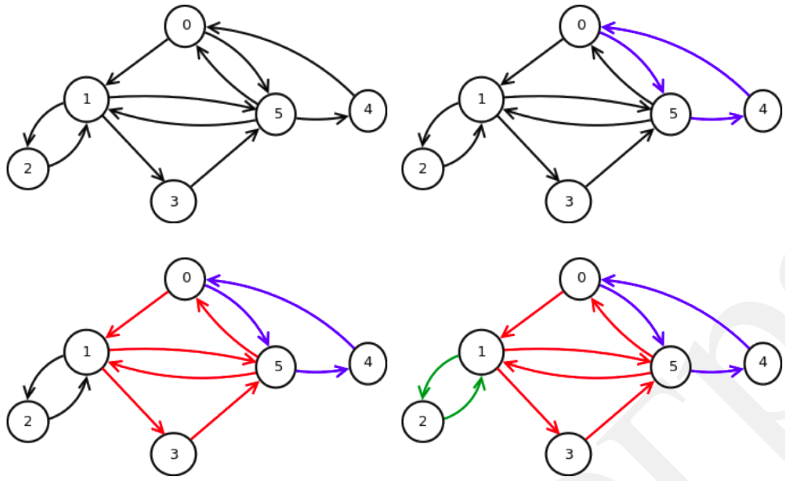
Ако не желимо рекурзивну имплементацију, током откривања циклуса ћемо гране тј. њихове крајње чворове стављати на *стек*. Приликом обраде циклуса тј. његове интеграције у крајњи резултат, чворове ћемо скидати са стека и то је разлог зашто се пронађени циклус у другој фази обилази уназад (алтернативно, чворове бисмо могли додавати у ред и тада би би се циклус и у другој фази обилазио унапред). Приликом скидања сваког чвора са стека проверавамо да ли су све гране које полазе из њега посећене. Ако јесу, завршили смо обраду тог чвора и њега додајемо у низ чворова који одређује Ојлеров циклус. У супротном, ако нису све гране које полазе из текућег чвора посећене, из тог чвора крећемо обилазак новог циклуса додајући чворове тог циклуса на врх стека. На тај начин је обезбеђено да ће тај нови циклус бити обрађен и интегрисан у резултат пре него што се настави са обрадом полазног циклуса. Када тај циклус буде обрађен и интегрисан у резултат, његови чворови ће бити скинути са стека и тада ћемо наставити са обиласком почетног циклуса (уназад). Приметимо да ни у једном тренутку не морамо експлицитно да проверавамо да ли смо се вратили у почетни чвор текућег циклуса, нити да проверавамо да ли се тренутно врши прва фаза (обилазак унапред којим се открива нови циклус) или друга фаза (обилазак уназад којим се чворови интегришу у резултат). Ако текући чвор има непосећене суседе, стављамо га на стек и прелазимо на неког његовог непосећеног суседа, а ако нема, скидамо га са стека и додајемо у резултујући циклус. Када стек постане празан, то значи да смо за све чворове размотрили све гране које полазе из њих, тј. да су све гране обрађене и да је Ојлеров циклус конструисан.

Ако граф G садржи само Ојлеров пут, а не и Ојлеров циклус, алгоритам започиње свој рад из чвора чији је излазни степен за један већи од улазног. Алтернативно, у неусмерен граф G се може додати грана којом се постиже услов да граф има Ојлеров циклус и затим искористити претходни алгоритам. Након проналаска Ојлеровог циклуса у допуњеном графу, та грана се уклања из циклуса и добија се Ојлеров пут.

У наставку је приказана нерекурзивна имплементација Хирхолцеровог алгоритма који проналази Ојлеров циклус у усмереном графу. Једноставности ради ћемо претпоставити да је граф повезан, па је обилазак довољно покренути једном, из произвољног

чвора.

```
vector<int> pronadjiOjlerovCiklus() {
  // rezultujuci ciklus, odredjen cvorovima kroz koje se prolazi
  vector<int> ojlerovCiklus;

  // stek na koji stavljamo cvorove u redosledu obilaska
  stack<int> tekuciPut;

  // na stek dodajemo polazni cvor
  tekuciPut.push(0);

  // sve dok postoji neki cvor na tekucem putu
  while (!tekuciPut.empty()) {
    int tekuciCvor = tekuciPut.top();

    // ako iz tekuceg cvora postoji jos neka grana koju nismo posetili
    if (izlazniStepen[tekuciCvor] > 0) {
      // pronalazimo cvor do koga postoji grana iz tekuceg cvora
      int naredniCvor = listaSuseda[tekuciCvor][--izlazniStepen[tekuciCvor]];
      // napredujemo ka narednom cvoru
      tekuciCvor = naredniCvor;
      tekuciPut.push(naredniCvor);
    }
    // ako iz tog cvora ne postoji neposecena grana
    else {
      // cvor dodajemo u Ojlerov ciklus
      ojlerovCiklus.push_back(tekuciCvor);
      // vracamo se unazad da bismo nasli preostale cikluse
      tekuciPut.pop();
    }
  }
  // obrcemo cvorove u ciklusu
  reverse(begin(ojlerovCiklus), end(ojlerovCiklus));

  return ojlerovCiklus;
}
```



Слика 2.63: Илустрација извршавања Хирхолцеровог алгоритма у случају датог усмереног графа.

Пример 2.7.4

На слици 2.63 илустровано је извршавање Хирхолцеровог алгоритма на датом усмереном Ојлеровом графу. Алгоритам стартује из чвора 0 *и проналази, рецимо, циклус* $(0, 5, 4, 0)$*. Гране овог циклуса се током обиласка уклањају из графа. Приликом повратка у чвор* 0 *проверава се да ли постоји нека непосећена грана у графу која полази из чвора* 0 *и пошто постоји грана* $(0, 1)$ *настављамо граном ка чвору* 1 *и проналазимо нови циклус* $(0, 1, 5, 1, 3, 5, 0)$*. Гране и овог циклуса се током обиласка уклањају из графа. Пут којим смо се до сад кретали садржи редом чворове* $(0, 5, 4, 0, 1, 5, 1, 3, 5, 0)$*. У овом тренутку констатујемо да из чвора* 0 *не постоји ниједна непосећена грана у графу, те један по један чвор текућег пута пребацујемо с краја пута у Ојлеров циклус; пре него што чвор пребацимо у Ојлеров циклус проверавамо да ли постоји још нека непосећена грана у графу која креће из тог чвора, ако постоји крећемо у потрагу за новим циклусом из тог чвора, а ако не постоји, онда чвор пребацујемо у Ојлеров циклус. Након пребацивања чворова* 0*,* 5 *и* 3 *утврђује се да из чвора* 1 *постоји грана ка чвору* 2 *која до сада није била посећена те откривамо нови циклус* $(1, 2, 1)$ *и чворове овог циклуса додајемо на крај текућег пута. Након овог циклуса, све док не исцрпемо све чворове из текућег пута нећемо пронаћи ниједну нову грану. Коначно, закључујемо да је у овом графу један од Ојлерових циклуса* $(0, 5, 4, 0, 1, 5, 1, 2, 1, 3, 5, 0)$*. Садржај текућег пута и дела конструисаног Ојлеровог циклуса корак по корак приказани су у табели 2.2.*

Табела 2.2: Пример извршавања Хирхолцеровог алгоритма за граф са слике 2.63.

tekuciCvor	*tekuciPut*	*ojlerovCiklus*
0	(0)	
5	(0, 5)	
4	(0, 5, 4)	
0	(0, 5, 4, 0)	
1	(0, 5, 4, 0, 1)	
5	(0, 5, 4, 0, 1, 5)	
1	(0, 5, 4, 0, 1, 5, 1)	
3	(0, 5, 4, 0, 1, 5, 1, 3)	
5	(0, 5, 4, 0, 1, 5, 1, 3, 5)	
0	(0, 5, 4, 0, 1, 5, 1, 3, 5, 0)	
5	(0, 5, 4, 0, 1, 5, 1, 3, 5)	(0)
3	(0, 5, 4, 0, 1, 5, 1, 3)	(0, 5)
1	(0, 5, 4, 0, 1, 5, 1)	(0, 5, 3)
2	(0, 5, 4, 0, 1, 5, 1, 2)	(0, 5, 3)
1	(0, 5, 4, 0, 1, 5, 1, 2, 1)	(0, 5, 3)
2	(0, 5, 4, 0, 1, 5, 1, 2)	(0, 5, 3, 1)
1	(0, 5, 4, 0, 1, 5, 1)	(0, 5, 3, 1, 2)
5	(0, 5, 4, 0, 1, 5)	(0, 5, 3, 1, 2, 1)
1	(0, 5, 4, 0, 1)	(0, 5, 3, 1, 2, 1, 5)
0	(0, 5, 4, 0)	(0, 5, 3, 1, 2, 1, 5, 1)
4	(0, 5, 4)	(0, 5, 3, 1, 2, 1, 5, 1, 0)
5	(0, 5)	(0, 5, 3, 1, 2, 1, 5, 1, 0, 4)
0	(0)	(0, 5, 3, 1, 2, 1, 5, 1, 0, 4, 5)
–		(0, 5, 3, 1, 2, 1, 5, 1, 0, 4, 5, 0)

Време извршавања Хирхолцеровог алгоритма у усмереном графу који је представљен листама повезаности износи $O(|E|)$. Наиме, уколико је $|V| > |E|$, онда граф има изоловане чворове који се током алгоритма не разматрају. Истакнимо и то да уколико је граф неусмерен, онда приликом брисања неке гране, треба обрисати обе њене копије: (u, v) и (v, u), што је операција која се не извршава ефикасно у случају репрезентације графа листама повезаности. Стога је Хирхолцеров алгоритам у случају неусмереног графа мање ефикасан.

Ако се тражи Ојлеров пут (а не циклус), алгоритам се примењује у истом облику, једино што извршавање мора да почне од чвора чији је излазни степен за један већи од улазног.

2.7.1.2 Флеријев алгоритам

Други познати алгоритам за конструкцију Ојлерових циклуса или путева у графу, поред Хирхолцеровог, је *Флеријев*[13] *алгоритам* (енгл. Fleury's algorithm). Он креће од произвољног чвора и у сваком кораку текући пут проширује неком граном из текућег чвора, коју затим уклања из графа. Ако је могуће, бира се грана која није мост тј. грана која не разбија граф на више компонената повезаности. Ако таква грана не постоји, онда се пут проширује граном која јесте мост. Наиме, ако бисмо изабрали грану која је мост када из тог чвора постоји још нека грана, прешли бисмо у компоненту повезаности којој не припада тај чвор и не бисмо могли да се вратимо до њега и да пређемо том граном. Нагласимо да иако полазни Ојлеров граф сигурно нема мостова, они се могу појавити током извршавања алгоритма, јер се из графа избацују гране.

Овај алгоритам се заснива на детекцији мостова у графу у сваком кораку и мање је ефикасан – његова сложеност износи $O(|E|^2)$. Постоје инкрементални алоритми за детекцију мостова чијом се употребом сложеност алгоритма може поправити, али Флеријев алгоритам и даље остаје сложенији и неефикаснији од Хирхолцеровог, па га нећемо даље анализирати.

2.7.2 Хамилтонови путеви и циклуси

Један од важних проблема у биоинформатици јесте мапирање генома, где је задатак искомбиновати велики број малих фрагмената генетског кода у јединствену геномску секвенцу. Овај проблем се може разматрати као графовски: сваки од фрагмената одговара чвору графа, а свако преклапање (поклапање краја неког фрагмента са почетком неког другог) грани графа. Питање које се поставља јесте да ли је могуће направити прост пут у графу који пролази кроз сваки чвор графа тачно једном.

Хамилтонов[14] *пут* (енгл. Hamiltonian path) је пут који посећује сваки чвор графа тачно једном. Ако Хамилтонов пут почиње и завршава се у истом чвору он се назива *Хамилтонов циклус* (енгл. Hamiltonian cycle, Hamiltonian circuit).

Пример 2.7.5

Граф приказан на слици 2.56 садржи Хамилтонов пут A, B, C, D, E. Граф са слике 2.56 има и Хамилтонов циклус који почиње и завршава се у чвору A: то је циклус (A, B, E, D, C, A).

Граф који садржи Хамилтонов циклус зове се *Хамилтонов граф* (енгл. Hamiltonian graph). Хамилтонови циклуси добили су име у част Вилијама Хамилтона који је 1857.

[13] Пјер Хенри Флери (фр. Pierre-Henry Fleury), рођен 1833, француски математичар и наставник.
[14] Вилијем Хамилтон (енгл. William Rowan Hamilton), (1805-1865), ирски математичар.

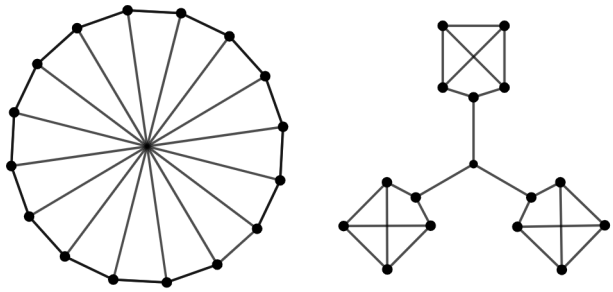
године изумео слагалицу која укључује потрагу за овом врстом циклуса у графу сачињеном од ивица додекаедра.

Природно се разматра проблем одређивања Хамилтонових циклуса.

Проблем

За дати граф (било усмерен, било неусмерен) утврдити да ли има Хамилтонов циклус и ако има одредити га.

За разлику од проблема Ојлерових циклуса, проблем налажења Хамилтонових циклуса (односно карактеризације Хамилтонових графова) је врло тежак. Да би се проверило да ли је граф Ојлеров, довољно је знати степене његових чворова. За утврђивање да ли је граф Хамилтонов то није довољно. Заиста, два графа приказана на слици 2.64 имају по 16 чворова степена 3 (дакле имају исти број чворова и исте степене чворова), али је први Хамилтонов, а други очигледно није. Проблем испитивања да ли је дати граф Хамилтонов спада у класу NP-комплетних проблема, па није познато да ли постоји субекспоненцијални алгоритам за његово решавање.



Слика 2.64: Два графа са по 16 чворова степена 3, од којих је први Хамилтонов, а други није.

Алгоритам грубе силе подразумева проверу свих пермутација чворова и његова сложеност је $O(|V|!)$, што је изразито неефикасно.

Могуће је дизајнирати алгоритам заснован на динамичком програмирању чија би сложеност била донекле боља $O(|V| \cdot 2^{|V|})$. Основна идеја је да се решавају потпроблеми облика: „За фиксирани почетни чвор u, скуп чворова S и завршни чвор v, да ли постоји пут који почиње из чвора u, пролази кроз све чворове из S и завршава се чвором v". Обележимо ову логичку вредност са $DP(S, v)$. Базу чини празан скуп S и тада $DP(\emptyset, v)$ важи тачно за чворове v за које постоји грана (u, v). За непразне скупове S важи $DP(S, v)$ ако и само ако постоји $v' \in S$ тако да важи $DP(S \setminus \{v\}, v')$ и постоји

грана (v', v). На крају, Хамилтонов циклус постоји ако постоји неки чвор v такав да важи $DP(V \setminus \{u\}, v)$ и постоји грана (v, u).

Ми се у овом уџбенику нећемо детаљније бавити Хамилтоновим путевима и циклусима.

Задатак: Де Брујнов низ

Де Брујнов[15] низ је циклични низ бројева из интервала $[0, k)$, који као своје сегменте садржи све могуће n-точлане ниске чији су елементи бројеви из интервала $[0, k)$. На пример, низ `0110` је де Брујнов за $k = 2$ и $n = 2$, јер садржи редом `01`, `11`, `10` и, на крају `00` (јер је допуштено да се елементи читају и циклично, део са краја низа, па затим са почетка). Најкраћи де Брујнов низ садржи тачно k^n елемената. Заиста, конструктивно ћемо показати да постоји де Брујнов низ дужине k^n. Са друге стране, постоји тачно k^n различитих n-точланих низова чији су елементи бројеви из интервала $[0, k)$, сваки од њих се јавља у де Брујновом низу и почиње на различитој позицији, па зато дужина де Брујновог низа мора бити бар k^n. Написати програм који за дато n и k исписује неки најкраћи де Брујнов низ.

Опис улаза

Број n ($1 \leq n \leq 10$), а затим и број k ($2 \leq k \leq 4$).

Опис излаза

На стандардни излаз исписати било који најкраћи де Брујнов низ.

Пример 1

Улаз	*Излаз*
`2 2`	`0110`

Пример 2

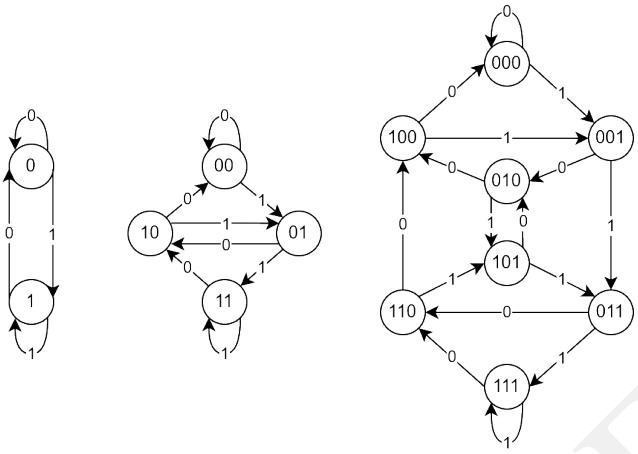
Улаз	*Излаз*
`3 3`	`0022212111220120211020010`

Решење

Сваки од k^n низова треба да се јави тачно једном као сегмент траженог низа, при чему се свака два узастопна сегмента поклапају на $n - 1$ позиција. Два сегмента могу бити узастопна ако и само ако се последњих $n - 1$ бројева првог поклапа са $n - 1$ првих бројева другог сегмента. Ово је могуће моделовати усмереним графом. Сваки низ бројева из интервала $[0, k)$ дужине n можемо представити једним чвором графа (граф, дакле, има n^k чворова), а грана од чвора A до чвора B постоји ако и само ако низ A и низ B могу да буду узастопни сегменти (који се преклапају на $n-1$ позиција). Примери таквих графова за $k = 2$ и $n = 1$, $n = 2$ и $n = 3$, су приказани на слици.

Сваки пут у графу одговара бинарном низу и обратно. На пример, де Брујновом низу `00111010` за $n = 3$ и $k = 2$, одговара пут кроз чворове `001`, `011`, `111`, `110`, `101`, `010`.

[15] Николас де Брујн (хол. Nicolaas Govert de Bruijn), (1918-2021), холандски информатичар.



Де Брујнов граф.

Де Брујнов низ се може добити проналаском Хамилтоновог циклуса у графу (јер Хамилтонов циклус обилази сваки чвор тачно једном, пре него што се врати у почетни чвор). Дужина низа која одговара Хамилтоновом циклусу је $k^n + n - 1$, при чему се последњих $n-1$ елемената низа поклапа са првих $n-1$ елемената, па се они могу изоставити (што одговара изостављању последњих $n-1$ елемената Хамилтоновог циклуса).

Међутим, де Брујнов низ за дато n и k се може добити и од графа за то k и $n-1$. Сваком сегменту $a_1, a_2, \ldots, a_{n-1}, a_n$ одговара грана од чвора $a_1, \ldots, a_{n-1}$ до чвора $a_2, \ldots a_n$. У том случају се кроз сваку грану пролази само једном, што значи да де Брујновим низовима одговарају Ојлерови циклуси и обратно.

Задатак зе зато може решити проналажењем Ојлеровог циклуса у графу за дато k и $n-1$. Ојлеров циклус можемо пронаћи Хирхолцеровим алгоритмом. Ако знамо све чворове Ојлеровог циклуса (при чему су први и последњи једнаки), низ можемо добити читајући редом њихове последње бројеве, и изостављајући последњи чвор.

У наредној имплементацији рекурзивно набрајамо све варијације и тако градимо све чворове графа као експлицитне ниске цифара (имплементација би се могла унапредити коришћењем неке ефикасније репрезентације графа).

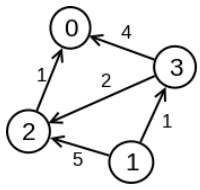
```
// pomocna funkcija za generisanje varijacija
void sviCvorovi(int m, int k, string& s, vector<string>& rezultat) {
  if (m == 0) {
    rezultat.push_back(s);
    return;
```

```
  }
  for (int i = 0; i < k; i++) {
    s[m-1] = '0' + i;
    sviCvorovi(m-1, k, s, rezultat);
  }
}

// cvorovi grafa su sve varijacije duzine m od azbuke {0, 1, .., k-1}
vector<string> sviCvorovi(int m, int k) {
  vector<string> rezultat;
  string s(m, ' ');
  sviCvorovi(m, k, s, rezultat);
  return rezultat;
}

int main() {
  int n;
  cin >> n;
  int k;
  cin >> k;

  // generisemo vektor svih cvorova grafa
  // cvorovi su varijacije duzine m od azbuke {0, 1, .., k-1}
  vector<string> cvorovi = sviCvorovi(n-1, k);

  // za svaki cvor pamtimo koja grana je na redu za obilazak
  unordered_map<string, int> tekucaGrana;
  // na pocetku nismo obisli nijednu granu
  for (const string& cvor : cvorovi)
    tekucaGrana[cvor] = 0;

  // Ojlerov put određen čvorovima kroz koje se prolazi
  vector<string> ojlerovPut;

  stack<string> tekuciPut;
  // krecemo, na primer, od cvora 00..00
  tekuciPut.push(string(n-1, '0'));
  while (!tekuciPut.empty()) {
    string tekuciCvor = tekuciPut.top();
```

```
    // ako postoji neobrađena grana iz tekućeg čvora
    if (tekucaGrana[tekuciCvor] < k) {
      // oznaka na toj grani
      char grana = '0' + tekucaGrana[tekuciCvor]++;
      // čvor do koga vodi ta grana
      string sledeciCvor = tekuciCvor.substr(1) + string(1, grana);
      tekuciPut.push(sledeciCvor);
    } else {
      ojlerovPut.push_back(tekuciCvor);
      tekuciPut.pop();
    }
  }

  // ispisujemo rezultujuci niz
  // (dobijen od poslednjih brojki na cvorovima kojima prolazi Ojlerov put)
  for (int i = 0; i < ojlerovPut.size() - 1; i++)
    cout << ojlerovPut[i].back();
  cout << endl;

  return 0;
}
```

2.8 Тежински графови

У проблемима који следе бавићемо се *тежинским графовима* (енгл. weighted graph), односно графовима у којима је свакој грани придружена њена *тежина* (енгл. weight). Уместо тежина гране често ћемо користити и термин дужина гране и под термином *дужина пута* сматраћемо збир дужина грана на том путу (а не број грана на том путу). На слици 2.65 приказан је један усмерени тежински граф. У овом графу дужина пута $(1, 2, 0)$ износи $5 + 1 = 6$, док дужина пута $(1, 3, 2, 0)$ износи $1 + 2 + 1 = 4$.



Слика 2.65: Усмерени тежински граф.

У програмском језику C++ тежински граф можемо представити или матрицом повезаности у којој се уместо логичких вредности чувају тежине грана (уз неку специјалну нумеричку вредност која означава да чворови нису повезани) или листама повезаности, где се у сваком елементу листе повезаности чува индекс крајњег чвора гране и тежина гране. Ако тежински граф представљамо листама повезаности и ако су дужине грана целобројне, граф можемо декларисати на следећи начин:

```
typedef int Cvor;
typedef int Tezina;
vector<vector<pair<Cvor, Tezina>>> listaSuseda(n);
```

Нову грану $(cvorOd, cvorDo)$ тежине t додајемо у граф на следећи начин:

```
listaSuseda[cvorOd].emplace_back(cvorDo, t);
```

Подсетимо се да се применом функције `emplace_back` најпре конструише уређени пар чвора и његове тежине, а затим тај уређени пар додаје на крај вектора.

Пример 2.8.1

Усмерени тежински граф са слике 2.65 задајемо листама повезаности:

```
vector<vector<pair<Cvor, Tezina>>> listaSuseda
    {{},  {{2,5}, {3,1}},  {{0,1}},  {{0,4}, {2,2}}};
```

Неусмерене графове можемо сматрати усмереним, при чему свакој њиховој неусмереној грани одговарају две усмерене гране исте дужине. Алгоритми које ћемо разматрати над тежинским графовима односе се и на усмерене и на неусмерене графове.

2.9 Најкраћи путеви из задатог чвора

Постоји много ситуација у којима се јавља потреба да се израчунају најкраћи путеви кроз граф. На пример, граф може одговарати ауто–карти: чворови су градови, а дужине грана дужине директних путева између градова (или време потребно да се тај пут пређе, или трошкови да се изгради). Задатак је пронаћи најкраћи пут (у смислу дужине, протеклог времена или потребног улагања) од једног града до другог. Задаци овог типа се формулишу на произвољним тежинским графовима, тако што се под дужином пута подразумева збир свих тежина грана на том путу (па је најкраћи пут заправо пут са најмањим збиром тежина грана).

Алгоритми за одређивање најкраћих путева играју важну улогу и у рутирању података кроз комуникациону мрежу, у контексту одређивања најефикаснијих путања (у смислу

минимизовања кашњења или загушења) од извора до одредишта. Алгоритми за рачунање најкраћих путева налазе примену и у оптимизовању ланца снабдевања, односно у одређивању најкраћих путева за транспорт робе од произвођача до дистрибутивних центара, а затим и крајњих потрошача.

Иако је често потребно да се нађе најкраћи пут од датог почетног до датог завршног чвора, налажење тог најкраћег пута није могуће без налажења најкраћих путева од почетног до многих других чворова графа. Зато се обично разматра следећи проблем.

Проблем

За дати усмерени тежински граф $G = (V, E)$ и његов чвор s пронаћи најкраће путеве (путеве са најмањим збиром тежина грана) од s до свих осталих чворова.

Пример 2.9.1

Размотримо пример одређивања најкраћих путева од чвора 1 до свих осталих чворова у графу са слике 2.65:

- *најкраћи пут до чвора 3 је директна грана $(1, 3)$ дужине 1,*
- *најкраћи пут до чвора 2 је пут $(1, 3, 2)$ укупне дужине $1 + 2 = 3$, а не директна грана $(1, 2)$ дужине 5,*
- *најкраћи пут до чвора 0 је пут $(1, 3, 2, 0)$ укупне дужине $1 + 2 + 1 = 4$.*

2.9.1 Ациклички графови

Претпоставимо најпре да је граф $G = (V, E)$ ациклички. У том случају проблем је лакши и његово решење помоћи ће нам да проблем решимо и у општем случају. Тежине грана могу бити произвољне (позитивне, нула, па чак и негативне).

На пример, ако је граф дрво и тражимо најкраће путеве од корена до свих осталих чворова, тада до сваког чвора постоји јединствен пут и он је уједно и најкраћи. Дужине тих путева се могу одредити било рекурзивно (тако што се за сваки чвор рекурзивно одреди најкраћи пут од корена до његовог родитеља, који се затим продужава граном од родитеља до чвора), било итеративно (тако што ће се најкраћи пут од сваког чвора продужити гранама до његових наследника). Приметимо да се у основи заправо крије алгоритам заснован на динамичком програмирању (који је у случају рекурзивне имплементације одозго-наниже, а у случају итеративне одоздо-навише).

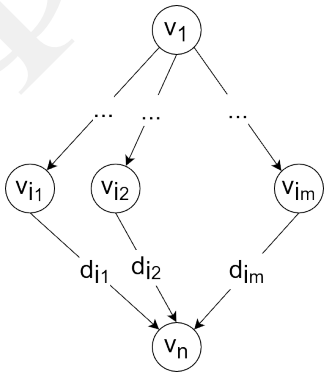
И за ацикличке графове који нису дрвета до решења се може доћи на сличан начин.

2.9.1.1 Рекурзивни приступ и динамичко програмирање

Опишимо рекурзивно решење проблема (не обазирући се у почетку на ефикасност). Нека је $|V| = n$. Пошто је граф ациклички, можемо да искористимо тополошко сортирање графа и да чворове у тополошком редоследу означимо са $v_1, v_2, \ldots, v_n$. Ако је редни број чвора s од кога треба одредити најкраће путеве у тополошком редоследу једнак k (тј. ако је $s \equiv v_k$), онда се чворови са редним бројевима мањим од k не морају ни разматрати, јер не постоји начин да се из чвора s до њих дође. Зато, једноставности ради, претпоставимо да је редни број чвора s од кога треба израчунати најкраће путеве у тополошком редоследу 1, тј. да је $s \equiv v_1$. Ако додатно претпоставимо да осим чвора s у графу нема других чворова чији је улазни степен нула, тада можемо бити сигурни да ће до сваког чвора постојати пут од полазног (у супротном, до неких чворова неће постојати пут, па за њих можемо претпоставити да је дужина најкраћег пута једнака $+\infty$). Редослед чворова добијен тополошким сортирањем је погодан за примену индукције, тј. сугерише једноставно рекурзивно решење, које се затим техником динамичког програмирања може оптимизовати и претворити у ефикасно итеративно решење.

Излаз из рекурзије наступа када се обрађује чвор v_1 и најкраћи пут од тог чвора до њега самог је дужине 0.

Посматрајмо последњи чвор у тополошком редоследу чворова, односно чвор v_n. Означимо дужину најкраћег пута од s до произвољног чвора v_i са $v_i.SP$ (енгл. shortest path), а дужину гране (x, y) између произвољна два чвора x и y са $d(x, y)$. Да бисмо одредили вредност $v_n.SP$, довољно је да проверимо само оне чворове v_i из којих постоји грана до v_n, јер се најкраћи пут до v_n мора завршити неком од тих грана (слика 2.66).



Слика 2.66: Рачунање најкраћег пута од чвора $s \equiv v_1$ до последњег чвора v_n у тополошком поретку. Чворови $v_{i_1}, \ldots, v_{i_m}$ су сви чворови из којих постоји грана до v_n.

Пошто се најкраћи путеви до свих осталих чворова могу одредити рекурзивно, вредност $v_n.SP$ биће једнака минимуму збира $v_i.SP + d(v_i, v_n)$, по свим чворовима v_i из којих

води грана до чвора v_n:

$$v_n.SP = \min_{(v_i,v_n)\in E}\{v_i.SP + d(v_i, v_n)\}.$$

Да ли је тиме проблем решен? Најкраћи путеви до чворова $v_1, \ldots v_{n-1}$ су одређени без разматрања чвора v_n. Питање је да ли додавање чвора v_n у скуп чворова до којих смо одредили најкраће путеве може да скрати најкраћи пут од v_1 до неког од њих, тј. да ли је могуће да је најкраћи пут преко v_n краћи од најкраћег пута који не води преко чвора v_n. Пошто је v_n последњи чвор у тополошком редоследу, ни један други чвор није достижан из v_n, па се дужине осталих најкраћих путева не мењају. Дакле, уклањање чвора v_n из графа, проналажење најкраћих путева у преосталом графу, враћање чвора v_n назад у граф и рачунање растојања до њега су основни делови алгоритма.

Из овог разматрања директно следи одговарајући рекурзивни алгоритам. Међутим, као што је то чест случај са рекурзивним алгоритмима, он може бити веома неефикасан, услед поновљених рекурзивних позива (јер из чвора може водити више грана). Ово се, наравно, оптимизује неким обликом динамичког програмирања. У овом случају је задатак боље решавати индуктивно, тако што се чворови обрађују у тополошком редоследу, најкраћи пут до почетног чвора се иницијализује на 0, док се за сваки наредни чвор најкраћи пут рачуна применом приказане рекурентне формуле. Ово одговара стандардној техници динамичког програмирања навише.

Графовски алгоритми су често засновани на неком облику индуктивно-рекурзивне конструкције. У таквим проблемима је сваком чвору графа потребно придружити вредност неке статистике (у нашем примеру то је најкраћи пут од почетног чвора). Вредност статистике чвора обично зависи од вредности статистика његових суседа и често је највећи изазов у решењу пронаћи редослед обраде чворова који ће омогућити да су приликом обраде чвора статистике свих његових суседа већ исправно израчунате тј. да могу бити израчунате рекурзивним позивима који неће обухватити позив за чвор који се тренутно обрађује (јер би се у том случају упало у бесконачну рекурзију). У нашем примеру, то је омогућио тополошки редослед. Пошто је чвор често сусед већем броју чворова, да би се избегло вишеструко израчунавање њему придружене статистике, пожељно је користити или мемоизацију (памћење вредности те статистике у чвору или посебном низу) или динамичко програмирање навише (где се вредности памте на исти начин).

2.9.1.2 Индуктивни приступ: истовремено тополошко сортирање и одређивање најкраћих путева

У претходно описаном поступку је пре почетка тражења најкраћих путева потребно извршити тополошко сортирање графа. Такође, потребно је за сваки чвор одредити гране које воде до њега, што није ефикасна операција када се користе листе повезано-

сти (гране су усмерене од предака ка потомцима у тополошком редоследу, а не обратно). Конструишимо сада алгоритам за тражење најкраћих путева у ацикличком графу, у ком се тополошко сортирање обавља истовремено са проналажењем најкраћих путева. Другим речима, циљ је објединити два пролаза кроз граф (један за тополошко сортирање и други за налажење најкраћих путева) у један; притом се све време израчунавања врше индуктивно, од предака ка потомцима у тополошком редоследу.

Размотримо начин на који се алгоритам рекурзивно извршава (после налажења тополошког редоследа). Први корак је позив рекурзивне процедуре за чвор v_n. Процедура затим позива рекурзивно саму себе, све док се не дође до чвора $s \equiv v_1$. У том тренутку се дужина најкраћег пута од чвора s до чвора s поставља на 0, и рекурзија почиње да се „размотава". Разматра се чвор v_2; дужина најкраћег пута до њега изједначује се са дужином гране (v_1, v_2), ако она постоји; у противном, не постоји пут од v_1 до v_2. Следећи корак је провера чвора v_3. У чвор v_3 улазе највише две гране — од чворова v_1 и/или v_2, па се упоређују дужине одговарајућих путева. Уместо оваквог извршавања рекурзије уназад, покушаћемо да исте кораке извршимо унапред.

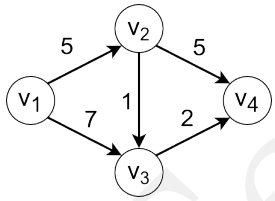
Индукција се примењује према растућим редним бројевима у тополошком редоследу почевши од $s \equiv v_1$. Овај редослед ослобађа нас потребе да редне бројеве унапред знамо, па ћемо бити у стању да извршавамо истовремено и алгоритам за одређивање тополошког уређења и алгоритам за одређивање најкраћих путева од чвора s. Дакле, можемо размотрити наредну индуктивну хипотезу.

Индуктивна хипотеза. Знамо првих $k-1$ чворова у тополошком редоследу чворова и дужине најкраћих путева од чвора s до њих.

Потребно је да одредимо чвор v_k са редним бројем k. То може да буде произвољни чвор у који не води ни једна грана из чвора са редним бројем већим од k (тј. било који чвор степена 0 када се избаце све гране из претходних чворова). Да бисмо пронашли најкраћи пут до v_k, морамо да проверимо све гране које воде у њега. Тополошки редослед гарантује да све такве гране полазе из чворова са мањим редним бројевима. Према индуктивној хипотези ти чворови су познати, као и дужине најкраћих путева до њих. За сваку грану (v_i, v_k) знамо дужину $v_i.SP$ најкраћег пута од s до v_i, па је дужина најкраћег пута од s до v_k преко v_i једнака $v_i.SP + d(v_i, v_k)$. Поред тога, као и раније, не морамо да водимо рачуна о евентуалним променама најкраћих путева ка чворовима са мањим редним бројевима од k, јер се до њих не може доћи из чвора v_k.

Да не бисмо за сваки чвор одређивали чворове из којих постоји грана ка њему, јер то није ефикасна операција у репрезентацији графа листама повезаности, можемо памтити дужине $v.\overline{SP}$ тренутно познатих најкраћих путева до свих чворова v, укључујући и оне са већим редним бројем од текућег чвора v_k. Приликом разматрања чвора v_k потребно је једино размотрити гране (v_k, v_j) које полазе из њега, за чвор v_j проверити да ли је

вредност $v_k.\overline{SP} + d(v_k, v_j)$ мања од $v_j.\overline{SP}$, и ако јесте, ажурирати вредност $v_j.\overline{SP}$. Дакле уместо да грану графа обрађујемо „уназад", односно у тренутку када обрађујемо њен завршни чвор, ми је обрађујемо „унапред", у тренутку када обрађујемо њен полазни чвор. Пошто се те гране већ обрађују током Кановог алгоритма приликом избацивања чвора v_k, када се смањују улазни степени чворова v_j, ажурирање најкраћих растојања можемо урадити у истој петљи. Пошто су пре обраде чвора v_k већ обрађене све гране до v_k из претходних чворова v_i и пошто је свака од њих обрађивана у тренутку када је већ било познато најкраће растојање до v_i, приликом обраде чвора v_k низ најкраћих растојања садржи најкраће растојање до v_k, тј. важи $v_k.SP = v_k.\overline{SP}$. Наиме, уместо да се алгоритам одређивања минимума покрене у тренутку обраде чвора v_k, он се мало по мало извршавао током обраде претходних чворова v_j.



Слика 2.67: Рачунање најкраћег пута у ацикличком графу: чворови су означени редом који одговара њиховој позицији у тополошком поретку.

Пример 2.9.2

Размотримо извршавање алгоритма на примеру ацикличког графа са слике 2.67. Редослед чворова у тополошком поретку графа је v_1, v_2, v_3, v_4. Нека је задатак одредити најкраће путеве из чвора $s \equiv v_1$.

- *На почетку постављамо вредност $v_1.\overline{SP}$ најкраћег пута до чвора v_1 на 0 (и то је коначна вредност $v_1.SP$ најкраћег пута до чвора v_1), а вредности најкраћих путева $v_j.\overline{SP}$ до свих осталих чворова v_j на $+\infty$.*

  Разматрамо гране које полазе из чвора v_1: то су (v_1, v_2) и (v_1, v_3), и ажурирамо најкраће путеве до чвора v_2 и до чвора v_3 на $v_2.\overline{SP} = \min\{+\infty, 0 + 5\} = 5$, а $v_3.\overline{SP} = \min\{+\infty, 0 + 7\} = 7$.

- *Други чвор у тополошком редоследу је v_2: текућу вредност $v_2.\overline{SP}$ најкраћег пута до њега проглашавамо за коначну вредност најкраћег пута, односно проглашавамо $v_2.SP = 5$.*

  Затим разматрамо гране које излазе из чвора v_2: то су гране (v_2, v_3) и (v_2, v_4) и ажурирамо најкраће путеве до чворова v_3 и v_4: $v_3.\overline{SP} = \min\{7, 5 + 1\} = 6$ и $v_4.\overline{SP} = \min\{+\infty, 5 + 5\} = 10$.

- *Трећи чвор у тополошком поретку је v_3, те текућу вредност $v_3.\overline{SP}$ најкраћег пута до њега проглашавамо за коначну, односно проглашавамо $v_3.SP = 6$.*

  Разматрамо једину грану (v_3, v_4) која полази из чвора v_3 и ажурирамо вредност $v_4.\overline{SP}$ најкраћег пута до чвора v_4: $v_4.\overline{SP} = \min\{10, 6 + 2\} = 8$.

- *На крају проглашавамо као коначну и дужину $v_4.\overline{SP}$ најкраћег пута до чвора v_4, односно проглашавамо $v_4.SP = 8$, чиме се алгоритам завршава.*

Из претходног разматрања следи наредна теорема о коректности овог алгоритма.

Теорема 2.9.1

Нека је $G = (V, E)$ усмерен ациклички тежински граф и v_1 неки његов чвор улазног степена 0. Ако се најкраћа растојања иницијализују тако да је $v_1.\overline{SP} = 0$ и $v_i.\overline{SP} = +\infty$, за свако $1 < i \leq n = |V|$, тада се након спровођења алгоритма добија тополошки редослед чворова $v_1, \ldots, v_n$ и за сваки чвор $v_i \in V$ важи $v_i.\overline{SP} = v_i.SP$, тј. низ $\overline{SP}$ садржи најкраће растојање од чвора v_1 до свих осталих чворова у графу.

До сада смо разматрали израчунавање дужина најкраћих путева, а не и рачунање самих најкраћих путева. Приметимо да смо најкраће путеве одређивали један по један: сваки нови најкраћи пут се добија продужавањем неког претходно одређеног најкраћег пута последњом граном на путу. Гране којима се врши продужавање формирају тзв. *дрво најкраћих путева* (енгл. shortest-path tree). Како бисмо могли да реконструишемо најкраће путеве до сваког чвора, за сваки чвор памтимо његовог претходника (родитеља) на најкраћем путу од чвора s. Једино чвор $s \equiv v_1$ нема родитеља на најкраћем путу (ако чвор s није први у тополошком редоследу, родитеље неће имати ни чворови који су испред њега у тополошком редоследу). На тај начин можемо једноставно да реконструишемо саме најкраће путеве.

Размотримо имплементацију алгоритма: претпоставићемо да је потребно одредити најкраће путеве од чвора 0 до свих осталих чворова. Претпоставимо, једноставности ради, и да је полазни чвор једини чвор улазног степена 0. Вредности $+\infty$ ћемо у имплементацији представљати највећим целим бројем који може да се запише у типу `int`. То олакшава имплементацију јер се избегавају гранања, али треба бити обазрив да се тај број током алгоритма не сабира са другим бројевима (иначе би дошло до прекорачења).

```
// umesto vrednosti beskonacno koja oznacava da put nije pronađen
// koristimo najveći broj koji se može zapisati u tipu težine grana
const Tezina INF = numeric_limits<Tezina>::max();

// funkcija koja stampa put od polaznog cvora v do datog cvora
```

```
// kroz grane drveta najkracih puteva
void odstampajPutDoCvora(Cvor cvor, vector<Cvor> roditelji) {
  // ako postoji put do roditeljskog cvora, stampamo ga
  if (roditelji[cvor] != -1)
     odstampajPutDoCvora(roditelji[cvor], roditelji);
  // stampamo tekuci cvor
  cout << " " << cvor;
}

// funkcija koja stampa najkraci put do datog cvora i njegovu duzinu
void odstampajNajkraciPut(Cvor cvor, vector<Cvor> roditelji,
                          vector<Tezina> minRastojanja) {
  cout << "Najkraci put do cvora " << cvor << " je:";
  odstampajPutDoCvora(cvor, roditelji);
  cout << " i duzine je " << minRastojanja[cvor] << endl;
}

// funkcija koja racuna najkrace puteve od cvora 0 u aciklickom grafu
void aciklicki_najkraci_putevi() {
  int brojCvorova = listaSuseda.size();
  // niz koji za svaki cvor cuva njegov ulazni stepen
  vector<int> ulazniStepeni(brojCvorova, 0);
  // niz koji za svaki cvor cuva duzinu
  // trenutno poznatog najkraceg puta do njega
  vector<Tezina> minRastojanja(brojCvorova, INF);
  // niz koji za svaki cvor cuva roditelja u najkracem putu
  vector<Cvor> roditelji(brojCvorova, -1);

  // najkraci put od cvora 0 do njega samog postavljamo na 0
  minRastojanja[0] = 0;

  // inicijalizujemo niz ulaznih stepena cvorova
  // potreban za izvršavanje Kanovog algoritma
  for (Cvor cvor = 0; cvor < brojCvorova; cvor++)
    for (const auto& [sused, tezina] : listaSuseda[cvor])
      ulazniStepeni[sused]++;

  // red koji cuva cvorove ulaznog stepena nula
  queue<Cvor> cvoroviStepenaNula;
```

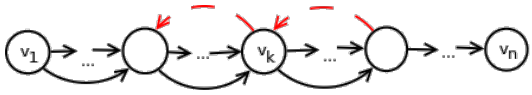
```
  // pretpostavljeno je da je polazni cvor 0 jedini stepena 0
  cvoroviStepenaNula.push(0);

  while(!cvoroviStepenaNula.empty()) {
    // cvor sa pocetka reda je naredni u topoloskom redosledu
    Cvor cvor = cvoroviStepenaNula.front();
    cvoroviStepenaNula.pop();
    // do njega je odredjen najkraci put i stampamo ga
    odstampajNajkraciPut(cvor, roditelji, minRastojanja);

    for (const auto& [sused, tezina] : listaSuseda[cvor]) {
      // ako je do nekog cvora kraci put preko upravo razmatranog cvora
      // vrsimo azuriranje najkraceg rastojanja do tog cvora
      if (minRastojanja[cvor] + tezina < minRastojanja[sused]) {
         minRastojanja[sused] = minRastojanja[cvor] + tezina;
         // azuriramo koji je roditeljski cvor na najkracem putu
         roditelji[sused] = cvor;
      }
      ulazniStepeni[sused]--;
      // ako je stepen nekog od suseda pao na 0, dodajemo ga u red
      if (ulazniStepeni[sused] == 0)
         cvoroviStepenaNula.push(sused);
    }
  }
}
```

У алгоритму се свака грана разматра по једном у току иницијализације улазних степена чворова, и по једном у тренутку кад се њен полазни чвор уклања из реда. Приступ реду захтева константно време. Сваки чвор се разматра тачно једном. Према томе, временска сложеност алгоритма за одређивање најкраћих путева у ацикличком графу је у најгорем случају $O(|V| + |E|)$.

2.9.2 Графови са ненегативним гранама: Дајкстрин алгоритам

Кад граф није ациклички, не постоји тополошки редослед, па се разматрани алгоритам не може директно применити. Међутим, основне идеје се могу искористити и у општем случају, додуше, само код графова код којих су дужине грана ненегативне. Наиме, једноставност алгоритма за одређивање најкраћих путева из задатог чвора у ацикличком графу последица је следеће особине тополошког редоследа: ако је v_k чвор са редним бројем k у тополошком поретку, онда:

- не постоје путеви од чворова са бројевима већим од k до чвора v_k,
- дуално, не постоје путеви од чвора v_k до чворова са бројевима мањим од k (слика 2.68).



Слика 2.68: Када чворове поређамо слево надесно у тополошком редоследу, у ацикличком графу могу постојати само гране слева надесно, које воде од чворова са мањим редним бројем ка чворовима са већим редним бројем (означене црном бојом), док гране које воде здесна улево, од чворова са већим редним бројем ка чворовима са мањим редним бројем, не могу постојати (означене црвеном бојом).

Ова особина тополошког редоследа омогућује нам да организујемо редослед рекурзивних тј. индуктивних израчунавања тј. да нађемо најкраћи пут од чвора $s \equiv v_1$ до чвора v_k, не водећи рачуна о чворовима који су после v_k у тополошком редоследу (јер од њих сигурно не постоје путеви до v_k). Може ли се некако дефинисати редослед чворова произвољног графа (који није нужно ациклички) који би омогућио нешто слично?

Кључна идеја је *размотрити чворове графа редом према дужинама најкраћих путева од чвора s до њих*. Наиме, ако се чворови уреде у низ $v_1, v_2, \ldots, v_n$ неопадајуће на основу дужина тих путева, и ако у графу нема грана негативне тежине, важе веома слична својства као код тополошког поретка:

- најкраћи пут до чвора v_k не пролази преко чворова са редним бројевима већим од k,
- дуално, најкраћи пут до неког чвора са редним бројем мањим од k не пролази преко чвора v_k.

Дакле, приликом одређивања најкраћег пута до v_k могу се занемарити путеви који воде преко чворова са редним бројевима већим од k, тј. треба размотрити само путеве преко чворова са редних бројева мањих од k, који могу могу бити одређени рекурзивно тј. индуктивно. Ако бисмо унапред могли да одредимо овај редослед чворова, рекурзивни алгоритам би могао да функционише по истом принципу као и у случају ацикличких графова. Избацили бисмо чвор v_n који је најудаљенији од чвора $s \equiv v_1$, израчунали растојања од s до свих осталих чворова, а затим бисмо анализирали гране (v_i, v_n) и растојање до v_n одредили као $\min_{(v_i,v_n)\in E}\{v_i.SP + d(v_i, v_n)\}$. Захваљујући претходним условима знамо да гране које воде из v_n, назад ка претходним чворовима не треба разматрати, тј. да одређивање најкраћег растојања до v_n не може ни на који начин ажурирати најкраћа растојања претходних чворова (она ће бити исправно одређена и пре разматрања чвора v_n). Излаз из овог рекурзивног поступка је случај $v_1 \equiv s$ (јер је

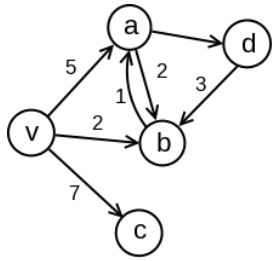
најближи чвор почетном чвору s управо чвор s) када је најкраће растојање једнако 0.

Међутим, алгоритам морамо организовати другачије, јер се, наравно, дужине путева, па ни овај поредак на почетку не знају, већ се израчунавају у току извршавања алгоритма.

- Пошто су све гране ненегативне дужине, најкраће растојање од чвора s до чвора s је 0 и чвор s је први у овом редоследу. Дакле, поново важи $s \equiv v_1$.
- Након тога проверавамо све гране које излазе из чвора v_1. Нека је (v_1, v_2) најкраћа међу њима. Пошто су, по претпоставци, све дужине грана ненегативне, најкраћи пут од чвора s до чвора v_2 је грана (v_1, v_2). Дужине свих других путева од чвора s до чвора v_2 су веће или једнаке од дужине ове гране. Чвор v_2 је, додатно, најближи од свих чворова чвору s. Према томе, знамо најкраћи пут до чвора најближег чвору s.
- Покушајмо да направимо следећи корак. Како можемо да пронађемо чвор до кога води наредни најкраћи пут, односно да одредимо чвор v_3 који је трећи најближи чвору s ($v_1 \equiv s$ и v_2 су прва два најближа). Једини путеви које треба узети у обзир су преостале гране из чвора v_1 (које не воде до v_2) или путеви који се састоје од две гране, тако да је прва грана (v_1, v_2), а друга је нека грана која полази из чвора v_2. Остали путеви могу бити само дужи од ових. Чвор v_3, дакле, одређујемо као онај чвор v_i различит од v_1 и v_2 за који је најмања вредност $\min\{d(v_1, v_i), d(v_1, v_2) + d(v_2, v_i)\}$.
- Сличан се поступак користи и за одређивање наредних чворова.

Пример 2.9.3

Размотримо граф са слике 2.69 и проблем тражења најкраћих путева од чвора s.



Слика 2.69: Усмерени тежински граф који садржи циклусе.

- *Први најближи чвор чвору s је он сâм.*
- *Други најближи чвор је један од суседа чвора s и то онај до кога води најкраћа грана – то је чвор b и најкраћи пут до b је дужине 2.*

- *Трећи најближи чвор чвору s је или неки други сусед чвора v или неки чвор до кога води грана из чвора b (као првог најближег чвора чвору s) – то је чвор a, до кога води пут (s, b, a) дужине $2 + 1 = 3$.*
- *Четврти најближи чвор чвору s је чвор до кога се стиже граном из неког од чворова s, b или a – то је чвор d, до кога води пут (s, b, a, d) дужине $2 + 1 + 1 = 4$.*
- *Пети најближи чвор је чвор c до кога се стиже граном из s, b, a или d (једина грана која из тих чворова води ка неком новом чвору је грана из s до c дужине 7).*

Може се формулисати следећа индуктивна хипотеза.

Индуктивна хипотеза. За задати граф $G = (V, E)$ и његов чвор s, умемо да одредимо k чворова најближих чвору s, као и дужине најкраћих путева до њих.

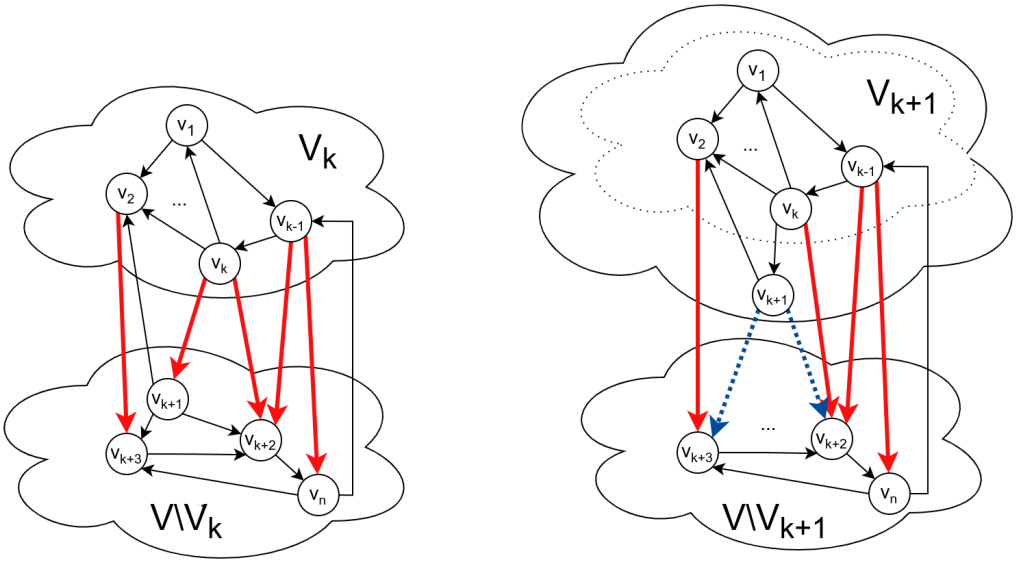
Индукција се примењује по броју чворова до којих су дужине најкраћих путева већ израчунате, а не по величини графа. Поред тога, претпоставља се да су то чворови најближи чвору s, и да умемо да их одредимо. Чвор s је најближи сам себи и на растојању је 0, па је база (случај $k = 1$) решена. Кад k достигне вредност $|V|$, решен је комплетан проблем. Ако се не траже растојања до свих чворова, него само до неког истакнутог чвора v', поступак се може прекинути и раније, чим се израчуна растојање до чвора v'.

Означимо са $V_k = \{v_1, \ldots, v_k\}$ скуп који се састоји од k најближих чворова чвору s (укључујући и $s \equiv v_1$). Проблем је пронаћи чвор v_{k+1} који је најближи чвору s међу чворовима ван V_k, и пронаћи најкраћи пут од s до v_{k+1}.

Најкраћи пут од s до v_{k+1} може да садржи само чворове из V_k. Он не може да садржи неки чвор v ван V_k, јер би тада чвор v био ближи чвору s од чвора v_{k+1}, ако су тежине грана позитивне. Ако постоје гране тежине нула, за чвор v_{k+1} ћемо одабрати неки од оних чворова ван V_k до којих најкраћи пут пролази само кроз чворове из V_k (бар један такав чвор мора да постоји). Према томе, да бисмо пронашли чвор v_{k+1}, довољно је да проверимо гране које спајају чворове из V_k са чворовима који нису у V_k; све друге гране се за сада могу игнорисати. Нека је (v_i, v) произвољна грана таква да је $v_i \in V_k$ и $v \notin V_k$. Таква грана одређује пут од s до v који се састоји од најкраћег пута од s до v_i (који је према индуктивној хипотези већ познат) и гране (v_i, v). Довољно је упоредити дужине свих таквих путева и изабрати најкраћи међу њима (слика 2.70, лево).

Алгоритам одређен овом индуктивном хипотезом извршава се на следећи начин. У свакој итерацији у скуп V_k се додаје нови чвор. У првој је то чвор s. У свакој наредној итерацији је то онај чвор $v \notin V_k$ који је најближи чвору s, тј. за који је најмања вредност израза:

$$\min \{v_i.SP + d(v_i, v) \mid v_i \in V_k\}. \tag{2.2}$$



Слика 2.70: Налажење следећег најближег чвора задатом чвору $s \equiv v_1$. Лево је приказана неоптимизована варијанта у којој се у сваком кораку израчунавају растојања до чворова скупа $V \setminus V_k$ тако што се анализирају све гране између скупова V_k и $V \setminus V_k$. Десно је приказана оптимизована варијанта. У тренутку проширивања скупа V_k чвором v_{k+1} довољно је ажурирати растојања само оних чворова до којих воде гране које иду из v_{k+1} ка чворовима из $V \setminus V_k$.

Из већ изнетих разлога, тако одабрани чвор v је заиста $(k+1)$-ви (следећи) најближи чвор чвору s, па га можемо обележити са v_{k+1}. Формулом (2.2) одређена је и стварна дужина најкраћег пута од чвора s до чвора v_{k+1} (она се неће даље смањивати). Према томе, додавање чвора v_{k+1} скупу V_k проширује индуктивну хипотезу.

Алгоритам је сада потпуно прецизиран, али му се ефикасност може побољшати. Основни корак алгоритма је проналажење следећег чвора v_{k+1}. То се остварује израчунавањем дужине најкраћег пута (преко чворова из V_k) до сваког чвора $v \notin V_k$ према формули (2.2). Можемо да, као и у случају ацикличког графа, у низу памтимо дужине $v.\overline{SP}$ тренутно познатих најкраћих путева до свих чворова $v \in V$, а да им при проширивању скупа V_k ажурирамо вредности. За чворове ван V_k то ће бити најкраћи путеви који воде искључиво преко чворова у V_k, тј. преко свих грана које полазе из чворова у V_k, а не и глобално најкраћи, јер можда до њих постоји и неки краћи пут који води преко чворова ван V_k, па укључује и неке до сада непрегледане гране.

Вредности $v.\overline{SP}$ за чворове ван V_k се мењају приликом проширивања скупа V_k (који фигурише у формули (2.2)). Ипак, реално је очекивати да проширивање скупа V_k чвором v_{k+1} не утиче на вредност растојања многих чворова $v \notin V_k$. Заиста, могу се променити само вредности за оне чворове v до којих постоји грана (v_{k+1}, v), тј. само оне вредности које одговарају путевима кроз новододати чвор v_{k+1}. Према томе, приликом додавања чвора v_{k+1} у скуп V_k треба проверити све гране од чвора v_{k+1} ка чворовима ван V_k (слика 2.70, десно). За сваку такву грану (v_{k+1}, v) упоређујемо збир $v_{k+1}.\overline{SP} + d(v_{k+1}, v)$ са тренутном вредношћу $v.\overline{SP}$, и по потреби ажурирамо (смањујемо) вредност $v.\overline{SP}$. Приметимо да смо исти овај начин ажурирања користили и за израчунавање најкраћих путева у ацикличком графу. Свака итерација алгоритма, дакле, обухвата налажење чвора $v \equiv v_{k+1}$ ван V_k са најмањом тренутном вредношћу $v.\overline{SP}$, и евентуално ажурирање вредности $v.\overline{SP}$ за његове суседе који нису у V_k. Пошто су приликом избора тог чвора v_{k+1} већ обрађене све гране које воде до њега од чворова v_i који су ближи чвору s од њега и пошто се на најкраћем путу до њега не може наћи ниједан други чвор ван V_k, текућа вредност $v_{k+1}.\overline{SP}$ се поклапа са стварном вредношћу $v_{k+1}.SP$ најкраћег растојања до њега.

Овај алгоритам добио је назив *Дајкстрин алгоритам* по Едзгару Дајкстри[16] који га је осмислио 1956. године. Он припада групи похлепних алгоритама, јер се у сваком кораку бира локално оптимално решење – најближи чвор и након обраде тренутно најближег чвора растојање до њега се више никада не разматра.

Најкраће путеве од чвора s до свих осталих чворова нашли смо тако што смо проналазили један по један најкраћи пут. Сваки нови пут је одређен једном граном, која продужује претходно познати најкраћи пут до новог чвора. Све те гране – продужеци

[16] Едзгар Дајкстра (хол. Edsger Wybe Dijkstra), (1930-2002), холандски информатичар.

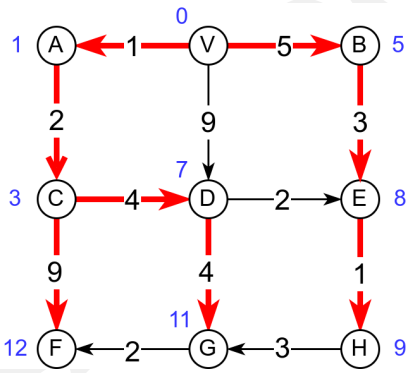
путева формирају дрво са кореном у чвору s. Ово дрво назива се *дрво најкраћих путева* и важно је за решавање многих проблема са путевима. Приметимо да ако би дужине свих грана у графу биле међусобно једнаке, онда би дрво најкраћих путева у ствари било BFS дрво са кореном у чвору s.

Претходним разматрањем смо доказали наредну теорему.

Теорема 2.9.2

Нека је дат тежински граф $G = (V, E)$ у ком су тежине свих грана ненегативне и нека је $s \in V$. Нека су у старту чворовима придружене вредности тако да је $s.\overline{SP} = 0$ и $v.\overline{SP} = +\infty$, за свако $v \in V$, $v \neq s$. Након што се понови n корака Дајкстриног алгоритма, тј. након што се сваки чвор графа убаци у скуп V_k, за сваки чвор $v \in V$ биће исправно одређен најкраћи пут од s, тј. важиће $v.SP = v.\overline{SP}$.

Пример 2.9.4



Слика 2.71: Пример извршавања Дајкстриног алгоритма. Приказано је дрво најкраћих путева из почетног чвора V. Уз сваки чвор је приказана дужина најкраћег пута до њега.

Извршавање Дајкстриног алгоритма за налажење најкраћих путева од чвора v у графу приказаном на слици 2.71 приказан је у табели 2.3. На слици су подебљане гране које припадају дрвету најкраћих путева од чвора $s \equiv V$, са кореном у чвору V. Приметимо да, рецимо, најкраћи пут до чвора H добијамо тако што граном (E, H) продужимо претходно пронађени најкраћи пут до чвора E: пут (V, B, E).

Табела 2.3: Пример извршавања Дајкстриног алгоритма за одређивање најкраћих путева из чвора V у графу са слике 2.71. .

	V	A	B	C	D	E	F	G	H
	0	∞	∞	∞	∞	∞	∞	∞	∞
V	*0*	1	5	∞	9	∞	∞	∞	∞
A	*0*	*1*	5	3	9	∞	∞	∞	∞
C	*0*	*1*	5	*3*	7	∞	12	∞	∞
B	*0*	*1*	*5*	*3*	7	8	12	∞	∞
D	*0*	*1*	*5*	*3*	*7*	8	12	11	∞
E	*0*	*1*	*5*	*3*	*7*	*8*	12	11	9
H	*0*	*1*	*5*	*3*	*7*	*8*	12	11	*9*
G	*0*	*1*	*5*	*3*	*7*	*8*	12	*11*	*9*
F	*0*	*1*	*5*	*3*	*7*	*8*	*12*	*11*	*9*

У табели су приказане промене у низу $\overline{SP}$ у ком се чувају дужине најкраћих путева од чвора v до свих осталих чворова преко чворова који припадају скупу V_k. Елементи скупа V_k (поред чвора V) су они чије су дужине најкраћих путева исписане искошено (то су коначне дужине најкраћих путева).

- *Прва врста табеле садржи иницијалне вредности низа растојања и односи се само на путеве који не садрже гране. Бира се најкраћи од тих путева и у овом случају он почиње из V и завршава се у V.*
- *Друга врста табеле односи се само на путеве који се састоје од једне гране која полази из чвора V. Бира се најкраћи од тих путева (грана) и у овом случају он води ка чвору A.*
- *Трећа врста табеле показује ажурирање дужина најкраћих путева укључујући сада све путеве који се састоје од једне гране која полази из чвора V или од две гране преко чвора A. Други најкраћи пут од чвора V води до чвора C.*
- *У сваком кораку алгоритма бира се нови, наредни најближи чвор чвору V, додаје се у скуп V_k и у наредној врсти табеле се приказују дужине тренутних најкраћих путева од чвора V до свих чворова.*

На основу табеле могуће је реконструисати и саме најкраће путеве. На пример, ако желимо да сазнамо који је најкраћи пут од чвора V до чвора H дужине 9, разматрамо колону која одговара чвору H и тражимо последњу промену вредности у тој колони – она одговара врсти означеној чвором E и то значи да је чвор E родитељ чвора H у дрвету најкраћих путева. Затим, на основу табеле одређујемо родитеља чвора E,

Гледајући колону табеле која одговара чвору E и утврђивањем на ком чвору се десила последња промена вредности у овој табели – то је чвор B те је он родитељ чвора E, док је родитељ чвора B полазни чвор V. Коначно, најкраћи пут реконструишемо читајући добијени низ родитељских чворова уназад и као најкраћи пут од чвора V до чвора H добијамо пут (V, B, E, H). Наравно, уместо да памтимо целу табелу, ефикасније је да родитељске чворове одржавамо у засебном низу.

У Дајкстрином алгоритму потребно је да $O(|V|)$ пута проналазимо дужине најкраћих путева до чворова ван скупа V_k и да често ажурирамо дужине путева. Структура података погодна за ефикасно налажење минималних елемената је ред са приоритетом. Претпоставићемо да је он реализован као мин-хип (мада може бити реализован и помоћу балансираног бинарног дрвета, као уређен скуп). Пошто је потребно да пронађемо чвор до кога је дужина пута најмања, све чворове v ван скупа V_k чувамо у реду са приоритетом, са кључевима $v.\overline{SP}$ једнаким дужинама тренутно најкраћих путева од чвора s до њих. На почетку су све дужине путева осим оне до чвора s једнаке $+\infty$, па редослед елемената у хипу није битан, сем што је чвор s на врху. Налажење чвора v_{k+1} који је међу чворовима ван V_k најближи чвору s је једноставно: он се узима са врха хипа. После тога се за сваку грану (v_{k+1}, v) проверава да ли је v ван V_k и да ли најкраћи пут до чвора v_{k+1} продужен граном (v_{k+1}, v) скраћује тренутно познати најкраћи пут до чвора v.

Међутим, кад се промени дужина пута $v.\overline{SP}$ до неког чвора v, потребно је да се промени и положај чвора v у хипу. Према томе, потребно је на одговарајући начин поправљати хип. С обзиром на то да се путеви до чвора могу само скраћивати, то значи да се вредност кључа у хипу може само смањити и евентуално при том постати мања од вредности кључа свог родитеља (пошто је претходна вредност елемента била мања од вредности његове деце у хипу, то ће важити и за смањену вредност кључа). Операција смањивања вредности кључа у мин-хипу није директно подржана у стандардној библиотеци језика C++, али се одговарајућа поправка хипа може ручно имплементирати разменом вредности елемента и његовог родитеља, све док услов хипа не буде задовољен. Ова операција се дакле може извести алгоритмом сложености $O(\log n)$, где је n број елемената у хипу. Међутим, проблем са овим поправкама је у томе што хип као структура података не подржава ефикасно проналажење задатог елемента. Наиме, тражење елемента у хипу је операција линеарне временске сложености у односу на број елемената у хипу. У ручној имплементацији бисмо могли у посебном низу чувати положај сваког чвора у хипу. Ипак, постоји једноставан начин да се ово избегне и да се употреби само основни интерфејс реда са приоритетом (додавање елемената, проналажење и брисање најмањег елемента). Наиме, уместо да се вредност растојања до неког чвора у хипу замени новом (мањом) вредношћу, врши се уметање новог чвора са истом ознаком чвора и новом (мањом) вредношћу растојања (ова техника се назива *техником лењог брисања* (енгл. lazy

deletion technique)). С обзиром на то да је нова вредност растојања мања од старе, нови чвор ће сигурно бити скинут из хипа пре старог. Остаје проблем што ће се у неком каснијем тренутку из хипа скинути и стари податак о чвору v са већом вредношћу најкраћег пута. Тај податак треба да се занемари, па је приликом узимања елемента са врха хипа потребно просто прескочити чворове који су раније били обрађени тј. који се већ налазе у скупу V_k.

Остаје бојазан да оваква имплементација може да повећа временску сложеност алгоритма. Ако бисмо вршили поправке кључева у хипу, у хипу бисмо чували у сваком тренутку највише $O(|V|)$ елемената. У имплементацији која чува копије чворова приликом обраде сваке (усмерене) гране може се додати максимално један нови елемент у хип. Дакле укупан број елемената у хипу биће сигурно мањи или једнак од $O(|V|+|E|)$, те ће свака од операција уметања елемента у хип и брисања минималног елемента из хипа бити сложености $O(\log(|V| + |E|))$. С обзиром на то да је $|E| \leq |V|^2$, и стога $\log|E| \leq 2 \cdot \log|V|$, сложености $O(\log|E|)$ и $O(\log|V|)$ се асимптотски не разликују, те ће операције над хипом који чува копије чворова бити исте асимптотске сложености као и у случају када нема копија. Наравно, меморијско заузеће може бити веће.

Дајкстрин алгоритам за налажење најкраћих путева од датог чвора приказан је у наставку.

```
// Dajkstrin algoritam za odredjivanje najkracih puteva
// iz datog polaznog cvora start do svih cvorova u grafu
void najkraciPuteviDajkstra(int start) {
   int brojCvorova = listaSuseda.size();
  // da li je cvoru odredjeno najkrace rastojanje
  vector<bool> resen(brojCvorova, false);
  // duzina trenutno poznatog najkraceg puta do cvora
  vector<Tezina> minRastojanja(brojCvorova, INF);
  // roditeljski cvor svakog cvora u drvetu najkracih puteva
  vector<Cvor> roditelji(brojCvorova, -1);

  // uredjen par koji čine rastojanje do cvora i broj cvora;
  // redosled elemenata mora biti ovakav zbog operacije poredjenja parova
  typedef pair<Tezina, Cvor> RastojanjeCvora;
  // min-hip u koji smestamo poznata rastojanja do svih cvorova
  priority_queue<RastojanjeCvora,
                 vector<RastojanjeCvora>,
                 greater<RastojanjeCvora>> rastojanja;
```

```
// ubacujemo polazni cvor u hip i postavljamo rastojanje do njega na 0
rastojanja.emplace(0, start);
minRastojanja[start] = 0;

// broj cvorova do kojih je konacno odredjeno najkrace rastojanje
int brojResenih = 0;

// dok se ne odredi najkrace rastojanje do svih cvorova
while (brojResenih < brojCvorova) {
  // izdvajamo naredni najblizi cvor
  auto [rastojanje, cvor] = rastojanja.top();
  rastojanja.pop();

  // ako je taj cvor ranije resen, preskacemo ga
  // ovo se moze desiti zbog lenjog azuriranja hipa
  if (resen[cvor])
     continue;

  // inace pamtimo da smo sada odredili najkrace rastojanje do njega
  brojResenih++;
  resen[cvor] = true;

  // stampamo informaciju o najkracem putu do tog cvora;
  // najkraci putevi se ispisuju u redosledu njihovog "otkrivanja"
  odstampajNajkraciPut(cvor, roditelji, minRastojanja);

  // prolazimo kroz sve grane iz tekuceg cvora
  for (const auto& [sused, tezina] : listaSuseda[cvor]) {
     // koje vode ka susedima kojima jos nije odredjeno najkrace rastojanje
     if (!resen[sused]) {
        // ako je put kroz tekuci cvor do suseda kraci od poznatog
        // najkraceg puta, azuriramo vrednost najkraceg puta do suseda
        // i vrednost njegovog roditeljskog cvora
        if (minRastojanja[cvor] + tezina < minRastojanja[sused]) {
           minRastojanja[sused] = minRastojanja[cvor] + tezina;
           roditelji[sused] = cvor;
           // ubacujemo element u hip;
           // ako je postojala prethodna vrednost, ne brisemo je:
           // nova vrednost ce se naci u hipu iznad stare
```

```
                rastojanja.emplace(minRastojanja[sused], sused);
            }
        }
    }
}
```

Као што смо већ поменули, операције уметања и брисања из хипа су сложености $O(\log|V|)$. Можемо имати највише $O(|V|+|E|)$ уметања у хип (у варијанти са чувањем копија чворова) и највише $O(|V|+|E|)$ брисања из хипа. Према томе, временска сложеност алгоритма је $O((|V|+|E|)\log|V|)$. Запажа се да је Дајкстрин алгоритам спорији него алгоритам који одређује најкраће путеве од задатог чвора у ацикличком графу. Ако би се уместо бинарног хипа користио Фибоначијев хип, временска сложеност Дајкстриног алгоритма била би једнака $O(|E|+|V|\log|V|)$. Напоменимо да је код густих графова, код којих је број грана асимптотски једнак $|V|^2$, уместо хипа ефикасније користити обичан низ и минимум тражити линеарном претрагом (сложеност је тада $O(|V|^2)$, док је сложеност варијанте са хипом $O(|V|^2\log|V|)$.

Тип претраге који се јавља у Дајкстрином алгоритму се назива и *претрага са приоритетом* — сваком чвору додељује се приоритет (у овом случају тренутно најмање познато растојање од полазног чвора), па се чворови обилазе редоследом који је одређен приоритетом. Кад се заврши разматрање текућег чвора, разматрају се све њему суседне гране и притом се могу променити приоритети неких других чворова. Начин извођења тих промена је детаљ по коме се једна претрага са приоритетом разликује од друге. Претрага са приоритетом је карактеристична за тежинске графове и сложенија је од обичне претраге за фактор $\log|V|$ ако се користи хип.

2.9.3 Графови са негативним гранама

Размотримо сада проблем одређивања најкраћих путева из задатог чвора у општем случају, када тежине грана могу бити и негативне. Природно је запитати се да ли негативне тежине грана имају икаквог смисла. У контексту растојања на мапама немају, међутим показује се да тежински графови чије тежине могу бити и позитивне и негативне нису неуобичајени у моделовању неких проблема из реалног света:

- На овај начин можемо моделовати проблем одржавања салда на рачуну: чворови графа одговарају рачунима, гране трансакцијама над рачунима, при чему извршавање трансакције може довести до зараде – коју ћемо моделовати граном позитивне тежине или до губитка – који ћемо моделовати граном негативне тежине.
- Проблем путовања из једног града у други, при чему је могуће на неком од делова пута покупити неког путника и зарадити одређену количину новца, може се

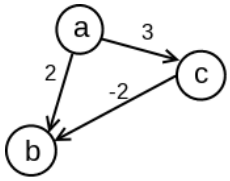
моделовати на овај начин. Циљ је стићи из једног града у други са највише новца на крају. Овај проблем можемо моделовати у виду проблема тражења најкраћег пута у графу чије тежине грана могу бити позитивне и одговарају трошковима путовања између та два града (ако на том делу пута путујемо сами) или негативне (ако на неком делу пута покупимо једног или више путника и од њих зарадимо више новца него што су трошкови тог дела пута).

- Налажење пута са најмањим производом тежина грана се може свести на налажење пута са најмањим збиром тежина грана у графу у ком је тежина сваке гране замењена њеним логаритмом (јер је логаритам монотона функција, а логаритам производа једнак је збиру логаритама). Приметимо да након логаритмовања неке тежине грана могу постати негативне, па се поново сусрећемо са проблемом најкраћег пута у графу са негативним тежинама грана.

Ако граф садржи неку грану негативне тежине, Дајкстрин алгоритам не гарантује да ћемо до сваког чвора одредити заиста најкраће путеве.

Пример 2.9.5

Размотримо проблем одређивања најкраћих путева из чвора a до свих осталих чворова у графу приказаном на слици 2.72. Ако бисмо применили Дајкстрин алгоритам, он би најпре одредио најкраћи пут до чвора b (као чвора који је од свих чворова до којих постоји грана из a најближи чвору a) и прогласио би да је он дужине 2, а након тога би одредио најкраћи пут до чвора c као другог најближег чвора чвору a и прогласио би да је дужина најкраћег пута до њега 3 чиме би се се извршавање алгоритма завршило. Јасно је да најкраћи пут од чвора a до чвора b води преко чвора c и дужине је 1, међутим Дајкстрин алгоритам не би разматрао овај пут.

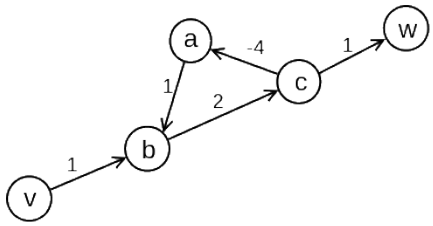


Слика 2.72: Пример графа који садржи грану негативне тежине и за који Дајкстрин алгоритам покренут из чвора a не рачуна добро најкраћа растојања.

У наставку текста ћемо често претпостављати да граф не садржи циклус негативне тежине. Овај услов је природан јер иначе до неких чворова не мора да постоји најкраћи пут (јер пут можемо увек додатно скратити још једним проласком кроз циклус). Можемо дефинисати да је најкраће растојање до таквих чворова $-\infty$ (иако је пут увек коначан, па му дужина не може бити $-\infty$).

Пример 2.9.6

Размотримо граф са слике 2.73: најкраћи пут од чвора v до чвора w се не може одредити јер сваки пролазак кроз циклус (b, c, a, b) смањује дужину пута за 1 (дужина најкраћег пута је $-\infty$).



Слика 2.73: Пример графа који садржи циклус негативне тежине.

2.9.3.1 Општи алгоритам заснован на релаксацији грана

Основни корак у алгоритму заснованом на релаксацијама грана је исти као и у Дајкстрином алгоритму и у алгоритму за одређивање најкраћих путева у ацикличким графовима, а то је тзв. *релаксација гране* (u, v), односно провера да ли је вредност $v.\overline{SP}$ тренутно најкраћег утврђеног растојања од почетног чвора s до чвора v већа од вредности $u.\overline{SP} + d(u, v)$; ако је то тачно, кажемо да је релаксација успешна, ажурира се вредност $v.\overline{SP}$ и памти да најкраћи пут до чвора v води кроз чвор u.

Заправо, сви алгоритми за решавање овог проблема могу се подвести под следећу општу схему приказану у алгоритму 7.

Алгоритам 7 Општи алгоритам релаксације грана

1: $s.\overline{SP} = 0$, $v.\overline{SP} = +\infty$, za sve $v \neq s$
2: **while** postoji grana (u, v) takva da je $v.\overline{SP} > u.\overline{SP} + d(u, v)$ **do**
3: $\quad v.\overline{SP} = u.\overline{SP} + d(u, v)$

Ако у графу постоји негативан циклус, овај општи алгоритам се не зауставља, а ако нема негативних циклуса алгоритам се зауставља и исправно израчунава најкраћа растојања до свих чворова (што ћемо ускоро и доказати). Приметимо да није прецизиран редослед обраде грана и конкретни алгоритми се разликују по томе којим редом и колико пута обрађују гране (на пример, Дајкстрин алгоритам обрађује чворове у редоследу неопадајућег растојања и редом релаксира гране које из њих излазе). Обично је редослед обраде такав да се под одређеним претпоставкама о графу унапред зна да је достигнуто коначно решење, тј. такав да се не врши ефективна провера да ли постоје гране које се могу релаксирати.

Докажимо заједничка својства алгоритама заснованих на релаксацији грана. Иако неке од наредних лема важе и у случају када у графу постоји негативни циклус, то нам неће бити потребно у анализи алгоритама, тако да ћемо у свим наредним лемама разматрати само графове без негативних циклуса (осим ако је наглашено другачије).

Лема 2.9.1 **[Неједнакост троугла]**

Нека је дат тежински граф $G = (V, E)$ без негативних циклуса и нека $x.SP$ означава најкраће растојање од чвора $s \in V$ до чвора $x \in V$. За сваку грану $(u, v) \in E$ важи $u.SP + d(u, v) \geq v.SP$.

Доказ. Не може да важи $v.SP > u.SP + d(u, v)$, јер би тада постојао пут од s до u на који би се могла надовезати грана (u, v), чиме би се добио пут од s до v који би био краћи од најкраћег пута од s до v, што је контрадикција. □

Јасно је да се релаксацијом само може смањити процена растојања $v.\overline{SP}$. Наредна лема тврди да та процена никада не може бити мања од стварне вредности најкраћег растојања $v.SP$.

Лема 2.9.2 **[Доња граница процене растојања]**

Нека је дат тежински граф $G = (V, E)$ без негативних циклуса и нека за сваки чвор $v \in V$, $v.SP$ означава најкраће растојање од чвора s до чвора v, а $v.\overline{SP}$ текућу процену тог растојања током извршавања поступка релаксације грана (у произвољном редоследу). Током читавог тог процеса за сваки чвор v важи $v.\overline{SP} \geq v.SP$.

Доказ. Тврђење доказујемо индукцијом по броју релаксираних грана. Базу индукције чини случај иницијализације. За све чворове $v \neq s$ мора да важи $v.SP \leq v.\overline{SP} = +\infty$. Пошто граф нема негативни циклус који садржи s, важи $s.\overline{SP} = s.SP = 0$.

Претпоставимо да тврђење важи након k релаксираних грана и докажимо да важи и након релаксирања додатне гране (u, v). За све чворове $x \neq v$ не мења се вредност $x.\overline{SP}$, па на основу индуктивне хипотезе важи $x.\overline{SP} \geq x.SP$. За чвор v након релаксације важи $v.\overline{SP} = u.\overline{SP} + d(u, v)$. На основу индуктивне хипотезе важи $u.\overline{SP} \geq u.SP$, па је $v.\overline{SP} \geq u.SP + d(u, v)$, међутим, на основу неједнакости троугла (леме 2.9.1) важи $u.SP + d(u, v) \geq v.SP$, па важи $v.\overline{SP} \geq v.SP$. □

Јасно је да када једном процена $v.\overline{SP}$ достигне вредност $v.SP$ она све време надаље остаје једнака тој вредности (јер релаксација само може смањити вредност $v.\overline{SP}$, а на основу претходне леме знамо да она не може бити мања од $v.SP$).

Наредна лема нам гарантује да ће након релаксација грана из чвора u, за који је већ израчуната вредност најкраћег растојања од почетног чвора s (тј. важи $u.\overline{SP} = u.SP$), бити исправно израчунате вредности најкраћег растојања оних његових суседа v за које се најкраћи пут завршава граном (u, v).

Лема 2.9.3 **[Релаксација последње гране на најкраћем путу]**

Нека је дат тежински граф $G = (V, E)$ без негативних циклуса и нека је (u, v) последња грана на најкраћем путу од $s \in V$ до $v \in V$. Након што се у тренутку када важи $u.\overline{SP} = u.SP$ та грана релаксира или се установи да се не може релаксирати, важи $v.\overline{SP} = v.SP$.

Доказ. Пошто је (u, v) последња грана на најкраћем путу до v, важи да је $v.SP = u.SP + d(u, v)$. Зато је $v.\overline{SP} \geq v.SP = u.SP + d(u, v) = u.\overline{SP} + d(u, v)$. Након што се грана (u, v) релаксира или се установи да то није могуће, важи да је вредност $v.\overline{SP}$ минимум старе вредности $v.\overline{SP}$ и вредности $u.\overline{SP} + d(u, v)$. Зато важи $v.\overline{SP} = u.\overline{SP} + d(u, v) = u.SP + d(u, v) = v.SP$. □

Наредна лема нам гарантује да ако не постоји грана која се може релаксирати, тада су сва најкраћа растојања исправно израчуната.

Лема 2.9.4 **[Услов за постојање гране која се може релаксирати]**

Нека је дат тежински граф $G = (V, E)$ без негативних циклуса и нека за неки чвор $v \in V$ важи $v.\overline{SP} \neq v.SP$. Тада постоји грана $e \in E$ која се може релаксирати.

Доказ. Нека је $(s \equiv v_1, v_2, \ldots, v_n = v)$ најкраћи пут од почетног чвора до чвора v и нека је v_i први чвор на том путу за који је $v_i.\overline{SP} \neq v_i.SP$. Пошто за чвор $v_1 \equiv s$ све време важи $s.\overline{SP} = s.SP = 0$, мора важити $i > 0$. Ако се грана (v_{i-1}, v_i) не би могла релаксирати, на основу леме 2.9.3 морало би да важи $v_i.\overline{SP} = v_i.SP$, што је контрадикција. Дакле, постоји грана која се може релаксирати. □

Сада је једноставно доказати коректност опште схеме релаксације грана.

Теорема 2.9.3 **[Заустављање и коректност општег алгоритма]**

Нека је дат тежински граф $G = (V, E)$ у коме нема негативних циклуса. Тада се алгоритам 7 зауставља и након заустављања за сваки чвор графа $v \in V$ важи $v.\overline{SP} = v.SP$.

Доказ. Пошто се све вредности $v.\overline{SP}$ смањују током извршавања алгоритма (у свакој релаксацији неке гране строго се смањи вредност $v.\overline{SP}$ за неки чвор v), а на основу леме

2.9.2 су оне ограничене одоздо, у једном тренутку све вредности $v.\overline{SP}$ ће достићи своје границе $v.SP$. То је сасвим јасно ако су у питању гране целобројне тежине, јер је збир свих вредности $v.\overline{SP}$ опадајући низ целих бројева који је одоздо ограничен збиром свих вредности $v.SP$, па мора бити коначан. Ако су гране рационални бројеви, могао би да постоји бесконачно дугачак опадајући низ рационалних бројева који је одоздо ограничен. Међутим, множењем свих тежина неким погодно одабраним степеном броја 10 може се лако постићи да тежине свих грана постану целобројне, без суштинске измене проблема, тако да је и у том случају јасно да се алгоритам мора зауставити након коначног броја корака.

Ако би се тада могла релаксирати нека грана (u, v), важило би да је $v.\overline{SP} > u.\overline{SP} + d(u, v)$, тј. $v.SP > u.SP + d(u, v)$, што је немогуће на основу неједнакости троугла (лема 2.9.1). Дакле, тада нема грана које се могу релаксирати, па се алгоритам зауставља.

Пошто се алгоритам заустави, не постоји грана која се може релаксирати. Ако би за неки чвор $v \in V$ важило $v.\overline{SP} \neq v.SP$, на основу леме 2.9.4 постојала би грана која би се могла релаксирати, што је контрадикција. □

2.9.3.2 Белман-Фордов алгоритам

Ако граф има негативне тежине грана, али не садржи циклус негативне тежине, најкраћи путеви из датог чвора s могу се одредити *Белман-Фордовим алгоритмом*. Важи и више од тога: ако у графу постоји циклус негативне тежине, Белман-Фордов алгоритам то детектује. Ричард Белман[17] и Лестер Форд[18] су објавили овај алгоритам 1958. и 1959. године, међутим, сматра се да је до њега први дошао Алфонсо Шимбел 1955. године.

У извођењу овог алгоритма биће нам значајна наредна лема, која гарантује да ће након што се *редом* релаксира једна по једна грана најкраћег пута до неког чвора успешно бити израчунато најкраће растојање до тог чвора (као и до свих осталих чворова на том путу).

Лема 2.9.5 **[Релаксација најкраћег пута]**

Нека је дат тежински граф $G = (V, E)$ без негативних циклуса и нека је $p = (s \equiv v_1, \ldots, v_j)$ најкраћи пут од s до v_j. Произвољни низ релаксација грана који укључује и редом релаксације грана (v_1, v_2), (v_2, v_3), …, (v_{j-1}, v_j) доводи до тога да важи $v_j.\overline{SP} = v_j.SP$. Пре, после и између ових релаксација је допуштено вршити било које друге релаксације грана.

[17] Ричард Белман (енгл. Richard Bellman), (1920-1984), амерички примењени математичар.
[18] Лестер Форд (енгл. Lester Ford), (1927-2017), амерички математичар.

Доказ. Тврђење доказујемо индукцијом по чворовима у низу p тј. доказујемо да за свако $1 \leq k \leq j$ након релаксација грана (v_1, v_2), (v_2, v_3), ..., (v_{k-1}, v_k) важи $v_k.\overline{SP} = v_k.SP$.

База индукције је чвор s. Пошто граф нема негативних циклуса, важи $s.SP = 0$. Вредност $s.\overline{SP}$ мора све време бити 0, јер је, знамо на основу леме 2.9.2, увек већа или једнака од $s.SP$, током релаксације се само може смањити, а иницијализује се на 0.

Претпоставимо да смо редом релаксирали гране (v_1, v_2), (v_2, v_3), ..., (v_{k-2}, v_{k-1}) и да је $v_{k-1}.\overline{SP} = v_{k-1}.SP$. У неком тренутку након тога се покушава релаксација гране (v_{k-1}, v_k). На основу леме 2.9.3 следи да након тога сигурно важи $v_k.\overline{SP} = v_k.SP$. На основу леме 2.9.2 важи $v_k.\overline{SP} \geq v_k.SP$, па се ова вредност не може даље смањивати и неће се мењати. □

Дакле, довољно је да наш алгоритам обезбеди да се *све гране свих најкраћих путева релаксирају редом од прве до последње*. Белман-Фордов алгоритам то постиже тако што укупно $|V|-1$ пута произвољним редоследом извршава релаксацију свих грана у графу.

Алгоритам 8 Белман-Фордов алгоритам

```
1: s.SP̄ = 0, v.SP̄ = +∞, za sve v ≠ s
2: for i = 1..|V| − 1 do
3:     for all (u, v) ∈ E do
4:         if v.SP̄ > u.SP̄ + d(u, v) then
5:             v.SP̄ = u.SP̄ + d(u, v)
```

Тврдимо да су, ако граф не садржи негативан циклус, након тога коректно израчунати најкраћи путеви од чвора v до свих чворова у графу. За чворове који су недостижни из почетног чвора, дужина најкраћег пута остаје $+\infty$. Ако граф садржи негативан циклус, након свих ових релаксација и даље постоје гране које се могу релаксирати (провера постојања такве гране је уједно и начин да се провери да ли постоји негативан циклус, тј. да ли су пронађена растојања коректна).

Докажимо ово тврђење.

Лема 2.9.6

Нека је дат тежински граф $G = (V, E)$ без негативних циклуса. Када се Белман-Фордов алгоритам (алгоритам 8) примени на граф G, за сваки чвор v важи $v.\overline{SP} = v.SP$.

Доказ. Ако је v достижан из s у графу без негативних циклуса мора да постоји прост пут од s до v. У том путу не може да буде више од $|V|$ чворова, па ни више од $|V| - 1$

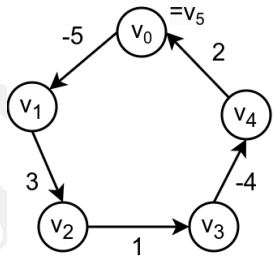
грана. Пошто се у свакој итерацији релаксирају све гране, у првој итерацији је сигурно релаксирана прва грана на том путу, у другој друга итд. (наравно могуће је да се у некој итерацији редом релаксира и више грана најкраћег пута). Пошто се врши $|V|-1$ итерација, све гране су редом релаксиране, па на основу леме 2.9.5 важи $v.\overline{SP} = v.SP$.

Ако v није достижан из s тада је $v.\overline{SP} = v.SP = +\infty$. Заиста, вредност $v.\overline{SP}$ се иницијализује на $+\infty$, а не може да се смањи испод стварне вредности $v.SP$ која је једнака $+\infty$. □

Теорема 2.9.4 **[Заустављање и коректност Белман-Фордовог алгоритма]**

Ако у графу нема циклуса негативне тежине, Белман-Фордов алгоритам се зауставља. Након заустављања за све чворове графа $v \in V$ важи $v.\overline{SP} = v.SP$ и ниједна грана се не може релаксирати. Ако у графу постоји циклус негативне тежине, по завршетку алгоритма постоји бар једна грана која се може релаксирати.

Доказ. Први део тврђења директно следи из леме 2.9.6. За сваку грану (u, v) на основу неједнакости троугла важи $v.\overline{SP} = v.SP \leq u.SP + d(u, v) = u.\overline{SP} + d(u, v)$, па се грана (u, v) не може релаксирати.



Слика 2.74: Циклус негативне тежине.

Претпоставимо да G садржи циклус негативне тежине $c = (v_0, v_1, \ldots, v_k), v_k = v_0$, достижан из чвора s (слика 2.74). Тада је:

$$\sum_{i=1}^{k} d(v_{i-1}, v_i) < 0 \tag{2.3}$$

Претпоставимо супротно, да се ниједна грана не може релаксирати, тј. да за сваку грану $(u, v) \in E$ важи $v.\overline{SP} \leq u.\overline{SP} + d(u, v)$. Тада и за сваку грану циклуса $(v_{i-1}, v_i) \in E, i = 1, \ldots, k$, важи: $v_i.\overline{SP} \leq v_{i-1}.\overline{SP} + d(v_{i-1}, v_i)$. Сабирањем ових неједнакости за све гране циклуса добија се:

$$\sum_{i=1}^{k} v_i.\overline{SP} \leq \sum_{i=1}^{k} v_{i-1}.\overline{SP} + \sum_{i=1}^{k} d(v_{i-1}, v_i)$$

Пошто је $v_0 = v_k$, важи:

$$\sum_{i=1}^{k} v_i.\overline{SP} = \sum_{i=1}^{k} v_{i-1}.\overline{SP}$$

те даље важи:

$$\sum_{i=1}^{k} d(v_{i-1}, v_i) \geq 0$$

супротно претпоставци (2.3). □

Приметимо да се релаксација грана не мора вршити истим редоследом у свакој од итерација – само је битно да се у свакој итерацији релаксирају све гране графа. Тиме се постиже да се на сваком најкраћем путу сигурно релаксира по једна додатна грана.

Приметимо да се након извршавања i-те итерације петље, проналазе најкраћи путеви чији број грана не превазилази вредност i.

С обзиром да се често може десити да се најкраћа растојања израчунају и пре спровођења свих итерација, алгоритам је могуће прекинути раније ако се деси да се у некој итерацији не успе релаксација ниједне гране.

Имплементација овог алгоритма је веома једноставна.

```
// funkcija koja koriscenjem Belman-Fordovog algoritma
// racuna najkrace puteve do svih cvorova
void najkraciPuteviBelmanFord(int start) {
  int brojCvorova = listaSuseda.size();
  // duzina trenutno poznatog najkraceg puta do cvora
  vector<Tezina> minRastojanja(brojCvorova, INF);
  // prethodnik cvora na najkracem putu
  vector<Cvor> roditelji(brojCvorova, -1);

  // postavljamo rastojanje do polaznog cvora
  minRastojanja[start] = 0;
```

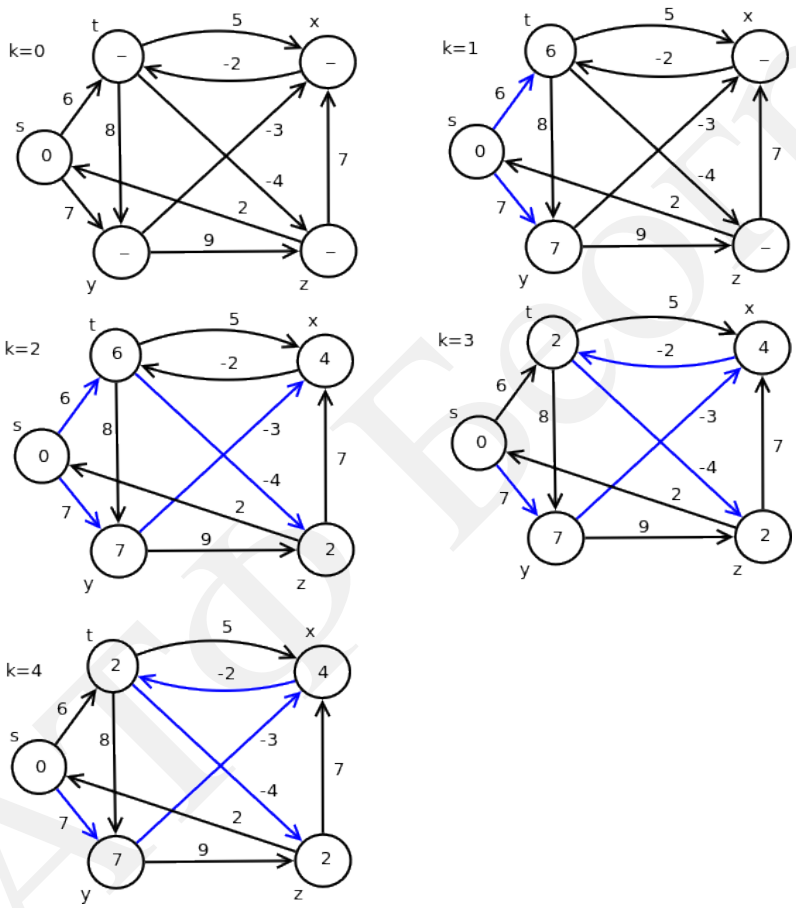
```
  // |V|-1 put prolazimo kroz skup svih grana
  for (int k = 0; k < brojCvorova - 1; k++) {
    bool biloRelaksacija = false;
    for (Cvor cvor = 0; cvor < brojCvorova; cvor++)
      for (const auto& [sused, tezina] : listaSuseda[cvor]) {
         // ukoliko je potrebno vrsimo relaksaciju grane
         if (minRastojanja[cvor] + tezina < minRastojanja[sused]) {
             minRastojanja[sused] = minRastojanja[cvor] + tezina;
             // i postavljamo koji cvor mu prethodi na najkracem putu
             roditelji[sused] = cvor;
             biloRelaksacija = true;
         }
      }
    if (!biloRelaksacija) break;
  }

  // ukoliko i dalje postoji put koji je moguce skratiti
  // onda graf sadrzi ciklus negativne duzine
  for (Cvor cvor = 0; cvor < brojCvorova; cvor++)
    for (const auto& [sused, tezina] : listaSuseda[cvor])
      if (minRastojanja[cvor] + tezina < minRastojanja[sused]) {
         cout << "Graf sadrzi ciklus negativne duzine" << endl;
         return;
      }

  // stampamo najkrace puteve do svih cvorova u grafu
  for (Cvor cvor = 0; cvor < brojCvorova; cvor++)
    odstampajNajkraciPut(cvor, roditelji, minRastojanja);
}
```

Сложеност Белман-Фордовог алгоритма износи $O(|V| \cdot |E|)$, јер спољашњом `for` петљом пролазимо $O(|V|)$ пута, а у свакој итерацији ове петље пролазимо скупом свих грана у графу.

Пример 2.9.7

На слици 2.75 илустровано је извршавање Белман-Фордовог алгоритма.

Граф има 5 чворова, те се алгоритам састоји од 4 итерације, за $k = 1, 2, 3, 4$. Прва слика одговара иницијализацији, а наредне слике појединачним пролазима



Слика 2.75: Пример извршавања Белман-Фордовог алгоритма.

кроз све гране. Вредности тренутно најкраћих растојања приказане су унутар чворова, а гране плаве боје воде од претходника чворова на (тренутно) најкраћим путевима: ако је грана (u, v) плаве боје, онда се до чвора v најкраћим путем долази преко чвора u. У сваком пролазу гране се релаксирају у наредном редоследу: $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$.

Белман-Фордов алгоритам се може унапредити тако што се смањи број релаксација које се разматрају у свакој итерацији алгоритма. Наиме, ако се тренутно најкраће растојање до чвора v није изменило од тренутка када су последњи пут релаксиране гране које полазе из чвора v, онда нема потребе поново вршити релаксацију грана које полазе из чвора v. На овај начин се број грана које треба релаксирати у свакој итерацији потенцијално смањује. Овај алгоритам познат је под називом *бржи алгоритам најкраћег пута* (енгл. Shortest Path Faster Algorithm – SPFA). Алгоритам се може имплементирати тако што се одржава ред у коме се чувају чворови из којих полазе гране које треба релаксирати. У почетку се само почетни чвор ставља у ред. Када се чвор извади из реда, покушава се релаксација свих грана које из њега полазе и ако се нека грана релаксира, њен крајњи чвор се додаје у ред (ако се већ не налази у њему).

Алгоритам 9 Алгоритам SPFA

```
1: s.SP = 0, v.SP = +∞, za sve v ≠ s
2: Q.push(s)
3: while Q nije prazan do
4:     u = Q.pop()
5:     for all (u, v) ∈ E do
6:         if v.SP > u.SP + d(u, v) then
7:             v.SP = u.SP + d(u, v)
8:             if v nije u Q then
9:                 Q.push(v)
```

Коректност се заснива на чињеници да се на сваком улазу у петљу `while` у реду налазе сви чворови из којих је могућа релаксација грана (што се може доказати индукцијом). Ако се ред испразни релаксације нису могуће и сва најкраћа растојања су исправно одређена. Ако у графу постоји циклус негативне тежине, релаксације су увек могуће и алгоритам се не зауставља.

Временска сложеност алгоритма је у најгорем случају иста као сложеност Белман-Фордовог алгоритма и износи $O(|V| \cdot |E|)$, међутим просечна сложеност на насумично генерисаним графовима је $O(|E|)$.

Задатак: Најмање напорна стаза

Мапа једног предела је представљена правоугаоном матрицом поља тако је за свако поље позната надморска висина тог дела предела. Планинари током кретања са сваког поља могу да прелазе само на неко од највише четири њему суседних поља и, пошто носе пуно терета, на путу не могу да савладају велике висинске разлике. Тежина неког пута се дефинише као највећа апсолутна вредност висинске разлике између нека два суседна поља на том путу. Написати програм који одређује најмању тежину неког пута који повезује горње лево и доње десно поље на мапи.

Опис улаза

Са стандардног улаза се уносе димензије матрице m и n, а затим и матрица димензије $m \times n$ која садржи надморске висине (природне бројеве између 0 и 1000).

Опис излаза

На стандардни излаз исписати најмању тежину пута.

Пример 1

Улаз	*Излаз*	*Објашњење*

`3 3`
`1 2 2`
`3 8 2`
`5 4 5`
	`2`	Најповољнији пут је онај где планинари прелазе преко поља висине 1, 3, 5, 4, 5. На том путу савладавају редом разлике од 2, 2, 1 и 1 метар, па је тежина пута једнака 2.

Пример 2

Улаз	*Излаз*	*Објашњење*

`5 5`
`1 2 1 1 1`
`1 2 1 2 1`
`1 2 1 2 1`
`1 2 1 2 1`
`1 1 1 2 1`
	`0`	Најлакши пут је онај где се све време крећу пољима чија је висина 1.

Решење

Дајкстрин алгоритам

Задатак се може решити једноставном модификацијом Дајкстриног алгоритма за проналажење најкраћег пута од горњег левог поља, па до свих осталих поља, укључујући и последње, доње десно поље. Једина разлика у односу на уобичајени Дајкстрин алгоритам је начин како се ажурира вредност најкраћег растојања. Ако је познато најмање растојање r_{uv} од чвора u до чвора v (највећа разлика висина на најбољем путу од u до v) и растојање r_{vw} од чвора v до њему суседног чвора w (разлика висина), тада је

најмање растојање од чвора u до чвора w једнако $\max(r_{uv}, r_{vw})$. Вредност растојања се ажурира на основу доделе $r_{uw} := \min(r_{uw}, \max(r_{uv}, r_{vw}))$.

„Тежина" гране (u, v) је у овом примеру дефинисана као разлика висина два чвора које та грана спаја. „Дужина" пута $(v_0, v_1, \ldots, v_k)$ је овде дефинисана као максимална разлика висина чворова на путу. „Збир" дужине два пута једнак је њиховом максимуму. Ова метрика задовољава следећа својства:

- *Ненегативност:* тежина сваке гране је ненегативна (јер се гледа апсолутна разлика висина).
- *Усмереност:* „дужина" пута који се састоји од једне усмерене гране једнака је „тежини" те гране (што директно следи из дефиниција).
- *Неједнакост троугла:* За свака три чвора u, v, w, „дужина" најкраћег пута од u до w је мања или једнака „збиру" „дужине" најкраћег пута од u до v и „тежине" гране v до w. Заиста, „дужина" пута од u до w преко чвора v је максимум највеће разлике висина на путу од u до v и разлике висина чворова v и w. Немогуће је да је најмања могућа разлика висина на путу од u до w већа од овог броја (јер увек можемо проћи путем преко v).

Може се доказати да су ова својства довољна да осигурају коректност Дајкстриног алгоритма.

```
int najmanjeNapornaStaza(const vector<vector<int>>& visine) {
  // dimenzije matrice
  int m = visine.size(), n = visine[0].size();
  // rastojanje od početnog do svakog drugog čvora pamtimo u matrici
  vector<vector<int>> rastojanje(m, vector<int>(n, INF));
  // podatak da li je do čvora određeno rastojanje pamtimo u matrici
  vector<vector<bool>> resen(m, vector<bool>(n, false));
  // red sa prioritetom koji koji sadrži trojke na čijem je prvom mestu
  // dužina grane, a zatim čvor iz kog ta grana polazi (njegove koordinate)
  priority_queue<tuple<int, int, int>,
                 vector<tuple<int, int, int>>,
                 greater<tuple<int, int, int>>> pq;
  // rastojanje do početnog čvora je 0
  rastojanje[0][0] = 0;

  // krećemo od početnog čvora
  pq.emplace(0, 0, 0);
  while (!pq.empty()) {
    // skidamo element sa vrha reda sa prioritetom
```

```
    auto [tezina, v, k] = pq.top();
    pq.pop();
    // ako je do čvora na poziciji (v, k) ranije određen najkraći put,
    // taj čvor preskačemo
    if (resen[v][k])
      continue;
    // pamtimo da smo upravo odredili najkraći put do čvora na poziciji (v, k)
    resen[v][k] = true;

    // prolazimo kroz sve susede ovog čvora
    int dv[] = {-1, 1, 0, 0};
    int dk[] = {0, 0, -1, 1};
    for (int i = 0; i < 4; i++) {
      int vv = v + dv[i];
      int kk = k + dk[i];
      if (!(0 <= vv && vv < m && 0 <= kk && kk < n))
        continue;

      // do kojih je najkraće rastonja još nepoznato
      if (resen[vv][kk])
        continue;

      // težina grane od trenutnog do susednog čvora
      int razlika = abs(visine[vv][kk] - visine[v][k]);
      // ažuriramo rastojanje do suseda ako je manje od dosadašnjeg
      int novoRastojanje = max(rastojanje[v][k], razlika);
      if (novoRastojanje < rastojanje[vv][kk]) {
        rastojanje[vv][kk] = novoRastojanje;
        pq.emplace(rastojanje[vv][kk], vv, kk);
      }
    }
  }
  // vraćamo najkraće rastojanje do ciljnog polja
  return rastojanje[m-1][n-1];
}
```

Задатак: Мењачница

У мењачници је могућа трговина валутама $v_0, v_1, \ldots, v_{n-1}$. Курс k_{ij} одређује количину јединица валуте v_j коју је могуће добити за јединицу валуте v_i. Написати програм који одређује највећи могући износ валуте v_B који је могуће добити за једну јединицу валуте v_A, при чему се замена може вршити и посредно, коришћењем других валута. Програм уједно треба да одреди и да ли је могућа тзв. арбитража, тј. да ли је мењачница лоше одредила курсеве и омогућила да се једна јединица неке валуте посредним заменама замени за више од једне јединице те валуте.

Опис улаза

Са стандардног улаза се уноси број валута n ($1 \leq n \leq 100$) и матрица димензије $n \times n$ која садржи све курсеве. На дијагонали матрице су уписане јединице. Вредност 0 на пољу k_{ij} означава да директна замена валуте v_i у валуту v_j није могућа. Након тога се учитавају бројеви A и B ($0 \leq A, B < n$).

Опис излаза

На стандардни излаз исписати износ највећи број јединица валуте v_B који се може добити продајом валуте v_A, односно -1 ако је могућа арбитража.

Пример

Улаз	*Излаз*
4 1.00 0.97 0.65 0.00 1.03 1.00 0.64 0.73 0.00 0.00 1.00 1.16 1.28 0.00 0.00 1.00 1 4	0

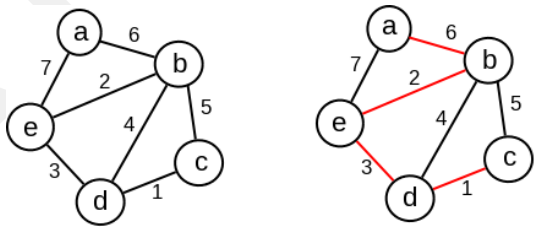
Решење

Проблем можемо моделовати тежинским графом у ком су валуте чворови, а курсеви тежине грана (гране постоје између свих парова различитих чворова за које је курс строго позитиван). Проблем се тада своди на одређивање пута на највећим производом тежина грана. Алгоритми за проналажење најкраћих путева проналазе пут са најмањим збиром тежина грана. Да би се производ свео на збир, могуће је употребити логаритамску функцију (јер важи $\log(xy) = \log x + \log y$). Она се може применити на сваку тежину гране (јер су курсеви позитивни). Приметимо да се након логаритмовања могу добити и позитивне и негативне тежине грана. Ако је основа логаритма већа од 1, логаритам је растућа функција, па се највећи производ добија када је највећи збир. Ми умемо да нађемо најкраћи пут (када је збир најмањи), па зато треба да тежине грана заменимо логаритмом за неку основу која је мања од 1 (или, еквивалентно, да узмемо

негативне вредности логаритама за основу већу од 1). Дакле, ако тежину сваке гране k_{ij} заменимо, на пример, вредношћу $-\ln k_{ij}$, тада се полазни проблем своди на одређивање најкраћег пута у графу у ком тежине грана могу бити и негативне и он може бити решен Белман-Фордовим алгоритмом. Ако постоји арбитража, тј. низ валута тако да је $k_{i_0,i_1} \cdot k_{i_1,i_2} \cdot \ldots \cdot k_{i_m,i_0} > 1$, тада је $-\ln k_{i_0,i_1} - \ln k_{i_1,i_2} - \ldots - \ln k_{i_m,i_0} < 0$, тј. у трансформисаном графу постоји циклус негативне тежине, што се опет може установити Белман-Фордовим алгоритмом.

2.10 Минимално повезујуће дрво

Размотримо систем рачунара које треба повезати оптичким кабловима тако да постоји веза између свака два рачунара. Познати су трошкови постављања кабла између свака два рачунара. Циљ је пројектовати мрежу оптичких каблова тако да укупна цена мреже буде минимална. Систем рачунара се може моделовати неусмереним графом, чији чворови одговарају рачунарима, а гране – потенцијалним везама између рачунара, са одговарајућим (позитивним) ценама. Тада се проблем своди на проналажење повезаног подграфа (са гранама које одговарају постављеним оптичким кабловима) који садржи све чворове, такав да му укупна сума тежина грана буде минимална (слика 2.76). Није тешко видети да тај подграф мора да буде дрво. Наиме, ако би подграф имао циклус, онда би се из циклуса могла уклонити произвољна грана, чиме би се добио подграф који је и даље повезан, а има мању укупну тежину. Тражени подграф зове се *минимално повезујуће (разапињуће) дрво* (енгл. minimum–cost spanning tree, MCST) и има много примена. Сличан претходном проблему је и проблем повезивања N градова аутопутевима, тако да постоји пут између свака два града, а да укупна цена изградње аутопута буде минимална могућа. Минимално повезујуће дрво има примену и у конструкцији приближних алгоритама, на пример, за решавање проблема трговачког путника.



Слика 2.76: Неусмерени повезани тежински граф и његово минимално повезујуће дрво.

Дакле, проблем којим се бавимо је конструкција ефикасног алгоритма за налажење минималног повезујућег дрвета у датом неусмереном тежинском графу.

Проблем

За задати неусмерени повезани тежински граф $G = (V, E)$ конструисати неко повезујуће дрво T минималне тежине.

Ако су све гране графа међусобно различите тежине, минимално повезујуће дрво је јединствено одређено (што не мора бити случај када постоје гране исте тежине).

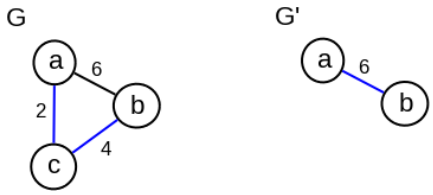
Постоји више различитих алгоритама за одређивање минималног повезујућег дрвета датог графа: ми ћемо размотрити два – Примов и Краскелов алгоритам.

2.10.1 Индукција по броју чворова и грана

Приликом конструкције графовских алгоритама природно је покушати индукцијом по броју чворова или по броју грана. На пример, проналажење најкраћих путева из једног чвора у ацикличном графу као и Дајкстрин алгоритам се могу описати тако што се из графа избаци један чвор (последњи у одговарајућем поретку), рекурзивно реши смањени проблем и на крају дода избачени чвор. Наравно, имплементација тече итеративно додајући један по један чвор, при чему се чворови морају обрађивати у одговарајућем редоследу.

Нажалост, такав приступ се не може једноставно прилагодити решавању проблема минималног повезујућег дрвета. Наиме, у општем случају не постоји неки природан редослед у ком би се чворови обрађивали. Избачени чвор може бити артикулациона тачка и разбити граф на неповезане компоненте тако да се не добије проблем истог облика. Можемо одредити минималну повезујућу шуму за посебне компоненте, али то компликује алгоритам. Чак и када се избацивањем чвора добије повезан граф, није јасно како би убацивање новог чвора утицало на пронађено минимално повезујуће дрво мањег графа. Наиме, немамо гаранцију да је минимално повезујуће дрво подграфа поддрво минималног повезујућег дрвета целог графа (слика 2.77). Да би убачени чвор био повезан са осталима, сигурно је потребно да нека грана која га спаја са суседима буде укључена у крајње дрво. Међутим, могуће је да уместо само једне такве гране у дрво треба убацити и више њих, а избацити неке гране раније конструисаног поддрвета. Анализа овог приступа постаје компликована и одустаћемо од ње, јер, као што ћемо видети постоје једноставнији, веома ефикасни алгоритми.

До сличног закључка се долази и ако се покуша индукција по броју грана. За разлику од чворова међу којима не можемо унапред успоставити неки природан редослед, гране се природно могу уредити по тежини и можемо утицати на то коју грану ћемо избацити, рекурзивно решити потпроблем и на крају је поново вратити у дрво. Важи (што ћемо касније и формално доказати) да најкраћа грана графа мора бити укључена у минимално повезујуће дрво. Зато је природно њу уклонити из графа, решити рекурзивно потпроблем (применити индуктивну хипотезу) и на крају је укључити. Да ли је ово

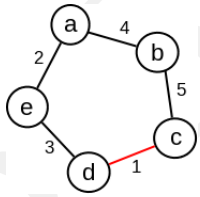


Слика 2.77: Пример када минимално повезујуће дрво подграфа G' графа добијеног избацивањем једног чвора из графа G није подграф минималног повезујућег дрвета графа G.

регуларна примена индукције?

Први проблем у оваквој примени индукције је у томе што после уклањања гране, добијени проблем није еквивалентан полазном. Наиме, избор једне гране ограничава могућности избора других грана.

Пример 2.10.1



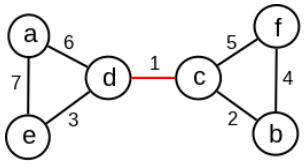
Слика 2.78: Граф који је у облику циклуса.

Размотримо пример графа са слике 2.78. Грана (c, d) је грана минималне тежине у графу, и ако њу уклонимо и проблем сведемо на тражење минималног повезујућег дрвета у графу G без гране (c, d), добијамо да њега чине све остале гране графа: гране $(d, e), (e, a), (a, b)$ и (b, c). Додавањем гране (c, d) у ово минимално повезујуће дрво добијамо циклус, а знамо да циклус не може бити део минималног повезујућег дрвета.

Други проблем са применом овакве индуктивне хипотезе је тај што уклоњена грана може бити мост и након њеног уклањања граф не мора да остане повезан.

Пример 2.10.2

Размотримо граф са слике 2.79: након уклањања гране (c, d) минималне тежине, граф се распада на две компоненте повезаности.



Слика 2.79: Граф који се након уклањања гране минималне тежине распада на две компоненте повезаности.

Можемо рекурзивно конструисати минималну повезујућу шуму, али поново, као и у случају индукције по броју чворова, одустајемо од овог приступа, јер постоје једноставнији.

2.10.2 Примов алгоритам

Један могући приступ решавању овог проблема је да се током извршавања алгоритма минимално повезујуће поступно гради, додајући у сваком кораку тренутном поддрвету једну нову грану и један нови чвор. Према томе, индукција се овде не изводи по величини графа (броју грана или чворова графа), већ према броју *грана у поддрвету минималног повезујућег дрвета које се гради*.

Индуктивна хипотеза. За задати повезан граф $G = (V, E)$ умемо да пронађемо дрво T_k са k грана ($k < |V| - 1$), тако да је дрво T_k поддрво неког минималног повезујућег дрвета T графа G.

Базни случај је $k = 0$. Пошто сваки чвор графа припада његовом минималном повезујућем дрвету T (које постоји, јер је граф повезан), дрво T_0 може да садржи произвољни чвор графа и ниједну грану.

Претпоставимо да смо пронашли дрво T_k које задовољава индуктивну хипотезу и да је потребно да T_k проширимо наредном граном. Како да пронађемо нову грану за коју ћемо бити сигурни да припада неком минималном повезујућем дрвету? У сваком повезујућем дрвету које проширује дрво T_k мора да постоји бар једна грана која повезује неки чвор из T_k са неким чвором у остатку графа. Нека је E_k скуп свих таквих грана. Пошто је граф повезан, тај скуп мора бити непразан. За наредну грану узимамо произвољну грану са најмањом тежином из E_k. Тврдимо да свака таква грана припада неком минималном повезујућем дрвету T.

Лема 2.10.1

Нека је T_k поддрво неког минималног повезујућег дрвета T тежинског графа $G = (V, E)$ и нека је E_k скуп грана које повезују чворове из T_k са чворовима ван T_k. Нека је $e = (u, v)$ нека од грана најмање тежине међу гранама из E_k. Тада постоји

минимално повезујуће дрво T' графа G које садржи све гране из T_k и грану e.

Доказ. Ако дрво T садржи грану e, онда је оно тражено дрво $T' = T$.

Ако T не садржи e, оно садржи тачно један пут од u до v. Пошто грана (u, v) не припада минималном повезујућем дрвету T, онда она не припада ни путу од u до v кроз гране дрвета T. Међутим, пошто u припада, а v не припада дрвету T_k, на том путу мора да постоји бар још једна грана (u', v') таква да $u' \in T_k$ и $v' \notin T_k$. Размотримо однос тежина грана (u, v) и (u', v'):

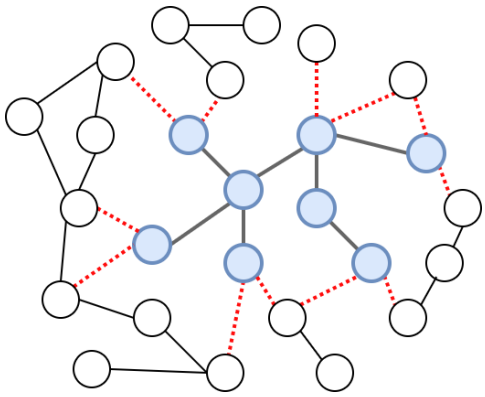
- Тежина (u, v) је најмања међу тежинама грана које повезују T_k са остатком графа, па тежина гране (u', v') не може бити мања од ње.
- Тежина гране (u', v') не може бити ни већа од тежине гране (u, v). Заиста, ако грану (u, v) додамо минималном повезујућем дрвету T, а из њега избацимо грану (u', v'), добијамо повезујуће дрво мање тежине, што је контрадикција.
- Ако је тежина гране (u', v') једнака тежини гране (u, v), тада се заменом гране (u', v') граном (u, v) добија повезујуће дрво T' исте тежине, па је и оно минимално. Дакле, постоји минимално повезујуће дрво које садржи грану (u, v).

□

Алгоритам се, дакле, спроводи на следећи начин. Дрво T се иницијализује дрветом T_0 које садржи само један (произвољно одабрани) чвор. У свакој итерацији се врши проширивање тренутно конструисаног дрвета T_k једном граном и чвором, тако што се проналази нека грана из скупа E_k који садржи све гране које повезују T_k са неким чвором ван T_k, а која има најмању тежину од свих грана у том скупу (ако има више таквих, бира се произвољна). Ово је илустровано на слици 2.80.

Уместо чувања и ажурирања целог скупа E_k, за сваки чвор v ван T_k можемо у низу памтити растојање од дрвета T_k, тј. минималну тежину гране од v до неког чвора из T_k (ако таква грана не постоји, вредност постављамо на $+\infty$) и у сваком кораку можемо одређивати минимум тог низа. Када се одабере грана најмање тежине из E_k и њен чвор $v \notin T_k$, анализирамо све гране које воде од v до неког чвора w ван T_k и ако је тежина неке такве гране (v, w) мања од тренутног растојања w од дрвета T_k, ажурирамо то растојање (и грану која од дрвета води до њега).

Ради ефикасног налажења минимума, све гране скупа E_k можемо чувати у реду са приоритетом, уређеном по дужинама грана. Након додавања новог чвора v у T_k, у ред се додају гране које га спајају са чворовима ван T_k. Међутим, неке гране које су биле у E_k не треба више да буду, јер сада спајају два чвора у дрвету. Слично као код Дајкстриног алгоритма и овде можемо користити лењо ажурирање реда са приоритетом и те гране не морамо брисати из реда. Зато, када се грана минималне тежине извади из реда са



Слика 2.80: Налажење следећег чвора и гране минималног повезујућег дрвета. Подебљано је дрво T_k, а гране скупа E_k су приказане испрекидано. Дрвету се додаје нека грана тог скупа која има најмању тежину.

приоретом, треба проверити да ли она спаја чвор у T_k са чвором ван њега. Ако спаја два чвора унутар T_k, треба је просто прескочити.

Поступак се завршава када се конструише дрво $T_{|V|-1} \equiv T$, јер оно садржи све чворове графа (пошто је број чворова увек за један већи од броја грана, тада дрво повезује свих $|V|$ чворова). Пошто је на основу леме 2.10.1 и оно поддрво неког минималног повезујућег дрвета, оно мора бити минимално повезујуће дрво.

Описани алгоритам познат је под називом *Примов алгоритам* и сличан је Дајкстрином алгоритму за налажење најкраћих путева од задатог чвора. Интересантно је да је овај алгоритам први осмислио Војтех Јарник[19] 1930. године, а да су тек касније независно до њега дошли Роберт Прим[20] 1957. године и Едзгер Дајкстра[21] 1959. године.

Из описа индуктивне конструкције и леме 2.10.1 следи следећа теорема.

Теорема 2.10.1 **[Коректност Примовог алгоритма]**

За произвољан неусмерен повезан тежински граф $G = (V, E)$ Примов алгоритам конструише једно минимално повезујуће дрво.

Једина разлика између Примовог и Дајкстриног алгоритма је у томе што се не тражи чвор ван тренутног скупа који има минимално растојање од почетног чвора, већ се

[19] Војтек Јарник (чеш. Vojtěch Jarník), (1897-1970), чешки математичар.
[20] Роберт Прим (енгл. Robert Prim), (1921-2021), амерички информатичар.
[21] Едзгар Дајкстра (хол. Edsger Wybe Dijkstra), (1930-2002), холандски информатичар.

тражи чвор ван тренутног скупа који има минимално растојање од тренутно конструисаног дрвета. Остатак алгоритма преноси се практично без промене.

```
// Primov algoritam za odredjivanje minimalnog povezujuceg drveta
vectorGrana> minimalnoPovezujuceDrvoPrim() {
   int brojCvorova = listaSuseda.size();
   // da li je cvor ukljucen u trenutno drvo
   vector<bool> uDrvetu(brojCvorova, false);
   // najkrace rastojanje od cvora do trenutnog drveta tj.
   // duzina najkrace grane koja povezuje trenutno drvo i cvor
   vector<Tezina> minRastojanja(brojCvorova, INF);
   // roditeljski cvor svakog cvora u minimalnom povezujucem drvetu
   vector<Cvor> roditelji(brojCvorova, -1);

   // uredjen par koji čine rastojanje do cvora i broj cvora;
   // redosled elemenata mora biti ovakav zbog operacije poredjenja parova
   typedef pair<Tezina, Cvor> RastojanjeCvora;
   // min-hip u koji smestamo duzine najkracih grana od drveta do svih cvorova
   priority_queue<RastojanjeCvora,
                  vector<RastojanjeCvora>,
                  greater<RastojanjeCvora>> rastojanja;

   // pocetni cvor moze biti bilo koji
   Cvor start = 0;
   // ubacujemo pocetni cvor u hip i postavljamo duzinu grane do njega na 0
   rastojanja.emplace(0, start);
   minRastojanja[start] = 0;

   // broj cvorova trenutno ukljucenih u minimalno povezujuce drvo
   int brojCvorovaUDrvetu = 0;

   // dok se svi cvorovi ne ukljuce u drvo
   while (brojCvorovaUDrvetu < brojCvorova) {
      // izdvajamo cvor cvor najblizi drvetu
      // (u prvom koraku izdvajamo pocetni cvor)
      auto [rastojanje, cvor] = rastojanja.top();
      rastojanja.pop();

      // ako je taj cvor vec u drvetu, treba ga preskociti
```

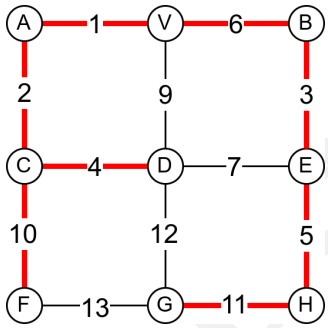
```
        // ovo se moze desiti jer vrsimo lenjo azuriranje hipa
        if (uDrvetu[cvor])
          continue;

        // dodajemo cvor u drvo
        uDrvetu[cvor] = true;
        brojCvorovaUDrvetu++;

        // za sve susede tekuceg cvora
        for (const auto& [sused, tezina] : listaSuseda[cvor]) {
           if (!uDrvetu[sused]) {
              // ako je grana iz tekuceg cvora do sused kraca od
              // prethodno najkrace grane iz drveta do tog suseda,
              // azuriramo vrednost najkrace grane i roditeljskog cvora
              if (tezina < minRastojanja[sused]) {
                 minRastojanja[sused] = tezina;
                 roditelji[sused] = cvor;
                 // ubacujemo element u hip;
                 // ako je postojala prethodna vrednost, ne brisemo je:
                 // nova vrednost ce se naci u hipu iznad stare
                 rastojanja.emplace(minRastojanja[sused], sused);
              }
           }
        }
    }

    // odredjujemo niz grana drveta na osnovu niza roditelja
    vector<Grana> grane;
    for (Cvor cvor = 0; cvor < brojCvorova; cvor++)
        if (roditelji[cvor] != -1)
           grane.emplace_back(roditelj[cvor], cvor);

    // vracamo konacan niz grana drveta
    return grane;
}
```

Када се користи ред са приоритетом, сложеност Примовог алгоритма идентична је сложености Дајкстриног алгоритма за налажење најкраћих растојања од задатог чвора и износи $O((|E| + |V|) \log |V|)$.

Уколико су тежине свих грана у графу међусобно различите, минимално повезујуће дрво графа биће јединствено.

Пример 2.10.3

Примов алгоритам за конструкцију минималног повезујућег дрвета за граф приказан на слици 2.81 илустрован је у табели 2.4. Чвор у првој колони табеле је онај који је додат у одговарајућем кораку. Први додати чвор је V, и у првој врсти наведени су чворови до којих постоје гране из чвора V са својим тежинама. У свакој врсти бира се грана са најмањом тежином. Списак тренутно најбољих грана и њихових тежина ажурира се у сваком кораку (приказани су само крајеви грана). На слици графа су гране које припадају минималном повезујућем дрвету подебљане.



Слика 2.81: Пример извршавања Примовог алгоритма за налажење минималног повезујућег дрвета.

Табела 2.4: Извршавање Примовог алгоритма за граф са слике 2.81.

	V	A	B	C	D	E	F	G	H
V	-	$1(V)$	$6(V)$	∞	$9(V)$	∞	∞	∞	∞
A	-	-	$6(V)$	$2(A)$	$9(V)$	∞	∞	∞	∞
C	-	-	-	-	$4(C)$	∞	$10(C)$	∞	∞
D	-	-	$6(V)$	-	-	$7(D)$	$10(C)$	$12(D)$	∞
B	-	-	-	-	-	$3(B)$	$10(C)$	$12(D)$	∞
E	-	-	-	-	-	-	$10(C)$	$12(D)$	$5(E)$
H	-	-	-	-	-	-	$10(C)$	$11(H)$	-
F	-	-	-	-	-	-	-	$11(H)$	-
G	-	-	-	-	-	-	-	-	-

Примов алгоритам је пример похлепног алгоритма, јер се у сваком кораку бира грана

скупа E_k са најмањом тежином, тј. избор локалног минимума води и ка глобалном минимуму.

2.10.3 Краскелов алгоритам

Интуитивно је јасно да је пожељно да минимално повезујуће дрво садржи гране графа са што мањим тежинама. Ако су све тежине графа међусобно различите, грана најмање тежине у графу мора бити укључена у минимално повезујуће дрво. Ако она не би била укључена, онда би њено додавање минималном повезујућем дрвету затворило неки циклус; уклањањем произвољне друге гране из тог циклуса поново се добија дрво, али мање тежине — што је у супротности са претпоставком о минималности конструисаног повезујућег дрвета. Ово инспирише алгоритам који обилази гране графа у неопадајућем редоследу тежина и укључује једну по једну у граф. Ако постоји више грана исте најмање тежине које могу бити додане, бира се било која од њих. Ипак, потребно је водити рачуна да неке гране са раније додатим гранама затварају циклус и њих морамо прескочити.

Дакле, и други ефикасан алгоритам за одређивање минималног повезујућег дрвета датог неусмереног тежинског графа $G = (V, E)$ је похлепан, јер у сваком кораку бира једну од најкраћих расположивих грана, али до минималног повезујућег дрвета не долази додавањем нових грана на тренутно дрво, него на тренутну шуму.

Индукција у овом алгоритму тече по броју обрађених грана у неопадајућем редоследу тежине грана.

Индуктивна хипотеза. За задати повезан граф $G = (V, E)$ и скуп његових k грана $E_k \subseteq E$ најмање тежине умемо да одредимо шуму K, која је део неког минималног повезујућег дрвета графа G. То дрво садржи све гране скупа K, а не садржи ни једну грану скупа $L = E_k \setminus K$.

Иницијално сваки чвор графа представља засебно дрво и одговарајућа шума садржи $|V|$ дрвета и ниједну грану. Затим се постепено спајају по два дрвета ове шуме, додавањем грана. Додавањем сваке нове гране, број дрвета у шуми се смањује за 1, те ће се након додавања $|V| - 1$ гране добити тачно једно дрво. При томе се гране које се додају разматрају у неопадајућем редоследу тежина; ако грана која је следећа на реду повезује два чвора у различитим дрветима тренутне шуме, онда се та грана укључује у шуму, чиме се та два дрвета спајају у једно. У противном, ако грана повезује два чвора из истог дрвета тренутне шуме, грана се прескаче.

Овај алгоритам је први објавио Џозеф Краскел[22] 1956. године и познат је под називом *Краскелов алгоритам*.

[22] Џозеф Краскел (енгл. Joseph Kruskal), (1928-2010), амерички математичар и информатичар.

Докажимо да ће овај алгоритам увек вратити неко минимално повезујуће дрво датог графа.

Теорема 2.10.2 **[Коректност Краскеловог алгоритма]**

За произвољан неусмерен повезан тежински граф $G = (V, E)$, Краскелов алгоритам конструише једно минимално повезујуће дрво.

Доказ. Лако се показује да алгоритам враћа повезујуће дрво датог графа (јер на крају имамо n чворова повезаних са $n - 1$ грана, без циклуса, што мора бити дрво). Треба показати да је оно и минимално.

Доказаћемо индукцијом да у сваком кораку алгоритма (приликом разматрања било које гране) постоји минимално повезујуће дрво T графа које садржи све гране скупа K које су одабране у претходним корацима и не садржи ни једну грану скупа L које су одбачене у претходним корацима. Након обраде свих грана, K је дрво које мора бити једнако минималном повезујућем дрвету које га садржи.

У почетном кораку није нити прихваћена, нити одбачена ни једна грана графа. Пошто је граф повезан он има минимално повезујуће дрво T. То дрво задовољава услов јер обухвата почетни празан скуп грана $K = \emptyset$ и не обухвата ни једну грану скупа $L = \emptyset$.

Нека је e наредна грана коју разматра Краскелов алгоритам у тренутку када су у шуму претходно додате гране скупа K и одбачене гране из неког скупа L. Она може бити додата у текућу шуму или одбачена.

Размотримо случај додавања нове гране e у шуму. Нека је T неко минимално повезујуће дрво које обухвата све гране скупа K и ниједну грану из L (оно постоји на основу индуктивне хипотезе). Нова шума $K' = K \cup \{e\}$ садржи и грану e и она не затвара циклус са осталим гранама из K.

- Ако грана e припада дрвету T, тада је $T' = T$ тражено минимално повезујуће дрво које садржи све гране из скупа $K' = K \cup \{e\}$ и ниједну грану из скупа $L' = L$.
- Ако грана e не припада дрвету T, онда се њеним додавањем у T затвара неки циклус у дрвету T (јер у дрвету T постоји неки други пут који повезује крајње чворове гране e). Тај циклус не постоји у шуми $K \cup e$ (јер се у супротном грана e не би додавала шуми). Зато у том циклусу мора да постоји нека грана e', различита од e, која припада дрвету T, али не припада шуми K. Пошто e' припада дрвету T, она не може да припада скупу L одбачених грана (јер T не садржи ни једну одбачену грану из скупа L). Дакле, e' не припада ни скупу L ни скупу K и различита је од e. Зато је Краскелов алгоритам још није разматрао, па њена

тежина мора бити већа или једнака тежини гране e. Избацивањем гране e' из дрвета T и додавањем гране e добијамо ново дрво T' (оно има исти број чворова и грана као дрво T, а остаје повезано, јер се једна грана која затвара циклус мења другом граном тог циклуса). Укупна тежина дрвета T' не може бити мања од тежине дрвета T, јер је T минимално повезујуће дрво, не може бити већа, јер тежина гране e која се додаје није већа од тежине гране e' која се избацује, па укупне тежине дрвета T и T' морају бити једнаке. T' је зато такође минимално повезујуће дрво, а оно садржи све гране из скупа $K' = K \cup \{e\}$ и не садржи ни једну грану из скупа одбачених грана $L' = L$.

Размотримо на крају случај одбацивања гране e. Једини разлог да се она одбаци је када она затвара неки циклус са гранама из K. Међутим, пошто се све гране из K налазе у дрвету T, а у T нема циклуса, грана e не припада дрвету T. Зато је T тражено минимално повезујуће дрво које садржи све гране из скупа $K' = K$ и ни једну грану из скупа $L' = L \cup \{e\}$. □

За утврђивање да ли су крајњи чворови u и v текуће гране (u, v) у истом или у различитим дрветима тренутне шуме, може се искористити структура података за дисјунктне скупове (енгл. union-find): тренутна дрвета шуме су дисјунктни подскупови скупа чворова графа. Операције `pronadji(u)` и `pronadji(v)` проналазе представнике u', v' два подскупа (корене дрвета којим су они представљени), па су чворови u и v у истом подскупу ако и само ако је $u' = v'$. Ако је $u' \neq v'$, онда се та два подскупа замењују својом унијом, тј. примењује се операција `unija(u',v')`, а два поддрвета тренутне шуме се граном спајају у једно.

```
// Kraskelov algoritam za odredjivanje minimalnog povezujuceg drveta
vector<pair<Grana, Tezina>> minimalnoPovezujuceDrvoKraskel() {
   int brojCvorova = listaSuseda.size();

   // inicijalizujemo union-find strukturu
   // svaki cvor grafa predstavlja podskup za sebe
   UF_inicijalizuj(brojCvorova);

   // kreiramo jedinstveni niz svih grana u grafu
   typedef pair<Cvor, Cvor> Grana;
   vector<pair<Tezina, Grana>> grane;
   for (Cvor cvor = 0; cvor < brojCvorova; cvor++)
      for (const auto& [sused, tezina]: listaSuseda[cvor]) {
        Grana grana{cvor, sused};
        grane.emplace_back(tezina, grana);
```

```
        }

    // sortiramo skup grana u neopadajucem redosledu težina
    sort(grane.begin(), grane.end());

    // grane koje pripadaju konacnom minimalnom povezujucem drvetu
    vector<pair<Grana, Tezina>> drvo;
    // drvo ce sadrzati |V| - 1 grana, pa unapred rezervisemo memoriju
    drvo.reserve(brojCvorova - 1);

    // prolazimo redom kroz skup grana
    for (const auto& [tezina, grana] : grane) {
        // krajnji cvorovi grane
        auto [u, v] = grana;
        // predstavnici cvorova u union-find strukturi
        Cvor predstavnik_u = UF_pronadji(u);
        Cvor predstavnik_v = UF_pronadji(v);

        // ako tekuca grana povezuje dva cvora koja pripadaju razlicitim drvetima,
        // onda tu granu dodajemo u minimalno povezujuce drvo
        // i pravimo uniju skupova cvorova koji pripadaju tim drvetima
        if (predstavnik_u != predstavnik_v) {
            // spajamo drveta kojima pripadaju u i v (dodavanjem grane uv)
            UF_unija(predstavnik_u, predstavnik_v);
            // granu dodajemo u skup grana koje cine sumu (tj. drvo na kraju)
            drvo.emplace_back(grana, tezina);
            // drvo je kompletno konstruisano kada je broj grana za jedan manji
            // od broja cvorova
            if (drvo.size() == brojCvorova - 1)
                break;
      }
    }

    return drvo;
}
```

Функција за иницијализацију структуре података за дисјунктне скупове је сложености $O(|V|)$, додавање грана у скуп грана је сложености $O(|E|)$, њихово сортирање је сложености $O(|E| \log |E|)$, а након тога се главна петља извршава $|E|$ пута, док су операције које се извршавају у петљи (`UF_pronadji` и `UF_unija`) сложености $O(\log |V|)$. Дакле,

укупна сложеност Краскеловог алгоритма износи $O(|V|) + O(|E|) + O(|E| \log |E|) + O(|E| \log |V|) = O(|E| \log |V|)$ (користимо чињеницу да је $O(\log |E|) = O(\log |V|)$ због $|E| \leq |V|^2$).

Пример 2.10.4

Пример извршавања Краскеловог алгоритма за граф са слике 2.81 приказан је у табели 2.5. Пролазимо редом кроз скуп грана према неопадајућем редоследу тежина и завршавамо обраду када у скуп додамо $|V| - 1$ (у овом случају 8) грана. Као резултат добијамо минимално повезујуће дрво приказано на слици 2.81.

Табела 2.5: Пример извршавања Краскеловог алгоритма за граф са слике 2.81.

наредна грана	*дужина гране*	*укључена у MCST?*	*тренутна шума*
(A, V)	1	*да*	$\underline{V}, \underline{A}, B, C, D, E, F, G, H$
(A, C)	2	*да*	$\underline{AV}, B, \underline{C}, D, E, F, G, H$
(B, E)	3	*да*	$ACV, \underline{B}, C, D, \underline{E}, F, G, H$
(C, D)	4	*да*	$A\underline{C}V, BE, \underline{D}, E, F, G, H$
(E, H)	5	*да*	$ACDV, B\underline{E}, F, G, \underline{H}$
(B, V)	6	*да*	$ACD\underline{V}, \underline{B}EH, F, G$
(D, E)	7	*не*	$ABC\underline{DE}HV, F, G$
(D, V)	8	*не*	$ABC\underline{D}EH\underline{V}, F, G$
(C, F)	9	*да*	$AB\underline{C}DEHV, \underline{F}, G$
(G, H)	10	*да*	$ABCDEF\underline{H}V, \underline{G}$
			$ABCDEFGHV$

Коначно, напоменимо да се на сличан начин може конструисати и повезујуће дрво са максималном могућом укупном тежином грана, тзв. *максимално повезујуће дрво* графа. Једино је потребно гране обрађивати у обрнутом редоследу.

Задатак: Вода до сваке куће

У једном селу има n кућа. Желимо да изградимо водовод тако да до сваке куће долази вода: то постижемо тиме што градимо бунаре и повезујемо куће цевима. За сваку кућу i можемо или да изградимо бунар или да је цевима повежемо са неким суседним кућама. Кроз сваку цев вода може да иде у произвољном смеру. Ако су познате цене изградње бунара у свакој кући и цене повезивања кућа цевима, напиши програм који израчунава најмању цену потребну да свака кућа добије воду.

Опис улаза

Са стандардног улаза се учитава број кућа n ($1 \leq n \leq 5 \cdot 10^4$), затим n бројева који представљају цене изградње бунара за сваку кућу, а затим до краја улаза цене изградњи цеви (по три цела броја у реду, где прва два броја представљају различите куће, а трећи цену изградње цеви између тих кућа, при чему укупан број цеви не прелази 10^6). Бројеви кућа су од 1 до n.

Опис излаза

На стандардни излаз исписати најмању цену изградње водовода.

Пример 1

Улаз	*Излаз*	*Објашњење*

`3`
`1 2 2`
`1 2 1`
`2 3 1`
	`3`	Најбоље је направити бунар у кући 1 и повезати друге две куће цевима са њом.

Пример 2

Улаз	*Излаз*	*Објашњење*

`4`
`2 2 2 2`
`1 2 5`
`1 3 5`
`1 4 5`
`2 3 5`
`2 4 5`
`3 4 5`
	`8`	Најбоље је да свака кућа добије свој бунар.

Решење

Када бунари не би постојали и када би само било потребе повезати све куће цевима, проблем би се директно сводио на проналажење минималног повезујућег дрвета датог скупа кућа.

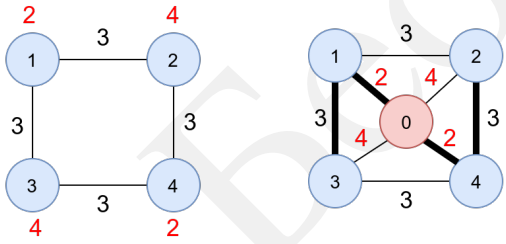
Могућност и потреба да се изграде бунари наизглед чине задатак компликованијим. Међутим, постоји веома елегантан начин да се и проблем са бунарима сведе на проблем минималног повезујућег дрвета, тј. да се изградња бунара моделује изградњом цеви. Наиме, сваки се бунар може замислити као цев која спаја кућу за земљом у којој се већ налази вода. Зато полазни граф у коме се налазе куће и цеви које их могу повезати треба допунити једним посебним чвором који представља земљу и сваку кућу треба повезати са тим чвором при чему је цена цеви која спаја земљу и кућу једнака цени изградње бунара у тој кући. Када се пронађе минимално повезујуће дрво, оно ће сигурно повезивати земљу са осталим кућама, што значи да ће вода из земље сигурно моћи да буде

допремљена до сваке друге куће (да би земља била повезана, сигурно ће бар један бунар бити изграђен). Ако градимо усмерен граф, довољно је да усмеримо гране од земље ка свакој кући (у земљи се већ налази вода, па није потребно враћати воду из неке куће назад до земље).

Када се направи овако проширени граф, минимално повезујуће дрво може да се пронађе уобичајеним алгоритмима.

Пример 2.10.5

На слици лево је приказан граф где су приказане моћуће цеви и цене бунара за сваку кућу, док је на слици десно приказан проширен граф где је уведен чвор 0 *(земља) и бунари су представљени цевима које спајају земљу и куће. Нацртано минимално повезујуће дрво указује на то да је потребно изградити бунаре у кућама* 1 *и* 4*, а да је куће* 2 *и* 3 *потребно цевима повезати са кућама* 1 *и* 4*.*



2.11 Сви најкраћи путеви

Проблем рутирања у мрежама рачунара (или дистрибуираним системима) представља одабир путање кроз мрежу којом пакети путују од своје почетне дестинације до одредишта. Један од критеријума по којима се врши одабир путање јесте тај да пређени пут буде што је могуће краћи. Како би се пренос података од једног уређаја до другог учинио што је могуће ефикаснијим, потребно је познавати најкраћи пут између свака два уређаја. Овај проблем могуће је моделовати у виду графа, тако што ћемо сваки од уређаја у мрежи представити чвором графа, а сваку од постојећих веза између уређаја граном графа чија тежина одговара мери растојања између та два уређаја. Тада се полазни проблем своди на проблем израчунавања најкраћег пута између свака два чвора у тежинском графу.

Проблем

Дат је тежински граф $G = (V, E)$ (усмерени или неусмерени). Одредити путеве минималне дужине између свака два чвора у графу.

Поново, пошто говоримо о најкраћим путевима, тежине грана можемо интерпретирати као дужине грана. Овај проблем називамо *проблем налажења свих најкраћих путева* (енгл. all pairs shortest path). За почетак ћемо се задовољити налажењем дужина свих најкраћих путева, уместо самих најкраћих путева. Претпоставимо да је граф усмерен; све што ће бити речено важи и за неусмерене графове. Тежине грана у графу могу бити негативне, али у графу не сме постојати циклус негативне тежине.

Један могући приступ је да се Белман-Фордов алгоритам (или Дајкстрин алгоритам, ако су тежине грана ненегативне) покрене из сваког чвора графа. Међутим, видећемо да постоје и ефикаснији алгоритми који директно решавају овај проблем.

Као и обично, покушајмо са директним индуктивно-рекурзивним приступом. Може се користити индукција по броју грана или по броју чворова у графу. Означимо са $d(u, v)$ дужину гране (u, v), а са $\overline{SP}(u, v)$ дужину тренутно познатог најкраћег пута од чвора u до чвора v.

2.11.1 Алгоритам заснован на индукцији по броју грана у графу

Размотримо најпре индукцију по броју грана. Претпоставимо да смо из графа G уклонили грану (u, v) и да смо решили проблем на преосталом графу G', тј. да знамо дужине најкраћих путева између свака два чвора у графу G'. Како се мењају најкраћи путеви у графу након додавања гране (u, v) у граф? Нова грана може пре свега да представља краћи пут од до сада пронађеног најкраћег пута од чвора u до чвора v. Дакле, потребно је проверити да ли важи $d(u, v) < \overline{SP}(u, v)$ и ако важи, поставити вредност $\overline{SP}(u, v)$ на $d(u, v)$. Поред тога, додавањем гране (u, v) може се променити и најкраћи пут између произвољна друга два чвора s и t. Да би се установило има ли промене, треба са претходно познатом најмањом дужином пута од чвора s до чвора t упоредити збир дужина најкраћег пута од s до u, дужине гране (u, v) и дужине најкраћег пута од v до t. Дакле, ако је $\overline{SP}(s, u) + d(u, v) + \overline{SP}(v, t) < \overline{SP}(s, t)$ потребно је нову вредност $\overline{SP}(s, t)$ поставити на $\overline{SP}(s, u) + d(u, v) + \overline{SP}(v, t)$.

Пошто је број парова чворова $O(|V|^2)$, при додавању гране (u, v) потребно је извршити $O(|V|^2)$ провера. Овај поступак је потребно спровести за сваку грану графа, те је сложеност алгоритма за налажење свих најкраћих путева заснованог на индукцији по броју грана у најгорем случају $O(|E| \cdot |V|^2)$. Пошто је број грана највише $O(|V|^2)$, сложеност овог алгоритма износи $O(|V|^4)$. Ово је веома неефикасно, па овај алгоритам нећемо имплементирати.

2.11.2 Алгоритам заснован на индукцији по броју чворова у графу

Размотримо сада индукцију по броју чворова. Претпоставимо да смо из графа G уклонили чвор u и да смо између свака друга два чвора у графу пронашли најкраћи пут. Како се мењају најкраћи путеви у графу ако се у граф дода нови чвор u? Потребно је најпре пронаћи најкраће путеве од чвора u до свих осталих чворова, и најкраће путеве од свих осталих чворова до чвора u. Пошто су дужине најкраћих путева који не садрже u већ познате, одређивање најкраћег пута од чвора u до произвољног чвора w своди се на одређивање само прве гране на том путу; ако је то грана (u, v), онда је дужина најкраћег пута од чвора u до чвора w једнака збиру дужине гране (u, v) и дужине најкраћег пута од v до w, који је већ познат. Потребно је, дакле, да упоредимо ове збирове за све гране које полазе из чвора u, и да међу њима изаберемо најмањи:

$$\overline{SP}(u, w) = \min_{(u,v)\in E} \{d(u, v) + \overline{SP}(v, w)\}.$$

Најкраћи пут од произвољног чвора w до чвора u може се пронаћи на сличан начин. Ове провере су у најгорем случају (када из чвора u полази/у њега долази $\Theta(|V|)$ грана) сложености $O(|V|^2)$.

Међутим, додавање чвора u може скратити путеве између нека друга два чвора, јер пут преко u може бити краћи него путеви који не пролазе кроз u. Зато је, додатно, потребно за сваки пар чворова проверити да ли између њих постоји нови краћи пут кроз нови чвор u. За произвољна два чвора s и t графа, да би се установило постоји ли краћи пут преко чвора u, потребно је са претходно познатом најмањом дужином пута од s до t упоредити збир дужине најкраћег пута од s до u и дужине најкраћег пута од u до t. Ако важи $\overline{SP}(s, u) + \overline{SP}(u, t) < \overline{SP}(s, t)$, ажурирамо вредност $\overline{SP}(s, t)$.

Ово укључује укупно $O(|V|^2)$ провера и сабирања после додавања сваког новог чвора, па је сложеност оваквог алгоритма у најгорем случају $O(|V| \cdot |V|^2) = O(|V|^3)$. Укупан број корака за одређивање дужина најкраћих путева од и до новододатог чвора је такође $O(|V|^3)$. Дакле, испоставља се да је за решавање овог проблема ефикаснија индукција по броју чворова него индукција по броју грана.

Размотримо имплементацију алгоритма за рачунање свих најкраћих путева у графу заснованог на индукцији по чворовима, у случају када је граф задат матрицом повезаности. Наиме, овде нам је та репрезентација погоднија од листи повезаности јер дужине грана одговарају иницијалним најкраћим растојањима између чворова.

```
const int INF = numeric_limits<int>::max();

// odredjujemo sve najkrace puteve indukcijom po broju cvorova
```

```
vector<vector<int>> sviNajkraciPutevi(
          const vector<vector<int>>& matricaPovezanosti) {
  int brojCvorova = matricaPovezanosti.size();
  vector<vector<int>> najkraciPut(brojCvorova);
  for (int i = 0; i < brojCvorova; i++)
    najkraciPut[i].resize(brojCvorova, INF);

  // dodajemo jedan po jedan cvor
  for (int i = 0; i < brojCvorova; i++) {
    najkraciPut[i][i] = 0;

    // odredjujemo najkrace puteve od cvora i do svih prethodnih
    // cvorova j
    for (int j = 0; j < i; j++) {
      // pretpostavljamo da je direktno rastojanje najkrace
      najkraciPut[i][j] = matricaPovezanosti[i][j];
      // proveravamo da li je mozda bolji put od i do j koji vodi preko
      // nekog prethodnog cvora k
      for (int k = 0; k < i; k++)
        // ako postoji grana od i do k i put od k do j
        if (matricaPovezanosti[i][k] != INF && najkraciPut[k][j] != INF &&
            matricaPovezanosti[i][k] + najkraciPut[k][j] < najkraciPut[i][j])
          najkraciPut[i][j] = matricaPovezanosti[i][k] + najkraciPut[k][j];
    }

    // odredjujemo najkrace puteve do cvora i od svih prethodnih
    // cvorova j
    for (int j = 0; j < i; j++) {
      // pretpostavljamo da je direktno rastojanje najkrace
      najkraciPut[j][i] = matricaPovezanosti[j][i];
      // proveravamo da li je mozda bolji put od cvora j do i koji
      // vodi preko nekog prethodnog cvora k
      for (int k = 0; k < i; k++)
        // ako postoji grana od i do k i put od k do j
        if (najkraciPut[j][k] != INF && matricaPovezanosti[k][i] != INF &&
            najkraciPut[j][k] + matricaPovezanosti[k][i] < najkraciPut[j][i])
          najkraciPut[j][i] = najkraciPut[j][k] + matricaPovezanosti[k][i];
    }
```

```
    // ažuriramo rastojanja od prethodnih cvorova j do prethodnih
    // cvorova k, analizirajuci puteve koji vode preko cvora i
    for (int j = 0; j < i; j++)
      // ako postoji put od j do i i ako postoji put od i do k
      for (int k = 0; k < i; k++)
        if (najkraciPut[j][i] != INF && najkraciPut[i][k] != INF &&
            najkraciPut[j][i] + najkraciPut[i][k] < najkraciPut[j][k])
          najkraciPut[j][k] = najkraciPut[j][i] + najkraciPut[i][k];
  }
  return najkraciPut;
}
```

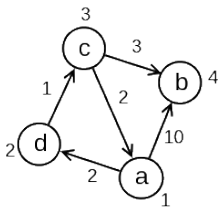
2.11.3 Флојд-Варшалов алгоритам

Показује се да постоји још једноставнија индуктивна конструкција за решавање овог проблема. Идеја на којој се заснива овај алгоритам је да се не мења број чворова или грана у графу, већ да се уведу ограничења на тип дозвољених путева. Наиме, разматрају се само путеви који као међучворове користе чворове из задатог скупа чворова, при чему се у сваком кораку тај скуп проширује једним новим чвором. На почетку је тај скуп празан, па у обзир долазе само гране графа, а на крају обухвата све чворове графа, па у обзир долазе сви могући путеви. Нумеришимо чворове графа на произвољан начин бројевима од 1 до $|V|$. Пут од чвора u до чвора v зове се k-*пут*, где је $k \in \{0, 1, \ldots, |V|\}$, ако су редни бројеви свих чворова на том путу (изузев u и v) мањи или једнаки k (скуп допуштених међучворова је, дакле, скуп чворова означен бројевима од 1 до k). Специјално, 0-пут од чвора u до чвора v може бити само директна грана од чвора u до чвора v (пошто се ниједан други чвор не може појавити на том путу).

Пример 2.11.1

Размотримо пример графа са слике 2.82. Нека су редни бројеви чворова a, d, c, b редом 1, 2, 3, 4. Пут (a, b) од чвора a до чвора b дужине 10 је 0-пут, јер се састоји од само једне гране (нема унутрашњих чворова на путу). Пут (a, d, c, b) дужине 6 је 3-пут, јер су редни бројеви свих чворова на том путу мањи или једнаки 3 (истовремено је и 4-пут), али није нпр. 2-пут. Слично, пут (c, a, b) је 1-пут, јер су редни бројеви свих чворова (у овом случају једног јединог чвора) на путу мањи или једнаки 1 (овај пут је, такође, и 2-пут, 3-пут и 4-пут). Приметимо да је најкраћи 0-пут (истовремено и најкраћи 1-пут и најкраћи 2-пут) од чвора a до чвора b пут (a, b) дужине 10, док је најкраћи 3-пут (и истовремено најкраћи 4-пут) (a, d, c, b) дужине 6.

Размотримо алгоритам заснован на индукцији по опадајућем броју ограничења на тип



Слика 2.82: Илустрација k-путева у усмереном тежинском графу.

дозвољених путева.

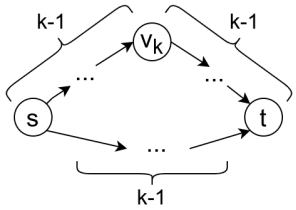
Индуктивна хипотеза. Умемо да одредимо дужине најкраћих $(k-1)$-путева између свака два чвора.

База индукције је случај $k = 1$, кад се разматрају само директне гране и решење је очигледно: најкраћи 0-пут између два чвора једнак је дужини гране ако она постоји, а $+\infty$ ако грана не постоји.

Претпоставимо да је индуктивна хипотеза тачна и да хоћемо да је проширимо на k-путеве. Потребно је да одредимо најкраће k-путеве између свака два чвора. Нека је v_k чвор са редним бројем k. Произвољан најкраћи k-пут садржи чвор v_k највише једном. Наиме, претпоставка је да у графу не постоји циклус негативне дужине, па најкраћи пут не може два пута да прође кроз чвор v_k — у противном би се пут могао скратити избацивањем циклуса од грана између првог и другог појављивања чвора v_k. Најкраћи k-пут од чвора s до чвора t је или најкраћи $(k-1)$-пут од чвора s до чвора t или се састоји од најкраћег $(k-1)$-пута од чвора s до чвора v_k, и најкраћег $(k-1)$-пута од чвора v_k до чвора t јер су чворови на k-путу од s до v_k и од v_k до t нумерисани бројевима мањим или једнаким $k-1$ (слика 2.83). Према индуктивној хипотези ми већ знамо дужине најкраћих $(k-1)$-путева. Дакле, да бисмо пронашли дужину најкраћег k-пута од чвора s до чвора t довољно је да саберемо ове две дужине и збир упоредимо са дужином најкраћег $(k-1)$-пута од чвора s до чвора t.

До овог алгоритма су независно дошли и објавили га Роберт Флојд[23] и Стивен Варшал[24] и познат је под називом *Флојд-Варшалов алгоритам*. Напоменимо да је неколико година пре њих до истог алгоритма дошао и Бернар Роj[25], међутим, тај резултат је прошао незапажено. Ипак, негде у литератури се овај алгоритам помиње и под називом Роj-Варшалов или Роj-Флојдов алгоритам. Флојд-Варшалов алгоритам је за константан фактор бржи од алгоритма заснованог на примени индукције по броју чворова и

[23] Роберт Флојд (енгл. Robert Floyd), (1936-2001), амерички информатичар.
[24] Стивен Варшал (енгл. Stephen Warshall), (1935-2006), амерички информатичар.
[25] Бернар Роj (фр. Bernard Roy), (1934-2017), француски математичар.



Слика 2.83: Илустрација процеса рачунања најкраћих k-путева: најкраћи $(k-1)$-пут од чвора s до чвора t поредимо са збиром најкраћих $(k-1)$-путева од s до v_m и од v_m до t.

лакше га је реализовати.

Претходним индуктивним разматрањем доказали смо коректност овог алгоритма.

Теорема 2.11.1 **[Коректност Флојд Варшаловог алгоритма]**

За дати усмерен тежински граф $G = (V, E)$ без циклуса негативне тежине, Флојд-Варшалов алгоритам исправно израчунава најкраћа растојања између свих парова чворова.

До сад смо се бавили израчунавањем дужина најкраћих путева између свака два чвора у графу. Размотримо сада како бисмо могли да реконструишемо саме најкраће путеве, а да притом чувамо што мање информација: пожељно је да за сваки најкраћи пут чувамо само по једну вредност. Присетимо се да је у претходним алгоритмима за сваки чвор било довољно чувати претпоследњи чвор на најкраћем путу до тог чвора, јер је сваки наредни најкраћи пут био добијен продуживањем неког претходно одређеног најкраћег пута једном новом граном. Овде таква реконструкција не би имала смисла, јер се најкраћи путеви конструишу на другачији начин, па алгоритам реконструкције најкраћих путева мора да прати поступак конструкције најкраћих путева. У алгоритму се текући најкраћи пут између два чвора мења онда када је пут кроз новоразматрани чвор v_k краћи него претходно одређени најкраћи $(k-1)$-пут. Дакле, можемо ангажовати додатну матрицу *medjucvor* у којој ћемо на позицији (i, j) памтити максималну вредност k такву да чвор v_k припада најкраћем путу од чвора i до чвора j, а ако је најкраћи пут директна грана од i до j, онда вредност $medjucvor_{ij}$ можемо поставити на i. Ако се приликом тражења најкраћег пута од чвора i до чвора j нађе краћи пут који води преко чвора v_k, ажурираћемо вредност $medjucvor_{ij}$ на k. Исписивање најкраћег пута од чвора i до чвора j се онда своди на исписивање најкраћег пута од чвора i до чвора $medjucvor_{ij}$ за којим следи најкраћи пут од чвора $medjucvor_{ij}$ до чвора j.

У наредној C++ имплементацији претпостављамо да је граф задат матрицом повезано-

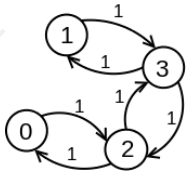
сти. Као нумерацију чворова искористићемо њихов индекс. Претпостављамо да се на местима која одговарају паровима чворова између којих не постоје гране налази вредност $+\infty$ (кодирана помоћу максималне вредности типа којим се представља тежина грана). Оваква улазна матрица одговара дужини нула путева између чворова. Наредна функција одређује дужине најкраћих путева између свака два пара чвора, а у помоћну матрицу `medjucvor`, у врсту `i` и колону `j` смешта највећу вредност међучвора `k` који се налази на најкраћем путу између чворова `i` и `j`.

```
// vrednost kojom predstavljamo +beskonacno (odnosno odsustvo grane tj. puta)
const int INF = numeric_limits<Tezina>::max();

// funkcija koja racuna najkrace puteve izmedju svaka dva cvora
// ulaz je tezinska matrica povezanosti (sa vrednostima INF gde nema grane)
// duzine najkracih puteva se smestaju u matricu najkraciPut
// matrica medjucvor sadrzi cvorove koji su deo najkracih puteva
void sviNajkraciPutevi(const vector<vector<Tezina>> &matricaPovezanosti,
                       vector<vector<Tezina>>& najkraciPut,
                       vector<vector<int>>& medjucvor) {
  int brojCvorova = matricaPovezanosti.size();
  // inicijalizujemo matricu koja cuva duzine najkracih puteva
  // na duzine direktnih grana
  najkraciPut = matricaPovezanosti;

  // alociramo i inicijalizujemo matricu medjucvor pomocu koje cemo
  // rekonstruisati najkrace puteve
  medjucvor.resize(brojCvorova);
  for (int i = 0; i < brojCvorova; i++) {
    medjucvor[i].resize(brojCvorova);
    for (int j = 0; j < brojCvorova; j++)
      // put od cvora do njega samog
      if (i == j)
        medjucvor[i][j] = 0;
      // postoji direktna grana izmedju dva razlicita cvora i i j
      else if (matricaPovezanosti[i][j] != INF)
        medjucvor[i][j] = i;
      // ne postoji direktna grana izmedju dva razlicita cvora i i j
      else if (i != j)
        medjucvor[i][j] = -1;
  }
```

```
  // proveravamo da li k-putevi skracuju puteve izmedju cvora i i j
  for (int k = 0; k < brojCvorova; k++)
    for (int i = 0; i < brojCvorova; i++)
      for (int j = 0; j < brojCvorova; j++)
        // ako postoji neki put od i do k i ako postoji neki put od k do j
        // tako da je put preko k kraci nego do sada najkraci put od i do j
        if (najkraciPut[i][k] != INF && najkraciPut[k][j] != INF &&
            najkraciPut[i][k] + najkraciPut[k][j] < najkraciPut[i][j]) {
          // smanjujemo duzinu puta
          najkraciPut[i][j] = najkraciPut[i][k] + najkraciPut[k][j];
          // postavljamo da je na putu od cvora i do cvora j
          // maksimalna oznaka medjucvora jednaka k
          medjucvor[i][j] = k;
        }
}
```

Напоменимо да је у троструко угнежђеној петљи неопходно да спољашња петља контролише параметар k који ограничава тип дозвољених путева, док се унутрашње две петље користе се за проверу свих парова чворова. Запажа се да се ова провера може извршавати са паровима чворова у произвољном редоследу, јер је свака од провера потпуно независна од осталих.

Пример 2.11.2



Слика 2.84: Граф који указује на важност редоследа петљи у Флојд-Варшаловом алгоритму. Варијанта алгоритма код које унутрашња (а не спољашња) петља контролише тип дозвољених путева за овај граф погрешно израчунава најкраћи пут од чвора 0 до чвора 1.

Размотримо пример графа са слике 2.84 код кога су тежине свих грана једнаке 1. *Нека су редни бројеви чворова* 0, 1, 2 *и* 3 *са слике редом једнаки* 1, 2, 3 *и* 4. *У њему постоји пут од чвора* 0 *до чвора* 1 *дужине* 3 *и он представља најкраћи пут од чвора* 0 *до чвора* 1. *Размотримо варијанту Флојд-Варшаловог алгоритма са наредним, мало измењеним редоследом петљи:*

```
for (int i = 0; i < brojCvorova; i++)
  for (int j = 0; j < brojCvorova; j++)
    for (int k = 0; k < brojCvorova; k++)
      if (najkraciPut[i][k] != INF && najkraciPut[k][j] != INF &&
          najkraciPut[i][k] + najkraciPut[k][j] < najkraciPut[i][j]) {
        najkraciPut[i][j] = najkraciPut[i][k] + najkraciPut[k][j];
```

И у овој варијанти променљива k контролише тип дозвољених путева, i полазни чвор, а j крајњи чвор пута, али је редослед петљи другачији. Овај алгоритам одговара томе да се за свака два фиксирана чвора нумерисана вредностима i и j пролази кроз све чворове k и разматра да ли постоји пут преко чвора k. За вредност променљивих $i = 0$ и $j = 1$ проверавају се редом вредности $0, 1, 2$ и 3 за k и пошто не постоји k тако да истовремено постоји и грана (i, k) и грана (k, j), пут од чвора 0 до чвора 1 не би био пронађен, иако он постоји у графу. Наиме, да бисмо открили најкраћи пут (тј. најкраћи 4-пут) од чвора 0 до чвора 1 потребно је претходно одредити најкраћи 3-пут од чвора 0 до чвора 1, што се у овој варијанти алгоритма не дешава. Закључујемо да овакав редослед петљи не омогућава налажење свих најкраћих путева. Дакле, важно је да спољашња од три петље пролази скупом вредности k, где k контролише тип дозвољног пута.

Флојд-Варшалов алгоритам ради коректно и за графове који имају негативне тежине грана (све док у графу не постоји циклус негативне тежине) јер коректност алгоритма не зависи од тога да су тежине грана у графу ненегативне.

Ако полазни граф има циклус негативне тежине, то се може утврдити тако што ће након извршавања Флојд-Варшаловог алгоритма дужина најкраћег пута од неког чвора до њега самог бити мања од 0, односно нека вредност на дијагонали матрице `najkraciPut` ће бити мања од 0. Наиме, иницијално важи `najkraciPut[i][i] = 0` за свако `i`. Флојд-Варшалов алгоритам рачуна најкраће путеве између свака два чвора у графу, па и између парова истих чворова. Ако чвор `i` припада циклусу негативне тежине, онда ће Флојд-Варшалов алгоритам размотрити и пут од чвора `i` до њега самог кроз гране овог циклуса и иницијалну вредност 0 смањити на тежину овог циклуса.

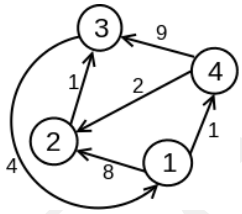
Ако се у неком кораку спољашње петље установи да се матрица `najkraciPut` није променила, алгоритам се може раније прекинути.

За сваку вредност k алгоритам извршава једно сабирање и једно упоређивање за сваки пар чворова. Број корака индукције је $|V|$, па је укупан број сабирања, односно упоређивања, највише $|V|^3$. Присетимо се да је временска сложеност Дајкстриног алгоритма за налажење најкраћих путева од једног задатог чвора у графу који не садржи гране негативне дужине $O((|V| + |E|) \log |V|)$. Ако је граф густ, па је број грана $\Theta(|V|^2)$, онда је за одређивање свих најкраћих путева у графу Флојд-Варшалов алгоритам ефика-

снији од извршавања Дајкстриног алгоритма од сваког полазног чвора у графу. С друге стране, ако граф није густ (па има, на пример, $O(|V|)$ грана), онда је боља временска сложеност $O(|V|(|V|+|E|)\log|V|)$ која потиче од $|V|$ пута употребљеног алгоритма за најкраће путеве од једног чвора. Једна од предности Флојд-Варшаловог алгоритма је, свакако, и његова једноставна реализација, као и то што је применљив и на графове који имају негативне гране.

Пример 2.11.3

Размотримо поступак одређивања најкраћих путева између свака два чвора у графу са слике 2.85.



Слика 2.85: Усмерени тежински граф за који тражимо најкраћи пут између свака два чвора.

Он се састоји из наредних корака:

$k=0$: *najkraciPut:*
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{pmatrix} \end{array}$$
medjucvor:
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & - & 1 \\ - & 0 & 2 & - \\ 3 & - & 0 & - \\ - & 4 & 4 & 0 \end{pmatrix} \end{array}$$

$k=1$: *najkraciPut:*
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{pmatrix} \end{array}$$
medjucvor:
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & - & 1 \\ - & 0 & 2 & - \\ 3 & 1 & 0 & 1 \\ - & 4 & 4 & 0 \end{pmatrix} \end{array}$$

$k=2$: *najkraciPut:*
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{pmatrix} \end{array}$$
medjucvor:
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 2 & 1 \\ - & 0 & 2 & - \\ 3 & 1 & 0 & 1 \\ - & 4 & 2 & 0 \end{pmatrix} \end{array}$$

$$k = 3\text{: } najkraciPut\text{:} \quad \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{pmatrix} \end{array} \qquad medjucvor\text{:} \quad \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 2 & 1 \\ 3 & 0 & 2 & 3 \\ 3 & 1 & 0 & 1 \\ 3 & 4 & 2 & 0 \end{pmatrix} \end{array}$$

$$k = 4\text{: } najkraciPut\text{:} \quad \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{pmatrix} \end{array} \qquad medjucvor\text{:} \quad \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 4 & 4 & 1 \\ 3 & 0 & 2 & 3 \\ 3 & 4 & 0 & 1 \\ 3 & 4 & 2 & 0 \end{pmatrix} \end{array}$$

Из последње матрице `najkraciPut` *можемо прочитати да је најкраћи пут од чвора* 1 *до чвора* 3 *дужине* 4*, а из матрице* `medjucvor` *да је максимални индекс чвора на том најкраћем путу једнак* 4*. Дакле, да бисмо реконструисали најкраћи пут, потребно је да на најкраћи пут од чвора* 1 *до чвора* 4 *надовежемо најкраћи пут од чвора* 4 *до чвора* 3*. Из матрице* `medjucvor` *можемо прочитати да је највећи индекс чвора на најкраћем путу од чвора* 1 *до чвора* 4 *баш једнак* 1*, што нам говори да је тај пут директна грана између тих чворова. Из матрице* `medjucvor` *можемо такође прочитати да је највећи индекс чвора на најкраћем путу од чвора* 4 *до чвора* 3 *једнак* 2*, што нам говори да тај пут одређујемо тако што на најкраћи пут од чвора* 4 *до чвора* 2 *надовежемо најкраћи пут од чвора* 2 *до чвора* 3*. Из матрице* `medjucvor` *можемо закључити да оба ова пута одговарају директним гранама. Коначно, закључујемо да је најкраћи пут од чвора* 1 *до чвора* 3 *једнак* $(1, 4, 2, 3)$*.*

Матрицу `medjucvor` употребљавамо да бисмо одредили и одштампали све најкраће путеве. Прикажимо сада C++ имплементацију ове функционалности. Рекурзивна помоћна функција `odrediPut_` одређује чворове на најкраћем путу између датог пара чворова `i` и `j`. Она прво у вектор `put` убацује чворове на путу од `i` до `k`, а затим чворове на путу од `k` до `j`. Да се чвор `k` не би два пута понављао, функцију правимо тако да се у вектор `put` не додаје последњи чвор на најкраћем путу од `i` до `j` (то је чвор `j`) и њега додајемо у функцији омотачу `odrediPut`.

```
// funkcija koja odredjuje najkraci put od cvora i do cvora j, bez cvora j
void odrediPut_(const vector<vector<int>>& medjucvor, int i, int j,
                vector<int>& put) {
  if (medjucvor[i][j] == -1)
    return;
  // put od i do j odgovara direktnoj grani (i,j)
  if (medjucvor[i][j] == i)
```

```
    put.push_back(i);
  else{
    // stampamo put od cvora i do cvora k, gde je k maksimalni redni broj
    // cvora koji pripada tom putu, pa zatim put od cvora k do cvora j
    odrediPut_(medjucvor, i, medjucvor[i][j]);
    odrediPut_(medjucvor, medjucvor[i][j], j);
  }
}

// funkcija koja odredjuje najkraci put od cvora i do cvora j
vector<int> odrediPut(const vector<vector<int>>& medjucvor, int i, int j) {
    vector<int> put;
    odrediPut(medjucvor, i, j, put);
    put.push_back(j);
    return put;
}
```

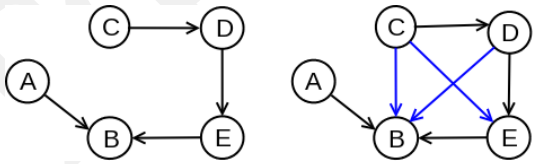
У главној функцији `odstampajPuteve` прво помоћу Флојд-Варшаловог алгоритма имплементираног у функцији `sviNajkraciPutevi` одређујемо најкраће путеве, затим проверавамо да ли у графу постоји циклус негативне тежине (што важи ако и само ако је неки елемент те матрице негативан) и ако није, онда за сваки пар чворова `i` и `j` одређујемо и исписујемо најкраћи пут.

```
// funkcija koja stampa najkrace puteve izmedju svaka dva cvora u grafu
void odstampajPuteve(const vector<vector<Tezina>>& matricaPovezanosti) {
  // primenjujemo Flojd-Varsalov algoritam i odredjujemo najkrace puteve
  vector<vector<int>> duzinaPuta;
  vector<vector<int>> medjucvor;
  sviNajkraciPutevi(matricaPovezanosti, duzinaPuta, medjucvor);

  // proveravamo da li u grafu postoji ciklus negativne duzine
  bool negativniCiklus = false;
  for (i = 0; i < brojCvorova; i++)
    if (najkraciPut[i][i] < 0) {
      negativniCiklus = true;
      break;
    }

  if (negativniCiklus) {
```

```
    cout << "U grafu postoji ciklus negativne duzine" << endl;
  } else {
    int brojCvorova = duzinaPuta.size();
    for (int i = 0; i < brojCvorova; i++)
      for (int j = 0; j < brojCvorova; j++)
        // odredjujemo i ispisujemo najkraci put izmedju svih parova
        // razlicitih cvorova i i j koji su povezani putem
        if (i != j && medjucvor[i][j] != -1) {
          cout << "Najkraci put od cvora " << i << " do cvora " << j << " "
               << "ima duzinu: " << najkraciPut[i][j] << endl;
          for (int cvor : odrediPut(i, j, medjucvor))
            cout << cvor << " ";
          cout << endl;
        }
  }
}
```

2.11.4 Транзитивно затворење и транзитивна редукција

За задати усмерени граф $G = (V, E)$ његово *транзитивно затворење* (енгл. transitive closure) је усмерени граф $G^* = (V, E^*)$ у коме постоји грана од чвора u до чвора v ако и само ако у графу G постоји усмерени пут од чвора u до чвора v. Дакле, скупу грана E треба додати што мањи број грана, тако да добијени нови скуп грана E^* одређује релацију која је транзитивна. На слици 2.86 приказан је један усмерен граф и његово транзитивно затворење.



Слика 2.86: Граф и његово транзитивно затворење: плавом бојом истакнуте су гране којима је полазни граф проширен како би се добило транзитивно затворење графа.

Постоји велики број различитих примена транзитивног затворења, па је важно имати ефикасан алгоритам за његово налажење. На пример, табелу у програму за табеларна израчунавања (нпр. Microsoft Excel) можемо представити у виду усмереног графа: поља табеле одговарају чворовима, а грана од чвора који одговара пољу a до чвора који одговара пољу b постоји ако вредност која се рачуна у пољу b зависи од вредности поља a. Када се измени нека од вредности у табели, потребно је ажурирати вредности свих поља које од ње зависе, односно свих чворова који су достижни из датог поља. Та поља

се могу одредити на основу транзитивног затворења датог графа. Додатно, та поља је потребно ажурирати у тополошком редоследу чворова.

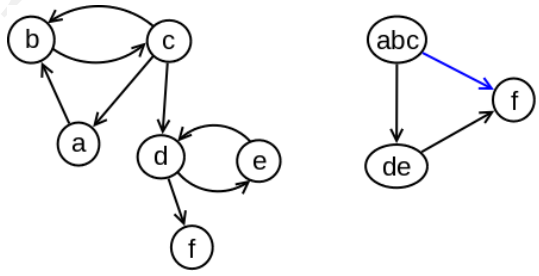
Слично, ако неки скуп аеродрома размотримо као скуп чворова, а постојање директног лета од једног до другог аеродрома представимо усмереном граном између одговарајућих чворова, онда нам транзитивно затворење графа даје информацију о томе са ког аеродрома до ког аеродрома је могуће стићи директним летом или путем више повезаних летова.

Проблем

Одредити транзитивно затворење задатог усмереног графа $G = (V, E)$.

Постоји више начина за рачунање транзитивног затворења датог графа. Један начин је да се из сваког чвора покрене претрага у дубину или ширину и да се сачува информација о свим чворовима достижним из полазног чвора. Овај алгоритам је временске сложености $O(|V| \cdot (|V| + |E|))$ и има добре перформансе ако је граф редак, док за густе графове постаје сложености $O(|V|^3)$.

Ако је транзитивно затворење графа G густ граф, ефикасније је најпре израчунати компоненте јаке повезаности графа G. За свака два чвора из исте компоненте јаке повезаности важи да су међусобно достижни, па у транзитивном затворењу треба да буду повезани гранама. Ако постоји грана (u, v) која повезује чворове из различитих јаких компоненти повезаности, сваки чвор из компоненте којој припада чвор v је достижан из сваког чвора компоненте којој припада чвор u и одговарајућу грану треба додати транзитивном затворењу графа G. Дакле, проблем се своди на проналажење транзитивног затворења компримованог графа, сачињеног од јаких компоненти повезаности, који обично има доста мање чворова и грана (слика 2.87). Ако постоји K компоненти јаке повезаности, сложеност одређивања транзитивног затворења компримованог графа је $O(K^3)$, што може бити знатно мање од $O(|V|^3)$ када је K довољно мање од $|V|$.



Слика 2.87: Граф и транзитивно затворење његовог компримованог графа (плавом бојом означене су гране које су додате у граф).

Трећи начин да решимо овај проблем јесте редукцијом (свођењем) на други проблем. Ако гранама графа доделимо произвољне тежине (на пример, свакој грани доделимо тежину 1) и израчунамо најкраће путеве између свих парова чворова тако добијеног тежинског графа, дужина најкраћег пута између чворова између којих постоји пут ће бити коначна, док ће дужина најкраћег пута између чворова између којих не постоји пут остати $+\infty$. Дакле, проблем одређивања транзитивног затворења, што је у суштини проблем испитивања достижности између чворова, природно се може свести на одређивање дужине најкраћих путева између свака два чвора.

Уместо да директно применимо алгоритам за одређивање свих најкраћих путева у графу, можемо га временски и просторно оптимизовати тако да директно решава проблем транзитивног затворења графа. Приметимо да нас једино интересује да ли је дужина најкраћег пута у графу G' коначна или бесконачна (а у случају када је коначна, не интересује нас њена конкретна вредност, тј. да ли је она једнака 10, 7 или 2). Дакле, уместо да користимо целобројну матрицу у којој се памте дужине најкраћих путева између свака два чвора, можемо користити логичку матрицу која на позицији (i, j) садржи вредност `true` ако је чвор j достижан из чвора i, а иначе `false`. Такође, уместо да користимо аритметичке операције, можемо прећи на логичке операције: уместо операције сабирања дужина користимо логичку конјункцију (дужина пута који се састоји из два дела ће бити коначна ако и само ако су оба дела пута коначна).

```
// funkcija koja za svaka dva cvora utvrdjuje
// da li izmedju njih postoji put
vector<vector<bool>> tranzitivnoZatvorenje(
  const vector<vector<bool>>& matricaPovezanosti) {
  // inicijalizujemo matricu tranzitivnog zatvorenja
  // na matricu povezanosti grafa
  vector<vector<bool>> zatvorenje = matricaPovezanosti;
  int brojCvorova = matricaPovezanosti.size();

  // proveravamo da li postoji put kroz cvor sa oznakom k
  for (int k = 0; k < brojCvorova; k++)
    for (int i = 0; i < brojCvorova; i++)
      for (int j = 0; j < brojCvorova; j++)
        if (zatvorenje[i][k] && zatvorenje[k][j])
          zatvorenje[i][j] = true;

  return zatvorenje;
}
```

Матрица `zatvorenje` на крају извршавања алгоритма кодира информације о достижно-

сти у полазном графу G, односно представља скуп грана у транзитивном затворењу графа G.

Размотримо основни корак алгоритма, наредбу `if`. Она се састоји од две провере: да ли важи `zatvorenje[i,k]` и да ли важи `zatvorenje[k,j]`. Акција се предузима само ако су оба услова испуњена. Ова `if` наредба извршава се $|V|$ пута за сваки пар чворова. Свака поправка ове наредбе водила би битној поправци алгоритма. Морају ли се сваки пут проверавати оба услова? Прва провера зависи само од вредности i и k, а друга зависи само од вредности k и j. Због тога се прва провера може за фиксиране вредности i и k извршити само једном (уместо $|V|$ пута). Наиме,

- ако први услов није испуњен, онда се други не мора проверавати ни за једну вредност j,
- ако је пак први услов испуњен, онда се његова испуњеност не мора поново проверавати за сваку вредност j.

Ова промена је уграђена у побољшани алгоритам чији је кључни фрагмент дат у наставку. Асимптотска сложеност алгоритма остаје непромењена, али се алгоритам у просеку извршава два пута брже.

```
// proveravamo da li postoji put kroz cvor sa oznakom k
for (int k = 0; k < brojCvorova; k++)
  for (int i = 0; i < brojCvorova; i++)
    if (zatvorenje[i][k])
      for (int j = 0; j < brojCvorova; j++)
        if (zatvorenje[k][j])
          zatvorenje[i][j] = true;
```

Овај алгоритам може се даље усавршити. Линија

```
if (zatvorenje[k][j])
   zatvorenje[i][j] = true;
```

може се еквивалентно заменити линијом

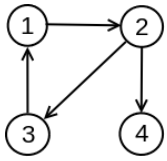
```
zatvorenje[i][j] = zatvorenje[i][j] | zatvorenje[k][j];
```

Запажа се да се после ове замене у унутрашњој петљи алгоритма операција `or` примењује на врсту i и врсту k матрице `zatvorenje`, а резултат је нова врста i. Због тога, ако је $n = |V| \leq 64$, врсте матрице `zatvorenje` представљају низ нула и јединица и могу се тумачити као n битова у бинарној репрезентација целог броја, па се примена операције `or` на врсте може заменити битском `or` операцијом два цела броја, што је n пута брже. Ако је $n > 64$, онда се врста може представити низом 64-битних целих бројева,

па се алгоритам извршава приближно 64 пута брже. Асимптотска сложеност је и даље $O(|V|^3)$, али убрзање за фактор 64 није занемарљиво.

Пример 2.11.4

Размотримо пример израчунавања транзитивног затворења графа са слике 2.88.



Слика 2.88: Пример графа за који је потребно одредити транзитивно затворење.

Он се састоји из наредних корака:

$$k=0\text{:}\ \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \quad k=1\text{:}\ \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \quad k=2\text{:}\ \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$k=3\text{:}\ \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \quad k=4\text{:}\ \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Приметимо да у графу са слике 2.88 не постоји директна грана од чвора 3 *до чвора* 2*, али постоји пут дужине два преко чвора* 1*. Слично, од чвора* 3 *до чвора* 4 *не постоји директна грана, али постоји пут дужине* 3*, преко чворова* 1 *и* 2*.*

Читаоцима се оставља за размишљање питање како би изгледало транзитивно затворење неусмереног графа.

Интересантан проблем у вези са налажењем транзитивног затворења је и проблем одређивања *транзитивне редукције* графа (енгл. transitive reduction). Ова операција представља операцију инверзну операцији проналажења транзитивног затворења графа: циљ је да се за дати усмерени граф конструише усмерени граф са истим скупом чворова и што мањим скупом грана, а да се притом не промени релација достижности. Транзитивно затворење графа G једнако је транзитивном затворењу транзитивне редукције графа G. Иако је транзитивно затворење графа G јединствено одређено, у општем случају граф може имати више различитих транзитивних редукција.

Задатак: Централни чвор

У једној области је неколико места повезано једносмерним или двосмерним путевима. Потребно је одабрати место у коме ће се изградити ватрогасна станица, тако да ватрогасци што брже могу да стигну до свих осталих места. Написати програм који одређује такво место.

Опис улаза

Са стандардног улаза се учитава број градова n, а затим и симетрична матрица A тако да вредност A_{ij} означава време путовања у минутама директним путем од града i до града j. Тамо где директан пут не постоји уписана је вредност -1.

Опис излаза

На стандардни излаз исписати број од 0 до $n-1$, који представља индекс места на ком је најбоље изградити ватрогасну станицу, а затим и број минута који ће ватрогасцима бити довољан да стигну до било ког пожара у тој области (места су мала, па се може занемарити време путовања унутар сваког места).

Пример

Улаз	*Излаз*	*Објашњење*

`4`
` 0  2  8 -1`
`-1  0  3  4`
`-1 -1  0  7`
` 5 -1 -1  0`
	`0 6`	Ако се ватрогасна станица постави у место број 0 (прво место), тада ће ватрогасци моћи да стигну до било ког другог места за највише 6 минута (до места 1 могу да стигну за 2 минута, до места 2 за 5 минута, а до места 3 за 6 минута).

Решење

Чвор који се тражи се понекад назива *средиште графа*, при чему код усмерених графова треба бити обазрив јер се поред места од ког је путовање до свих других места најкраће, може разматрати и место до ког је путовање из свих других места најкраће.

Да би се одредило средиште потребно је израчунати најкраћа растојања између свих парова места, што се може урадити Флојд-Варшаловим алгоритмом. У добијеној матрици је затим потребно одредити максимуме свих врста и позицију оне врсте у којој се налази минимум тако добијених максимума.

У задатом примеру дужине свих најкраћих путева су

$$\begin{matrix} 0 & 2 & 5 & 6 \\ 9 & 0 & 3 & 4 \\ 12 & 14 & 0 & 7 \\ 5 & 7 & 10 & 0 \end{matrix}$$

Сложеношћу алгоритма доминира Флојд-Варшалов аглоритам, па је укупна сложеност $O(n^3)$ — за одређивање максимума и минимума довољно је време $O(n^2)$.

```
// odredjujemo najkrace puteve izmedju svih parova mesta
vector<vector<Tezina>> najkraciPutevi = sviNajkraciPutevi(A);
// globalni minimum (najmanje vreme potrebno da se iz do sada obradjenih
// mesta stigne do drugih mesta)
Tezina min = INF;
// redni broj mesta iz kog se taj minimum  dostize
int pozMin = -1;
// analiziramo sva mesta
for (int i = 0; i < n; i++) {
  // maksimum vrste tj.
  // najvece vreme potrebno da se iz mesta i stigne do svih drugih mesta
  Tezina max = 0;
  for (int j = 0; j < n; j++)
    if (najkraciPutevi[i][j] > max)
      max = najkraciPutevi[i][j];
  // ako se iz ovog mesta moze brze stici do ostalih, azuriramo
  // globalni minimum
  if (max < min) {
    min = max;
    pozMin = i;
  }
}

cout << pozMin << " " << min << endl;
```

3. Алгебарски алгоритми

Увек када извршавамо неку алгебарску операцију, као што је рецимо множење два броја или њихово степеновање, ми, у ствари, извршавамо неки алгоритам. Такве операције користимо као градивне елементе приликом извођења сложенијих алгоритама и често не залазимо дубље у анализу њихове сложености. Међутим, и сами алгоритми сабирања, одузимања, множења, дељења и степеновања бројева (посебно ако су бројеви дати низовима својих цифара) представљају важне алгебарске алгоритме. У алгебарске алгоритме спадају и многи алгоритми са којима смо се раније сусретали, као што су израчунавање вредности броја на основу датих цифара (у некој основи) или, насупрот томе, одређивање цифара броја на основу његове вредности, рачунање највећег заједничког делиоца два дата броја, затим разни алгоритми над полиномима као што су израчунавање вредности полинома и множење полинома. У овом поглављу бавићемо се алгебарским алгоритмима са којима се до сада нисмо сусрели. Многи од њих играју важну улогу у области криптографије, али и у другим областима, попут алгебарске теорије бројева и квантног рачунарства.

Алгебарски алгоритми који се разматрају у овом материјалу раде са природним бројевима и у великој мери се заснивају на особинама дељивости природних бројева (нпр. појму простог броја).

У овом поглављу бавићемо се разним темама из теорије бројева.

- Прво ћемо се подсетити **Еуклидовог алгоритма** за рачунања **највећег заједничког делиоца** два броја, а затим и приказати његову проширену варијанту која омогућава да се израчунати НЗД представи као целобројна линеарна комбинација полазних бројева.

- Размотрићемо затим алгоритме **факторизације** броја: **алгоритам заснован на дељењу** и **Фермаов алгоритам**.
- Важна примена факторизације је при рачунању вредности **мултипликативних функција**. У тексту ће бити размотрена **Ојлерова функција** φ која рачуна број бројева мањих од датог броја, који су узајамно прости са њим. Затим ћемо размотрити функције σ_l **збира** l**-тих степена делилаца** датог броја (функција σ_0 рачуна број, а σ_1 збир делилаца).
- Позабавићемо се затим основама **модуларне аритметике**. Обновићемо релације **конгруенција по модулу**, описати све **аритметичке операције по модулу** (укључујући и **брзо степеновање** и одређивање **модуларног мултипликативног инверза**) а затим увести **модуларне групе** (адитивне и мултипликативне). Доказаћемо **Ојлерову теорему** и **малу Фермаову теорему** и навести неке њихове примене. Доказаћемо **Кинеску теорему о остацима** и описати неколико алгоритама за решавање система конгруенција.
- Уведене концепте и алгоритме ћемо илустровати на примеру **алгоритма RSA**, који има примену у криптографији.
- Коначно, представићемо **алгоритам FFT** (брзу Фуријеову трансформацију) који представља један од најзначајних алгоритама данашњице и илустровати како се он може применити на брзо множење полинома. Размотрићемо и његову варијанту у модуларној аритметици (**алгоритам NTT**).

3.1 Еуклидов алгоритам

Ако су a и b цели бројеви и важи $b \neq 0$, кажемо да је број q *количник*, а број r *остатак* при дељењу броја a бројем b ако и само ако важи $a = q \cdot b + r$ и $0 \leq r < y$. Бројеви q и r су овим условом јединствено одређени. Писаћемо $q = a \operatorname{div} b$ и $r = a \bmod b$. У програмским језицима се операција mod обично означава симболом `%`, док је div целобројно дељење уз заокруживање наниже (што је подразумевано понашање оператора `/` примењеног на целе бројеве), односно $a \operatorname{div} b = \left\lfloor \frac{a}{b} \right\rfloor$. Неки програмски језици имају посебан оператор за израчунавање целобројног количника (нпр. оператор `//` у језику Python). Треба бити обазрив јер различити програмски језици другачије третирају случајеве када је неки од бројева a или b негативан (у неким језицима оператор `%` израчунава позитиван, а у неким негативан остатак, што је у супротности са дефиницијом остатка коју смо навели).

Број a је *дељив* бројем $b \neq 0$, што означавамо $b|a$, ако и само ако је $a \bmod b = 0$. Број b је *делилац* броја a, а број a је *садржалац* броја b.

Бројеви који имају тачно два различита делиоца називају се *прости*: број $n > 1$ је прост ако су му једини делиоци 1 и n. Природни бројеви који имају више од два различита делиоца су *сложени*. Број 1 није ни прост ни сложен. Простих бројева има бесконачно много. Наиме, ако претпоставимо да их има коначно много, број који би се добио множењем свих простих бројева и додавањем броја 1 на тај производ био би већи од свих простих бројева, а не би био дељив ниједним од њих, те би морао да буде прост, чиме бисмо добили контрадикцију. Сви бројеви мањи од датог броја n се могу ефикасно одредити алгоритмом познатим под именом *Ератостеново*[1] *сито*.

Највећи заједнички делилац (НЗД) бројева a и b је највећи број $d = \text{nzd}(a, b)$ такав да $d|a$ и $d|b$, а *најмањи заједнички садржалац* (НЗС) бројева a и b је најмањи број $s = \text{nzs}(a, b)$ такав да $a|s$ и $b|s$. Подсетимо се основног Еуклидовог[2] алгоритма за одређивање највећег заједничког делиоца бројева a и b.

3.1.1 Основни Еуклидов алгоритам

Размотримо проблем проналажења НЗД два броја.

Проблем

За дате ненегативне целе бројеве a и b који нису истовремено једнаки нули израчунати $\text{nzd}(a, b)$.

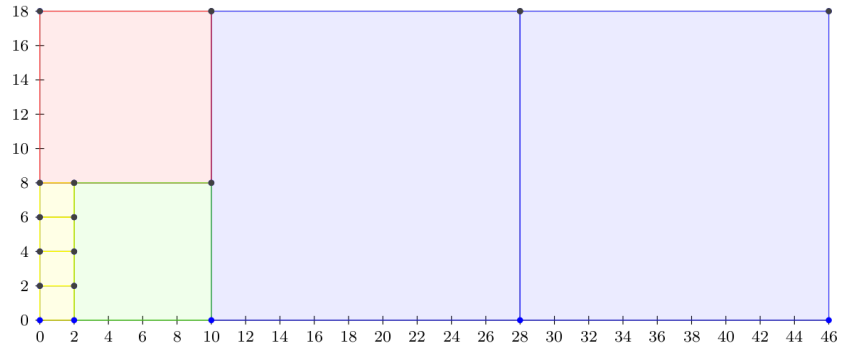
Разматрамо чувени *Еуклидов алгоритам*. Алгоритам се може илустровати и геометријски и у директној је вези са проблемом одређивања максималне димензије једнаких квадрата којима може да се поплоча правоугаоник чије су дужине страница a и b.

Пример 3.1.1

Нека је дат правоугаоник димензије $a = 46$ и $b = 18$, приказан на слици 3.1. Тада се прво из њега могу исећи два квадрата димензије 18×18 и остаје нам правоугаоник димензије 18×10. Уколико неким мањим квадратима успемо да поплочамо тај преостали правоугаоник, тим квадратима ћемо моћи да поплочамо и ове квадрате димензије 18×18 (јер ће димензија тих малих квадрата делити број 18 једнак једној димензији преосталог правоугаоника), па ћемо самим тим моћи да поплочамо и цео полазни правоугаоник димензија 46×18. Од правоугаоника димензије 18×10 можемо исећи квадрат димензије 10×10 и преостаје нам правоугаоник димензије 10×8. Поново, квадратићи којима се може поплочати тај преостали правоугаоник биће такви да се њима може поплочати и исечени квадрат димензије 10×10. Од тог правоугаоника исецамо квадрат димензије 8×8 и добијамо правоугаоник димензије 8×2. Њега

[1] Ератостен (гр. Ἐρατοσθένης), (276-195. пне), грчки математичар.
[2] Еуклид (гр. Εὐκλείδης), (око 300. пне), грчки математичар.

можемо разложити на четири квадрата димензије 2×2 *и то је највећа димензија квадрата којима се може поплочати полазни правоугаоник.*



Слика 3.1: Поплочавање правоугаоника квадратима.

Имплементација која директно прати претходну дефиницију је рекурзивна, а могуће је једноставно направити и итеративну имплементацију, у којој се у сваком кораку вредност већег од два броја замењује њиховом разликом, све док се бројеви не изједначе.

Рекурзивно се алгоритам може изразити на следећи начин.

```
int nzd(int a, int b) {
   if (a > b) return nzd(a-b, b);
   if (a < b) return nzd(a, b-a);
   return a;
}
```

Рекурзија се из претходног алгоритма може јако лако уклонити.

```
int nzd(int a, int b) {
   while (a != b) {
      if (a > b)
         a -= b;
      else
         b -= a;
   }
   return a;
}
```

Докажимо формално коректност алгоритма (доказујемо само коректност рекурзивне варијанте, пошто јој је итеративна варијанта еквивалентна).

Теорема 3.1.1 **[Коректност Еуклидовог алгоритма са одузимањем]**

За дате природне бројеве a и b, који нису истовремено једнаки нули, Еуклидов алгоритам са одузимањем коректно израчунава њихов НЗД.

Доказ. Коректност рекурзивне имплементације се једноставно може доказати индукцијом.

Базу чини случај $a = b$. Пошто су бројеви различити од нуле, очигледно је да је њихов НЗД једнак $a = b$.

Претпоставимо да је $a > b$. Тада је $\text{nzd}(a, b) = \text{nzd}(a - b, b)$. Заиста, ако неки број d дели бројеве a и b, тада он дели и $a - b$. Дакле, $d = \text{nzd}(a, b)$ сигурно дели и $a - b$ и b. Ако он не би био НЗД бројева $a-b$ и b, тада би постојао неки већи број d' који би делио и $a - b$ и b. Међутим, тада би d' делио и збир $a = (a - b) + b$, па би d' био и делилац бројева a и b, који је већи од d, што је контрадикција са тим да је d НЗД бројева a и b. На основу индуктивне хипотезе $\text{nzd}(a - b, b)$ умемо исправно да израчунамо.

Случај $a < b$ је потпуно симетричан. □

Овај се алгоритам лако може оптимизовати. Размотримо одређивање НЗД два броја на једном примеру.

Пример 3.1.2

$\text{nzd}(279, 45) = \text{nzd}(234, 45) = \text{nzd}(189, 45) = \text{nzd}(144, 45) = \text{nzd}(99, 45) = \text{nzd}(54, 45) = \text{nzd}(9, 45) = \text{nzd}(9, 36) = \text{nzd}(9, 27) = \text{nzd}(9, 18) = \text{nzd}(9, 9) = 9.$

Можемо приметити дугачак низ корака у којима се мало по мало од броја 279 одузима број 45, све док се не дође до броја који је мањи од броја 45. За тим нема потребе, јер знамо да ће се после тог дугог низа корака поступак зауставити када нам један аргумент буде баш 45, а други буде једнак остатку при дељењу броја 279 бројем 45, а то је број 9. Дакле, уместо да итеративно тај остатак рачунамо узастопним одузимањима, бољи приступ је да применимо дељење и у једном кораку га израчунамо као остатак при дељењу. Ако сличан принцип применимо на бројеве 9 и 45, доћи ћемо до тога да ће нам остати број 9 и остатак при дељењу та два броја, што је нула. То није баш у потпуности идентично као у случају одузимања, где смо се зауставили код пара $(9, 9)$, међутим, сасвим је исправно и може се сматрати продужењем претходног поступка у ком би се пре пријављивања резултата урадило још једно одузимање и дошло се до тога да је један од бројева једнак нули, када је НЗД једнак другом броју.

Бржи Еуклидов алгоритам, у ком се користи дељење, заснован је на следећим тврђењима.

- за $a \neq 0$ важи $\text{nzd}(a, 0) = a$.
- за $b \neq 0$ важи $\text{nzd}(a, b) = \text{nzd}(b, a \bmod b)$.

Приметимо да нема потребе анализирати који је број мањи, а који је већи. Ако је $a < b$, тада важи $a \bmod b = a$, па се у првом кораку добија да је $\text{nzd}(a, b) = \text{nzd}(b, a)$. Пошто је $a \bmod b < b$, једном када је први аргумент већи од другог, то ће тако остати до краја.

И Еуклидов алгоритам са дељењем, дакле, допушта једноставну рекурзивну имплементацију.

```
int nzd(int a, int b) {
   if (b == 0)
      return a;
   return nzd(b, a % b);
}
```

Еуклидов алгоритам се може формулисати и имплементирати и итеративно, тако што у петљи која се извршава све док је број b већи од нуле пар променљивих (a, b) замењујемо вредностима $(b, a \bmod b)$. На крају петље је $b = 0$, тако да као резултат можемо пријавити текућу вредност броја a.

```
int nzd(int a, int b) {
   while (b != 0) {
      int ost = a % b;
      a = b;
      b = ost;
   }
   return a;
}
```

Докажимо формално коректност алгоритма (доказујемо само коректност рекурзивне варијанте, пошто јој је итеративна варијанта еквивалентна).

Теорема 3.1.2 **[Коректност Еуклидовог алгоритма са дељењем]**

За дате природне бројеве a и b, који нису истовремено једнаки нули, Еуклидов алгоритам са дељењем коректно израчунава њихов НЗД.

Доказ. Коректност рекурзивне варијанте се доказује математичком индукцијом.

Базу чини случај $\text{nzd}(a, 0)$, за $a \neq 0$. Пошто важи $a|0$ и $a|a$, а не постоји ни један број $a' > a$ такав да $a'|a$, важи да је $\text{nzd}(a, 0) = a$.

На основу дефиниције целобројног дељења важи $a = (a \operatorname{div} b) \cdot b + (a \bmod b)$. Обележимо са d НЗД бројева b и $a \bmod b$. Да бисмо доказали да је он уједно НЗД бројева a и b довољно је доказати да он дели та два броја и да сваки број који дели та два броја дели њега.

- Пошто број d дели бројеве b и $a \bmod b$, он дели оба сабирка на десној страни, па зато дели и њихов збир који је једнак a, те зато дели и a и b.
- Даље, ако неки број d' дели бројеве a и b, он мора делити и број $a \bmod b$ (јер се он може исказати као разлика два броја дељивих бројем d'), па пошто је d' делилац бројева b и $a \bmod b$, он мора делити и њихов НЗД, тј. мора делити број d.

Пошто $\operatorname{nzd}(b, a \bmod b)$ можемо израчунати на основу индуктивне хипотезе, тврђење је доказано. □

Еуклидов алгоритам који је заснован на дељењу можемо представити и на следећи начин:

$$\begin{array}{rcll}
r_0 & = & a & \\
r_1 & = & b & \\
r_2 & = & r_0 \bmod r_1 = r_0 - q_1 r_1, & 0 < r_2 < r_1 \\
\ldots & & & \\
r_{i+1} & = & r_{i-1} \bmod r_i = r_{i-1} - q_i r_i, & 0 < r_{i+1} < r_i \\
\ldots & & & \\
r_k & = & r_{k-2} \bmod r_{k-1} = r_{k-2} - q_{k-1} r_{k-1}, & 0 < r_k < r_{k-1} \\
r_{k+1} & = & r_{k-1} \bmod r_k = r_{k-1} - q_k r_k, & r_{k+1} = 0
\end{array}$$

Вредност r_k је НЗД бројева a и b. Заиста, пошто је $r_{k+1} = 0$, важи $r_{k-1} = q_k r_k$, па број r_k дели r_{k-1}. Међутим, пошто је $r_{k-2} = q_{k-1} r_{k-1} + r_k$, r_k дели и r_{k-2}. Сличним резоновањем, уназад, може се закључити да r_k дели и r_1 и r_0, тј. a и b. Обратно, ако неки број дели и a и b онда он дели и r_0 и r_1, а пошто је $r_2 = r_0 - q_1 r_1$, он дели и r_2. Сличним резоновањем, унапред, може се закључити да тај број мора делити и r_k. Стога је r_k НЗД бројева a и b.

Дакле, полазећи од бројева $r_0 = a$ и $r_1 = b$, израчунава се остатак $r_2 = r_0 \bmod r_1$, затим остатак $r_3 = r_1 \bmod r_2$, итд. На тај начин добија се опадајући низ остатака r_i за који важи:

$$r_{i-1} = q_i r_i + r_{i+1}, \quad 0 \leq r_{i+1} < r_i, \text{ za } i = 1, 2, \ldots, k \tag{3.1}$$

При дељењу r_{i-1} са r_i количник је q_i, а остатак r_{i+1}. Добијени низ r_i је коначан јер је опадајући (важи $r_{i+1} < r_i$ за свако i), а састоји се од природних бројева. Ако је $r_{k+1} = 0$ и $r_k \neq 0$ последњи члан овог низа различит од нуле, како је

$$\text{nzd}(r_0, r_1) = \text{nzd}(r_1, r_2) = \ldots = \text{nzd}(r_{k-1}, r_k) = \text{nzd}(r_k, 0) = r_k,$$

закључујемо да је $d = \text{nzd}(a, b)$ управо једнако r_k, последњем остатку у низу остатака који је различит од нуле.

Што се тиче имплементације, у сваком кораку алгоритма одржавамо две узастопне вредности низа остатака. У почетку, то су чланови r_0 и r_1, тј. оригиналне вредности a и b. Важи да је $q_1 = a \operatorname{div} b$, а $r_2 = a \bmod b$. У другом кораку промењиве a и b треба да имају вредности чланова r_1 и r_2, што значи да се пар промењивих a, b замењује вредностима b и $a \bmod b$. Поступак се наставља све док пар узастопних вредности не постане r_k, r_{k+1}, тј. пошто пар узастопних вредности одржавамо у промењивама a и b, док b не постане нула и тада је НЗД који је једнак r_k садржан у промењивој a.

Приметимо да се после највише једног корака осигурава да је $a > b$ (јер се у сваком кораку пар (a, b) замењује паром $(b, a \bmod b)$, а увек важи да је $a \bmod b < b$). После било које две итерације се од пара (a, b) долази до пара $(a \bmod b, b \bmod (a \bmod b))$ (наравно, под претпоставком да је $b \neq 0$ и да је $a \bmod b \neq 0$). Докажимо да је $a \bmod b < a/2$. Ако је $b \leq a/2$, тада је $a \bmod b < b \leq a/2$. У супротном, за $b > a/2$ важи да је $a \bmod b = a - b < a/2$. Зато се први аргумент после свака два корака смањује бар двоструко. До вредности 1 први аргумент стиже у логаритамском броју корака у односу на већи од полазна два броја и тада други број сигурно достиже нулу (јер је строго мањи од првог) и поступак се завршава. Дакле, сложеност алгоритма је логаритамска у односу на већи од два броја, односно $O(\log(a+b))$. Ако се сложеност рачуна у односу на број цифара броја (што је величина улаза), онда је она заправо линеарна. Ово указује на то да је проблем одређивања НЗД веома лак проблем и да се ефикасно може решити и за огромне бројеве (бројеве и са неколико хиљада цифара).

3.1.2 Проширени Еуклидов алгоритам

Уз малу допуну, Еуклидов алгоритам се може искористити и за решавање следећег проблема.

Проблем

Највећи заједнички делилац d два природна броја a и b изразити као њихову целобројну линеарну комбинацију. Другим речима, одредити целе бројеве x и y, тако да важи $d = \text{nzd}(a, b) = x \cdot a + y \cdot b$.

Пример 3.1.3

Највећи заједнички делилац бројева $a = 10$ и $b = 18$ је $d = 2$ и он се може представити као њихова целобројна линеарна комбинација на следећи начин: $2 = 2 \cdot 10 - 1 \cdot 18$.

Бројеви x и y постоје на основу тврђења које је познато као *Безуов став*, а које ћемо у наставку конструктивно доказати на два различита начина (из којих ће произаћи два различита алгоритма за израчунавање коефицијената x и y).

Теорема 3.1.3 **[Безуов став]**

Ако су a и b два природна броја (не истовремено једнака нули) и важи да је $\text{nzd}(a, b) = d$, *тада постоје цели бројеви x и y такви да је $x \cdot a + y \cdot b = d$.*

3.1.2.1 Представљање НЗД преко узастопних остатака

Пошто је $r_0 = a$ и $r_1 = b$, проблем можемо формулисати нешто другачије: задатак је број $d = \text{nzd}(a, b)$ изразити у облику целобројне линеарне комбинације остатака r_0 и r_1, односно као $d = r_k = x \cdot r_0 + y \cdot r_1$, где је r_k последњи члан низа остатака различит од нуле. Овако формулисан проблем се може решити индукцијом.

Полази се од базног случаја који одговара представљању броја d као целобројне линеарне комбинације два узастопна остатка r_{k-2} и r_{k-1}: $d = r_k = r_{k-2} - q_{k-1} r_{k-1}$. Овај израз је еквивалентан претпоследњем дељењу у Еуклидовом алгоритму (израз (3.1)) за $i = k - 1$.

Корак индукције био би да се полазећи од израза за d

$$d = x' \cdot r_i + y' \cdot r_{i+1} \tag{3.2}$$

као целобројне линеарне комбинације остатака r_i и r_{i+1}, број d изрази као целобројна линеарна комбинација остатака r_{i-1} и r_i, односно у облику $d = x'' \cdot r_{i-1} + y'' \cdot r_i$. Ово се постиже заменом вредности r_{i+1} у једнакости (3.2) са $r_{i-1} - q_i r_i$ из једнакости (3.1). На тај начин се добија

$$d = x' \cdot r_i + y' \cdot r_{i+1} = x' \cdot r_i + y' \cdot (r_{i-1} - q_i r_i) = y' \cdot r_{i-1} + (x' - q_i y') \cdot r_i \tag{3.3}$$

односно:

$$\begin{aligned} x'' &= y' \\ y'' &= x' - q_i y' \end{aligned}$$

тј. изразили смо d у облику целобројне линеарне комбинације остатака r_{i-1} и r_i. Дакле, индукцијом по i, $i = k-2, k-3, \ldots, 1$, доказано је да се d може изразити као целобројна линеарна комбинација произвољна два узастопна члана r_i и r_{i+1} низа остатака. Специјално, за $i = 0$, добија се тражени израз. Ова верзија Еуклидовог алгоритма се понекад назива *проширени Еуклидов алгоритам* (енгл. extended Euclidean algorithm).

Пример 3.1.4

Потребно је одредити целе бројеве x и y тако да важи $3 = 33x + 24y$. С обзиром на то да је $\text{nzd}(33, 24) = 3$, *можемо искористити претходни поступак за одређивање бројева x и y. Приликом израчунавања највећег заједничког делиоца бројева 33 и 24 имамо наредни низ једнакости:* $\text{nzd}(33, 24) = \text{nzd}(24, 9) = \text{nzd}(9, 6) = \text{nzd}(6, 3) = \text{nzd}(3, 0) = 3$. *Пратећи уназад низ дељења са остатком добијамо:* $d = 3 = 9 - 6 = 9 - (24 - 2 \cdot 9) = 3 \cdot 9 - 24 = 3 \cdot (33 - 24) - 24 = 3 \cdot 33 - 4 \cdot 24$, *односно* $x = 3, y = -4$.

Претходна индуктивна конструкција погодна је за рекурзивну имплементацију јер нове вредности променљивих x и y добијамо на основу претходних. У наредној имплементацији, вредност коју функција враћа јесте највећи заједнички делилац датих бројева, док вредности коефицијената x и y враћамо преко референци, кроз листу параметара функције.

Приметимо да је база индукције у наредној имплементацији случај $d = r_k = 1 \cdot r_k + 0 \cdot r_{k+1}$.

```
// funkcija koja vraca nzd brojeva a i b i racuna koeficijente x i y
int nzdProsireni(int a, int b, int &x, int &y) {
  // ako je b jednako 0, onda je nzd(a, b) = a
  if (b == 0) {
    x = 1;
    y = 0;
    return a;
  }

  int x1, y1;
  // rekurzivno resavamo problem za naredna dva elementa u nizu ostataka,
  // a to su brojevi b i a%b
  int nzd = nzdProsireni(b, a % b, x1, y1);
  // azuriramo koeficijente
  x = y1;
```

```
    y = x1 - (a / b) * y1;
    return nzd;
}
```

Приметимо да се у овој верзији алгоритама два пута пролази кроз количнике и остатке. У пролазу унапред се одређује вредност НЗД, а затим се у пролазу уназад одређују коефицијенти (у рекурзивној имплементацији тај други пролаз се извршава тек након што се стигне до излаза из рекурзије). Зато се овај алгоритам теже имплементира нерекурзивно (и меморијска сложеност му је, као и временска, $\log(a+b)$, јер захтева истовремено чување више стек оквира).

3.1.2.2 Представљање остатака преко a и b

У претходном извођењу смо све време последњи остатак $d = r_k$ представљали као целобројну линеарну комбинацију два суседна елемента из низа остатака почев од r_{k-1} и r_{k-2}. Алтернативно, могуће је једну по једну вредност из низа остатака почев од r_0 представити као целобројну линеарну комбинацију бројева a и b и, на крају, и број r_k представити на овај начин. Наиме, пошто је $r_0 = a$ и $r_1 = b$, важи следећа база индукције:

$$\begin{aligned} r_0 &= 1 \cdot a + 0 \cdot b \\ r_1 &= 0 \cdot a + 1 \cdot b \end{aligned}$$

односно за $r_0 = a$ важи $x_0 = 1$ и $y_0 = 0$, а за $r_1 = b$ важи $x_1 = 0$ и $y_1 = 1$.

Желимо да r_{i+1} представимо као целобројну линеарну комбинацију бројева a и b, ако знамо да важи:

$$\begin{aligned} r_{i-1} &= x_{i-1} \cdot a + y_{i-1} \cdot b \\ r_i &= x_i \cdot a + y_i \cdot b \end{aligned}$$

На основу једнакости (3.1), тј. $r_{i+1} = r_{i-1} - q_i \cdot r_i$ добијамо:

$$r_{i+1} = (x_{i-1} \cdot a + y_{i-1} \cdot b) - q_i(x_i \cdot a + y_i \cdot b) = (x_{i-1} - q_i x_i) \cdot a + (y_{i-1} - q_i y_i) \cdot b$$

Дакле, вредности x_{i+1} и y_{i+1} можемо израчунати на основу претходних вредности x_{i-1} и x_i, односно y_{i-1} и y_i, тј. важе наредне једнакости:

$$\begin{aligned} x_{i+1} &= x_{i-1} - q_i x_i \\ y_{i+1} &= y_{i-1} - q_i y_i \end{aligned}$$

На основу ових једакости долази се до наредне C++ имплементације итеративног алгоритма за решавање полазног проблема.

```
// funkcija koja vraca nzd brojeva a i b i racuna koeficijente x i y
int nzdProsireni(int a, int b, int &x, int &y) {
  // vrednost x i y za r_0 = a
  int x_preth = 1;
  int y_preth = 0;
  // vrednost x i y za r_1 = b
  int x_tek = 0;
  int y_tek = 1;

  while (b != 0) {
    // azuriramo vrednosti za a i b
    // kao u standardnom Euklidovom algoritmu
    int q = a/b;
    int r = a - q * b;
    a = b;
    b = r;

    // azuriramo tekucu i prethodnu vrednost niza x
    int xpom = x_preth - q * x_tek;
    x_preth = x_tek;
    x_tek = xpom;

    // azuriramo tekucu i prethodnu vrednost niza y
    int ypom = y_preth - q * y_tek;
    y_preth = y_tek;
    y_tek = ypom;
  }

  // postavljamo konacne vrednosti za x i y
  // kao vrednosti x i y za r_k
  x = x_preth;
  y = y_preth;

  // vracamo a kao nzd brojeva
  return a;
}
```

Пример 3.1.5

Применимо алгоритам на бројеве 33 *и* 24.

$$
\begin{array}{rcrcrcrcrcrcrcr}
33 & & & & & & & & & & & = & 1\cdot 33 & + & 0\cdot 24 \\
24 & & & & & & & & & & & = & 0\cdot 33 & + & 1\cdot 24 \\
9 & = & 33 & - & 1\cdot 24 & = & (1\cdot 33+0\cdot 24) & - & 1\cdot(0\cdot 33+1\cdot 24) & = & 1\cdot 33 & - & 1\cdot 24 \\
6 & = & 24 & - & 2\cdot 9 & = & (0\cdot 33+1\cdot 24) & - & 2\cdot(1\cdot 33-1\cdot 24) & = & -2\cdot 33 & + & 3\cdot 24 \\
3 & = & 9 & - & 1\cdot 6 & = & (1\cdot 33-1\cdot 24) & - & 1\cdot(-2\cdot 33+3\cdot 24) & = & 3\cdot 33 & - & 4\cdot 24 \\
0 & = & 6 & - & 2\cdot 3 & & & & & & & & &
\end{array}
$$

Везе између (x_{i+1}, y_{i+1}) и (x_i, y_i) су биле изажене на начин погодан имплементацији (наредни коефицијенти су били изражени преко претходних). Приметимо да се ове везе могу изразити и на следећи начин, тако да је облик једнакости исти и за количнике и остатке r и за коефицијенте x и y:

$$
\begin{array}{rcrcr}
r_{i-1} & = & q_i r_i & + & r_{i+1} \\
x_{i-1} & = & q_i x_i & + & x_{i+1} \\
y_{i-1} & = & q_i y_i & + & y_{i+1}
\end{array}
$$

При том у свакој колони у важе једнакости истог облика:

$$
\begin{array}{rcrcr}
r_{i-1} & = & x_{i-1}a & + & y_{i-1}b \\
r_i & = & x_i a & + & y_i b \\
r_{i+1} & = & x_{i+1}a & + & y_{i+1}b
\end{array}
$$

Број операција које се извршавају у проширеном Еуклидовом алгоритму пропорционалан је броју операција у стандардном Еуклидовом алгоритму, тј. износи $O(\log(a+b))$, па је и ово изразито ефикасан алгоритам. Приметимо и да се у овој варијанти алгоритма врши само један пролаз, па се након одређивања наредних вредности коефицијената, претходне могу заборавити. Меморијска сложеност је, стога, константна (за разлику од претходне, рекурзивне варијанте која због чувања стек оквира захтева простор $O(\log(a+b))$).

Из претходно описаних разлога ова (једнопролазна) варијанта алгоритма је пожељнија од претходне (двопролазне), па стога неки аутори под називом *проширени Еуклидов алгоритам* подразумевају само ову варијанту.

3.1.3 Решавање линеарних Диофантових једначина

Једна од примена Безуовог става и проширеног Еуклидовог алгоритма је у решавању једне класе Диофантових[3] једначина, тзв. *линеарних Диофантових једначина* које су облика $a \cdot x + b \cdot y = c$.

Проблем

За дате ненегативне целе бројеве a и b (који нису истовремено једнаки нули) и ненегативан цео број c одредити целе бројеве x и y тако да важи $a \cdot x + b \cdot y = c$.

Безуов[4] став даје једно решење једначине $ax + by = d$, за $d = \text{nzd}(a, b)$. Наредни став говори о структури свих решења.

Лема 3.1.1 **[Сва решења Безуовог идентитета]**

Ако је (x_0, y_0) једно решење једначине $ax + by = d$, где је $d = \text{nzd}(a, b)$, онда су сва решења облика $x = x_0 - k\frac{b}{d}$, $y = y_0 + k\frac{a}{d}$, за $k \in \mathbb{Z}$. Постоје тачно два „мала" пара решења, за које је $|x| \leq |b/d|$ и $|y| \leq |a/d|$. Једнакост важи ако и само ако је један од бројева a или b умножак оног другог.

Доказ. Пошто је $ax_0 + by_0 = d = ax + by$, важи $a(x_0 - x) = b(y - y_0)$. Ова се једнакост може поделити са d и и добија се $\frac{a}{d}(x_0 - x) = \frac{b}{d}(y - y_0)$. Бројеви $\frac{a}{d}$ и $\frac{b}{d}$ су узајамно прости па мора да постоји цео број k такав да је $x_0 - x = k\frac{b}{d}$, а $y - y_0 = k\frac{a}{d}$. Одатле се добија и опште решење:

$$(x, y) = \left(x_0 - k\frac{b}{d}, y_0 + k\frac{a}{d}\right)$$

„Мали" парови решења се добијају тако што се за k узме било који од два цела броја који су најближи вредности $x_0\frac{d}{b}$. □

Напоменимо да проширени Еуклидов алгоритам увек враћа један од два „мала" пара решења о којима се говори у претходној леми.

[3] Диофант (гр. Διόφαντος), (око 250.), грчки математичар.
[4] Етјен Безу (фр. Étienne Bézout), (1730-1783), француски математичар.

Пример 3.1.6

Размотримо сада једначину $4x + 10y = 2$. *Важи* $d = \text{nzd}(4, 10) = 2$, *па коришћењем проширеног Еуклидовог алгоритма највећи заједнички делилац* $d = 2$ *бројева* $a = 4$ *и* $b = 10$ *изражавамо као њихову целобројну линеарну комбинацију:* $x_0 a + y_0 b = d$, *тј.* $-2 \cdot 4 + 1 \cdot 10 = 2$. *Остала решења једначине су облика* $x = x_0 - k \cdot b/d = -2 - 5k$ *и* $y = y_0 + k \cdot a/d = 1 + 2k$. *Заиста, важи* $4x + 10y = 4(-2 - 5k) + 10(1 + 2k) = (-8 + 10) + (20k - 20k) = 2$. *„Мала" решења су она за која важи* $|x| \leq |b/d| = 5$ *и* $|y| \leq |a/d| = 2$ *и то су једино* $(-2, 1)$ *и* $(3, -1)$.

Теорема 3.1.4 **[Постојање решења линеарне диофантске једначине]**

Једначина $a \cdot x + b \cdot y = c$ *има бар једно решење ако и само ако важи да број* $d = \text{nzd}(a, b)$ *дели број* c.

Доказ. Наиме, пошто d дели леву страну једначине, мора да дели и десну, па је овај услов потребан. Докажимо да је и довољан, тако што ћемо описати поступак конструкције решења. Наиме, ако је овај услов испуњен, једно од решења ове једначине лако се добија наредним поступком: најпре се d изрази у облику $d = x' \cdot a + y' \cdot b$ а затим се множењем леве и десне стране једнакости целим бројем c/d добија:

$$c = \frac{x'c}{d} \cdot a + \frac{y'c}{d} \cdot b$$

тј. једно решење полазне једначине представља пар

$$(x_0, y_0) = \left(\frac{x'c}{d}, \frac{y'c}{d}\right)$$

Структура свих решења следи из леме 3.1.1, тј. сва решења су описана паром:

$$(x, y) = \left(\frac{x'c - kb}{d}, \frac{y'c + ka}{d}\right)$$

□

Пример 3.1.7

Размотримо једначину $4x + 10y = 7$. *Важи да је* $\text{nzd}(4, 2) = 2$, *међутим* 2 *не дели десну страну једнакости, тј. број* 7, *те ова Диофантова једначина нема решења.*

Пример 3.1.8

Размотримо сада једначину $4x + 10y = 8$. *Дакле,* $a = 4$, $b = 10$ *и* $c = 8$. *Важи* $d = \text{nzd}(4, 10) = 2$ *и* $2 \mid 8$, *па пошто* $d \mid c$, *ова једначина има решења. Једно њено решење можемо добити на следећи начин: коришћењем проширеног Еуклидовог алгоритма највећи заједнички делилац бројева* $a = 4$ *и* $b = 10$ *изражавамо као њихову целобројну линеарну комбинацију:* $x'a + y'b = d$, *тј.* $-2 \cdot 4 + 1 \cdot 10 = 2$, *а затим обе стране једнакости множимо бројем* $c/d = 8/2 = 4$ *чиме добијамо:* $x_0 a + y_0 b = c$, *тј.* $-8 \cdot 4 + 4 \cdot 10 = 8$. *Дакле, једно решење ове једначине је* $x_0 = -8$ *и* $y_0 = 4$. *Остала решења су облика* $x = x_0 - k \cdot b/d = -8 - 5k$ *и* $y = y_0 + k \cdot a/d = 4 + 2k$. *Заиста, важи* $4x + 10y = 4(-8 - 5k) + 10(4 + 2k) = (-32 + 40) + (20k - 20k) = 8$.

Даље примене проширеног Еуклидовог алгоритма биће приказане у наредним поглављима (у поглављу 3.4 овај алгоритам се примењује на проблем одређивања модуларног мултипликативног инверза, а у поглављу 3.4.5 на решавање система конгруенција применом Кинеске теореме о остацима).

Задатак: Исти остаци свих бројева

Дат је низ природних бројева $a_1, \ldots, a_n$. Одредити све могуће вредности d такве да сви елементи при дељењу са d имају исти остатак.

Опис улаза

Са стандардног улаза се учитава дужина низа n ($2 \leq n \leq 100$), а након тога у наредном реду и елементи низа (природни бројеви мањи од милијарду, раздвојени са по једним размаком). Претпоставља се да нису сви учитани бројеви исти.

Опис излаза

На стандардни излаз исписати све могуће вредности d.

Пример 1

Улаз	*Излаз*
5 5 17 23 14 83	3

Пример 2

Улаз	*Излаз*
3 12 36 48	2 3 4 6 12

Решење

Груба сила

Решење грубом силом подразумева да се испитају све могуће вредности d. Ниједан број d већи од максималног броја у низу a (обележимо га са M) не може бити решење

(тада би остаци били тачно елементи низа a, а то није могуће, јер нису сви елементи у низу a једнаки). Дакле, проверавамо редом све вредности d од 2 до M и за сваки линеарном претрагом проверавамо да ли сви елементи низа a дају исти остатак.

Сложеност овог алгоритма је $O(M \cdot n)$, где је M вредност највећег броја у низу. Пошто се очекује да се за већину вредности d веома брзо установи да не дају сви елементи исти остатак (тј. неће бити потребе проверавати све елементе у низу), на сложеност доминантно утиче величина броја M.

НЗД разлика бројева

Два броја дају исти остатак при дељењу са d ако и само ако им је апсолутна вредност разлике дељива са d. Дакле, сви бројеви имају исте остатке при дељењу са d ако и само ако d дели све разлике $|a_i - a_j|$ за $0 \leq i < j < n$. Да би број d делио све разлике $|a_i - a_j|$ за $0 \leq i < j < n$, довољно је да дели све разлике $|a_0 - a_i|$, за $0 < i < n$.

Докажимо претходно тврђење. Размотримо произвољну разлику $|a_i - a_j|$. Она је једнака или збиру или разлици бројева $|a_0 - a_i|$ и $|a_0 - a_j|$ (једнака је збиру у случају да је a_0 између a_i и a_j, а разлици у супротном). Међутим, ако d дели оба броја, дели и њихов збир и њихову разлику, па ако d дели све разлике првог броја и осталих, онда дели и разлике свих парова (наравно, ако дели разлике свих парова, онда дели и разлике између првог броја и осталих).

Неки број дели све бројеве неког скупа ако и само ако дели НЗД елемената тог скупа. Зато тражене вредности d можемо да одредимо као све делиоце НЗД-а свих разлика првог елемента од осталих елемената учитаног низа a.

НЗД бројева можемо лако одредити Еуклидовим алгоритмом. Делиоце можемо ефикасно одредити коришћењем чињенице да се делиоци јављају у пару, тј. да сваком делиоцу мањем од корена броја одговара један делилац који је већи од корена броја.

Израчунавање разлика свих бројева и првог броја врши се у сложености $O(n)$. Израчунавање њиховог НЗД у времену $O(n \log M)$ где је M величина највећег броја у низу (јер је највећа апсолутна разлика сигурно мања од $2M$). Делиоци се затим израчунавају у сложености $O(\sqrt{M})$, јер је вредност НЗД сигурно одозго ограничена вредношћу највеће разлике.

```
long long nzd(long long a, long long b) {
  while (b > 0) {
    long long ost = a % b;
    a = b;
    b = ost;
  }
```

```
  return a;
}

long long nzd(const vector<long long>& a) {
  long long NZD = 0;
  for (long long x : a)
    NZD = nzd(NZD, x);
  return NZD;
}

vector<long long> delioci(long long n) {
  vector<long long> Delioci;
  long long i = 2;
  while (i * i < n) {
    if (n % i == 0) {
      Delioci.push_back(i);
      Delioci.push_back(n / i);
    }
    i++;
  }
  if (i * i == n)
    Delioci.push_back(i);
  Delioci.push_back(n);
  sort(begin(Delioci), end(Delioci));
  return Delioci;
}

vector<long long> deliociKojiDajuIsteOstatke(const vector<long long>& a) {
  int n = a.size();
  vector<long long> razlike(n-1);
  for (int i = 1; i < n; i++)
    razlike.push_back(abs(a[i] - a[0]));

  return delioci(nzd(razlike));
}
```

Задатак: Кутије за кликере

Потребно је да n кликера запакујемо у кутије које ћемо да купимо, тако да све кутије, ако је то могуће, буду у потпуности напуњене. Можемо користити две врсте кутија:

- прва кошта c_1 по комаду и у њу стаје n_1 кликера,
- друга кошта c_2 по комаду и у њу стаје n_2 кликера.

Напиши програм који проверава да ли је могуће да се сви кликери запакују у кутије те врсте и ако јесте одређује најмањи износ новца који је потребно потрошити на куповину кутија.

Опис улаза

Са стандардног улаза се учитава број n, затим бројеви c_1, n_1 и c_2, n_2 (сви учитани бројеви су природни бројеви мањи од $2 \cdot 10^9$).

Опис излаза

На стандардни излаз исписати најмању цену кутија које треба купити или -1 ако није могуће спаковати све кликере.

Пример 1

Улаз	*Излаз*	*Објашњење*

`43`
`1 3`
`2 4`
	`15`	У 13 кутија типа 1 које можемо купити за укупно 13 динара можемо спаковати 39 кликера, а у једну кутију типа 2 коју можемо купити за 2 динара можемо спаковати 4 кликера. На тај начин можемо спаковати свих 43 кликера.

Пример 2

Улаз	*Излаз*

`40`
`5 9`
`7 12`
	`-1`

Решење

Нека је d НЗД бројева n_1 и n_2. Ако број n није дељив бројем d, кликере није могуће спаковати. У супротном, проширеним Еуклидовим алгоритмом можемо одредити целе бројеве x_1 и x_2 тако да важи $x_1 n_1 + x_2 n_2 = d$, тј.

$$x_1 \cdot \frac{n}{d} \cdot n_1 + x_2 \cdot \frac{n}{d} \cdot n_2 = n$$

Коефицијенти уз n_1 и n_2 представљају бројеве кутија и морају бити ненегативни, што тренутно не мора бити случај. Сва решења претходне једначине су облика

$$\frac{x_1 n - k n_2}{d} n_1 + \frac{x_2 n + k n_1}{d} n_2 = n$$

Желимо да оба коефицијента буду позитивна, па мора да важи $x_1 n - k n_2 \geq 0$ и $x_2 n + k n_1 \geq 0$, тј.

$$L = \frac{-x_2 \cdot n}{n_1} \leq k \leq \frac{x_1 \cdot n}{n_2} = U$$

За дату вредност k цена коју плаћамо је

$$\begin{aligned} \frac{x_1 n - k n_2}{d} \cdot c_1 \quad &+ \quad \frac{x_2 n + k n_1}{d} c_2 = \\ \frac{x_1 c_1 n + x_2 c_2 n}{d} \quad &+ \quad \frac{n_1 c_2 - n_2 c_1}{d} \cdot k \end{aligned}$$

Видимо да је у питању линеарна функција по k, што значи да је монотона и да се оптимална вредност добија на границама интервала. Најмања могућа вредност k је $\lceil L \rceil$, а највећа $\lfloor U \rfloor$ (ако је $L \leq U$). За ове две вредности k израчунавамо цену и бирамо повољнију.

```
// odredjujemo x1, x2 tako da je x1*n1 + x2*n2 = d
// gde je d nzd brojeva n1 i n2
long long x1, x2;
long long d = prosireni_euklid(n1, n2, x1, x2);
if (n % d != 0)
  cout << -1 << endl;
else {
  // interval [kL, kU] u kom se nalaze vrednosti k tako
  // da oba koeficijenta (x1*n-k*n2) / d i (x2*n+k*n1) / d
  // budu pozitivna
  long long kL = ceil(-x2 * n / n1);
  long long kU = floor(x1 * n / n2);
  if (kL > kU) {
    cout << -1 << endl;
  } else {
    // odredjujemo manju cenu
    long long m1L = (x1*n - kL*n2) / d;
    long long m2L = (x2*n + kL*n1) / d;
    long long cL = m1L*c1 + m2L*c2;
```

```
      long long m1U = (x1*n - kU*n2) / d;
      long long m2U = (x2*n + kU*n1) / d;
      long long cU = m1U*c1 + m2U*c2;
      cout << min(cL, cU) << endl;
    }
  }
```

3.2 Растављање на просте чиниоце (факторизација)

Проблем

Дат је број n. Растставити га на просте чиниоце.

Пример 3.2.1

За $n = 315$, добија се разлагање $315 = 3 \cdot 3 \cdot 5 \cdot 7$.

Као што ћемо видети нешто касније, факторизација бројева игра важну улогу у области криптографије. Међутим, она може бити од користи и за решавање неких стандардних математичких проблема, попут рачунања највећег заједничког делиоца (без Еуклидовог алгоритма) или најмањег заједничког садржаоца два броја, за израчунавање квадратног корена потпуних квадрата итд. У наставку ћемо описати основни алгоритам за факторизацију бројева заснован на покушајима дељења потенцијалним делиоцима (енгл. trial division), који је сложености $O(\sqrt{n})$. Постоје и многи ефикаснији алгоритми (на пример, Полардов ρ, квадратно сито, Фермаова и Ојлерова факторизација итд.), међутим, и са њима проблем факторизације остаје тежак проблем, чије велике инстанце (нпр. бројеве са неколико стотина цифара) ни најсавременији рачунари не могу никако решити.

3.2.1 Факторизација испробавањем делилаца

Проблем факторизације броја n можемо решавати индуктивно-рекурзивним приступом. Формулишемо следећу индуктивну хипотезу.

Индуктивна хипотеза. Све бројеве мање од n умемо да раставимо на просте чиниоце.

Ако некако установимо да је број n прост, онда је једини прост чинилац броја n он сам и проблем је решен (то је база индукције). Ако n није прост број, онда се он може представити као производ $n = d \cdot n_1$, где је d најмањи од свих делилаца броја n (већи од 1).

Лема 3.2.1 **[Најмањи прост чинилац броја]**

Ако је n природан број који није прост, његов најмањи делилац $d > 1$ је прост број и важи $d \leq \sqrt{n}$.

Доказ. Најмањи делилац d броја n мора бити прост број, јер ако би био сложен тј. ако би важило $d = d_1 \cdot d_2$, бројеви d_1 и d_2 (већи од 1) би били делиоци броја n још мањи од d. Додатно, важи $d \leq \sqrt{n}$. Наиме, ако претпоставимо супротно – да је $d > \sqrt{n}$, пошто је $n = d \cdot n_1$, онда би важило $n_1 < \sqrt{n} < d$, те би најмањи делилац броја n_1 био уједно и најмањи делилац броја n, супротно претпоставци да је d најмањи делилац броја n. Дакле, важи $d \leq \sqrt{n}$. □

Дакле, да би се пронашао најмањи прост чинилац броја n, довољно је пронаћи најмањи делилац броја n и он ће обавезно бити прост (то се не мора експлицитно проверавати). Најмањи делилац се може пронаћи проласком кроз скуп природних бројева редом од 2 до $\sqrt{n}$. Ако постоји делилац $d \leq \sqrt{n}$, број n је сложен и проблем се своди на факторизацију броја $n_1 = n/d$, чије просте чиниоце према индуктивној хипотези умемо да одредимо. Ако не постоји делилац $d \leq \sqrt{n}$ броја n, тада је број n прост и он је свој једини прост чинилац.

На основу ове конструкције једноставно је формулисати рекурзивни алгоритам.

Итеративна варијанта овог рекурзивног алгоритма састоји се из наредних корака: пролази се скупом бројева d од 2 до $\sqrt{n}$ и ако је n дељиво бројем d, све док је n дељиво са d памтимо или штампамо вредност d (то су прости чиниоци) и постављамо вредност n на n/d. Након тога, ако је $n > 1$, n је прост и он је прост чинилац полазног броја.

С обзиром на то да ниједан прост број већи од 2 није паран, ефикасније је засебно разматрати број 2, а затим проћи скупом непарних бројева од 3 до $\sqrt{n}$. Слично, могуће је и након бројева 2 и 3 разматрати само бројеве облика $6k \pm -1$, чиме се добија уштеда за фактор 3.

```
// funkcija koja racuna sve proste cinioce broja n
vector<int> prostiCinioci(int n) {
  // vektor u koji upisujemo rezultat
  vector<int> cinioci;
  // ako je n parno, uzimamo cinilac 2 onoliki broj puta koliko puta deli n
  while (n % 2 == 0) {
    cinioci.push_back(2);
    // azuriramo n
    n /= 2;
```

```
  }
  // n je neparno
  // prolazimo kroz neparne brojeve
  for (int d = 3; d*d <= n; d += 2) {
    // sve dok d deli n, uzimamo cinilac d
    while (n % d == 0) {
      cinioci.push_back(d);
      // azuriramo n
      n /= d;
    }
  }
  // ako je n veci od 1, on je prost
  if (n > 1)
    cinioci.push_back(n);

  return cinioci;
}
```

Инваријанта спољашње `for` петље алгоритма чија је имплементација у C++ приказана је то да је у сваком кораку производ до тада одређених простих чинилаца и тренутне вредности промењиве n једнак полазном броју n, као и да тренутна вредност n није дељива ни са једним бројем који је мањи или једнак d. Када се спољашња петља заврши, пошто број n није дељив ни са једним бројем мањим или једнаким од свог корена, n мора бити прост, па ако је већи од 1, он је највећи прост чинилац полазног броја.

Истакнимо једну важну ствар: нигде у програму се посебно не испитује да ли су пронађени делиоци прости бројеви. На први поглед ово може деловати збуњујуће, међутим редослед којим тражимо делиоце броја n нам гарантује да ће сваки одређени делилац d бити прост. Наиме, на основу инваријанте знамо да када се вредност d дода у колекцију простих чинилаца броја n, тренутна вредност промењиве n није дељива ниједним бројем мањим од d, а ако би d био сложен, тј. ако би важило $d = d_1 \cdot d_2$, n би био дељив са d_1 и d_2 који су већи од 1, а мањи од d, што је на основу инваријанте немогуће.

Пошто се петља `for` извршава до корена из n (које је све време мање или једнако полазној вредности броја $n_0 \equiv n$), можемо закључити да је сложеност алгоритма $O(\sqrt{n_0})$. Приметимо да се за сложене бројеве граница $\sqrt{n}$ петље `for` смањује са сваким новим пронађеним простим чиниоцем (јер ће нова вредност за n бити строго мања од старе), па у неким случајевима алгоритам може извршити и доста мање корака од $\sqrt{n_0}$. Ако су прости чиниоци (не нужно различити) броја n_0 редом једнаки $p_1 \leq p_2 \leq \ldots \leq p_{k-1} \leq p_k$, може се показати да је сложеност овог алгоритма за факторизацију $O(\max(p_{k-1}, \sqrt{p_k}))$, јер се дели до p_{k-1}, а затим евентуално до $\sqrt{p_k}$. Приметимо да је

у питању излазно-зависна сложеност, јер сложеност зависи од резултата. При том претпостављамо да радимо са машинским типовима података, па се унутрашња петља којом се врши дељење извршава увек мали број пута (збир свих степена у факторизацији је ограничен бројем битова у запису броја).

Пример 3.2.2

Илуструјмо сложеност на неколико различитих примера:

- *ако је* $n = 71 \cdot 73$*, први (а уједно и претпоследњи) прост чинилац броја* n *је једнак* $p_{k-1} = 71$ *и њега бисмо одредили када променљива* i `for` *петље стигне до* 71*, након чега вредност променљиве* n *постаје* 73*, а пошто граница за* i *иде до* $\lfloor \sqrt{n} \rfloor = \lfloor \sqrt{73} \rfloor = 8$*, овде се прекида* `for` *петља. Дакле, када су бројеви* p_{k-1} *и* p_k *блиски по вредности, број итерација петље биће једнак* $O(p_{k-1})$*.*

- *ако је* $n = 2 \cdot 73$*, први (а уједно и претпоследњи) прост чинилац броја* n *је једнак* $p_{k-1} = 2$ *и њега бисмо одредили у старту – када* i *има вредност* 2*, након чега вредност променљиве* n *постаје* 73*, а пошто граница за* i *иде до* $\lfloor \sqrt{n} \rfloor = \lfloor \sqrt{73} \rfloor = 8$*,* `for` *петља би се извршавала* $O(\sqrt{p_k})$ *пута. Дакле, када је* p_{k-1} *доста мање од* p_k*, број итерација петље биће једнак* $O(\sqrt{p_k})$*.*

- *ако је* $n = 73$*, број је прост и* `for` *петља се извршава* $O(\sqrt{n})$ *пута, а једини прост чинилац броја* n *би био пронађен накнадно, након изласка из* `for` *петље.*

Дакле, важи следеће: није једнако тешко факторисати све бројеве исте дужине. Најтеже инстанце овог проблема, ако се примењује описани алгоритам, чине бројеви који су прости или су производи два приближно једнака проста броја. Наиме, када су оба ова проста чиниоца велика и сличне величине, овај проблем постаје јако тешко решити. Као илустрацију овога, наведимо да се применом веома ефикасног алгоритма факторизације, 2009. године окончао двогодишњи напор групе истраживача да се факторише број од 232 цифре коришћењем више стотина рачунара. Претпостављена тежина овог проблема је основа за процену сигурности великог броја криптографских алгоритама, као што је алгоритам RSA (видети поглавље 3.5).

Нагласимо и да је то што се петља зауставља када је $d > \sqrt{n}$ и што се случај простог броја n обрађује ван петље довело до алгоритма сложености $O(\sqrt{n})$. Ако би се делиоци редом проверавали све док се број дељењем не сведе на вредност 1, уместо да се стане када се пређе $\sqrt{n}$ (како се често ради када се ручно одређују прости чиниоци), добио би се алгоритам сложености $O(n)$. Дакле, то што су за базу индукције узети сви прости бројеви а не број 1, довело је до значајно ефикаснијег алгоритма.

3.2.2 Фермаов алгоритам факторизације

Када број n има делиоце близу вредности $\sqrt{n}$, они се ефикасније могу наћи Фермаовим[5] алгоритмом. Он је заснован на томе да се сваки непаран природан број n може написати као разлика два потпуна квадрата $n = a^2 - b^2$. Заиста, број n се увек може написати као производ $n = n_1 \cdot n_2$ (ако је n прост, тада је $n_1 = n$ и $n_2 = 1$, а ако је n сложен, n_1 и n_2 могу бити његови произвољни делиоци). При том су n_1 и n_2 непарни, јер је n непаран. Тада важи

$$n = \left(\frac{n_1 + n_2}{2}\right)^2 - \left(\frac{n_1 - n_2}{2}\right)^2,$$

при чему су квадрати целобројни, јер су n_1 и n_2 непарни, па су њихов збир и разлика парни.

Ако је $n = a^2 - b^2$, тада се n може раставити на чиниоце $(a-b)\cdot(a+b)$. Ако је $a-b$ различито од 1, тада смо пронашли два чиниоца, који се даље могу факторисати.

Алгоритам се заснива на испробавању разних вредности a, кренувши од $\lceil\sqrt{n}\rceil$, навише и испитивању да ли је $a^2 - n$ квадрат неког целог броја b. То се у неком тренутку мора десити. Заиста, ако је $n = n_1 \cdot n_2$, у неком тренутку ће a достићи вредност $(n_1 + n_2)/2$. Ако је n прост, то ће се десити тек када се достигне $(n+1)/2$, што је прилично неефикасно. Са друге стране, ако постоји делилац броја n близак вредности $\sqrt{n}$, он ће се брзо пронаћи, јер претрага креће од вредности $a = \lceil\sqrt{n}\rceil$, а ако постоје вредности n_1 и n_2 блиске броју $\sqrt{n}$, и њихова аритметичка средина ће му бити блиска и брзо ће бити пронађена. Приликом провере да ли је b^2 потпун квадрат пуно бројева може бити одбачено на основу једноставних тестова (на пример, последња цифра тј. остатак при дељењу са 10 потпуног квадрата мора бити 0, 1, 4, 5, 6, или 9, а слични критеријуми се могу формулисати и за друге бројевне основе).

Пример 3.2.3

Факторишимо број 5959. *Важи* $\lceil\sqrt{5959}\rceil = 78$, *па алгоритам креће од* $a = 78$.

Корак	1	2	3
a	78	79	80
$b^2 = a^2 - n$	125	282	441
b	-	-	21

[5] Пјер де Ферма (фр. Pierre de Fermat), (1607-1665), француски математичар.

Дакле, у само три корака се проналази да је $5959 = 80^2 - 21^2$*, што даје чиниоце* $a-b = 89-21 = 59$ *и* $a+b = 80+21 = 101$*. Они су доста мањи од полазног броја и њихова факторизација тече брже. Приметимо да би алтернативни приступ у ком се разматрају вредности* b *од* 1 *до* $\sqrt{n}$ *и проверава се да ли је број* $b^2 + n$ *потпун квадрат захтевао много више корака.*

3.2.3 Факторизација више бројева помоћу Ератостеновог сита

Некад је потребно велики број пута извршити факторизацију броја и то се може урадити ефикасно коришћењем идеје Ератостеновог сита. Зато наредни поступак називамо *факторизација помоћу Ератостенове сита*.

Проблем

За дати број n *пронаћи факторизацију већег броја бројева из интервала* $[1, n]$ *(на пример извршити факторизацију свих бројева који су мањи или једнаки* n*).*

Могуће је $O(n)$ пута извршити испочетка основни алгоритам факторизације и укупна сложеност овог приступа износи $O(n\sqrt{n})$.

Кључни корак размотреног алгоритма за факторизацију је проналажење најмањег (простог) чиниоца текућег броја. Када факторишемо више бројева доћи ћемо у ситуацију да више пута за исти број одређујемо најмањи прости чинилац. На пример, приликом факторисања броја 221 одредићемо да је његов најмањи прост чинилац 13, а приликом факторисања броја 442 одредићемо да је његов најмањи прост чинилац 2, међутим, тада ћемо број 442 поделити са 2 и поново доћи у ситуацију да треба да одређујемо најмањи прост чинилац броја 221, што смо већ урадили. Дакле, до ефикаснијег решења долази се ако применимо технику динамичког програмирања тј. мемоизације и конструишемо помоћни низ дужине n који за свако $k \leq n$ садржи најмањи прост чинилац броја k. Наиме, кад бисмо знали вредност најмањег простог чиниоца сваког броја k за $k \leq n$, број $m \leq n$ бисмо могли факторисати на следећи начин: рачунамо вредност p_1 најмањег простог чиниоца броја m, затим вредност p_2 најмањег простог чиниоца броја m/p_1, затим вредност најмањег простог чиниоца броја $m/(p_1 \cdot p_2)$, ... Дакле, проблем факторизације бројева од 1 до n можемо свести на проблем ефикасног израчунавања низа најмањих простих чинилаца свих бројева од 1 до n.

Низ који за сваки природан број мањи или једнак n садржи вредност његовог најмањег простог чиниоца, може се добити малим прилагођавањем Ератостеновог сита. Подсетимо се, Ератостеново сито се користи да се одреде сви прости бројеви до датог броја n, тако што се прецртају сви умношци броја 2, затим броја 3, затим броја 5 и тако редом, све док се не прецртају и сви умношци броја $\lfloor\sqrt{n}\rfloor$. Непрецртани бројеви су сви тражени прости бројеви.

Овај се алгоритам лако прилагођава тако да одређује најмањи прост чинилац сваког броја до n. Иницијално за сваки број i постављамо да је његов најмањи прост чинилац баш i. У оригиналном Ератостеновом ситу се приликом проласка кроз умношке неког простог броја d означава да су ти умношци сложени (за шта се користи низ логичких вредности). Уместо тога, у овом алгоритму ћемо приликом проласке кроз умношке неког простог броја d, свим умношцима којима није раније умањена вредност најмањег простог чиниоца, поставити ту вредност на d. За број d закључујемо да је прост, ако је његов најмањи прост чинилац он сам. Приликом разматрања умножака простог броја d, можемо да кренемо од d^2, јер су бројеви $2d, 3d, \ldots, (d-1)d$ дељиви редом са $2, 3, \ldots, d-1$, те сигурно имају прост чинилац мањи од d.

Када се прва фаза заврши и попуни низ најмањих простих чинилаца, тада се у другој фази попуњени низ може употребити за брзу факторизацију било ког броја од 1 до n.

Пример 3.2.4

Нека је потребно факторисати све бројеве који су мањи или једнаки 50.

- *Крећемо од низа у коме је на позицији* i *уписана вредност* i.

```
 1  2  3  4  5  6  7  8  9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
```

- *У првом кораку свим парним бројевима постављамо да је најмањи прост чинилац једнак* 2*:*

```
 1  2  3  2  5  2  7  2  9  2
11  2 13  2 15  2 17  2 19  2
21  2 23  2 25  2 27  2 29  2
31  2 33  2 35  2 37  2 39  2
41  2 43  2 45  2 47  2 49  2
```

- *Умношцима броја* 3 *почев од* 9 *који нису дељиви са* 2 *постављамо најмањи прост чинилац на* 3*: то су* 9, 15, 21, 27, 33, 39 *и* 45.

```
 1  2  3  2  5  2  7  2  3  2
11  2 13  2  3  2 17  2 19  2
 3  2 23  2 25  2  3  2 29  2
31  2  3  2 35  2 37  2  3  2
41  2 43  2  3  2 47  2 49  2
```

- *Разматрамо број* 4. *Закључујемо да је он сложен јер је његов најмањи прост чинилац постављен на вредност мању од* 4, *те број* 4 *прескачемо.*
- *Умношцима броја* 5 *почев од* 25, *који нису дељиви неким мањим простим бројем (*2 *и* 3*), постављамо да је најмањи прост чинилац једнак* 5. *То ће бити бројеви* 25 *и* 35.

```
 1  2  3  2  5  2  7  2  3  2
11  2 13  2  3  2 17  2 19  2
 3  2 23  2  5  2  3  2 29  2
31  2  3  2  5  2 37  2  3  2
41  2 43  2  3  2 47  2 49  2
```

- *Број* 6 *као сложен прескачемо.*
- *Умношцима броја* 7 *почев од* 49, *који нису дељиви са* 2, 3 *нити* 5, *постављамо да је* 7 *њихов најмањи прост чинилац (то је само број* 49*).*

```
 1  2  3  2  5  2  7  2  3  2
11  2 13  2  3  2 17  2 19  2
 3  2 23  2  5  2  3  2 29  2
31  2  3  2  5  2 37  2  3  2
41  2 43  2  3  2 47  2  7  2
```

- *Тиме се обрада завршава јер је* $8^2 \geq 50$.

На пример, приликом факторизације броја 48 *исписују се редом вредности* $2, 2, 2, 2$ *и* 3 *као вредности најмањих простих чинилаца редом бројева* 48, $48/2 = 24$, $24/2 = 12$, $12/2 = 6$ *и* $6/2 = 3$.

У наставку је дата имплементација алгоритма факторизације свих бројева од 1 до n коришћењем Ератостеновог сита.

```
vector<int> eratosten(int n) {
  // niz koji cemo popuniti tako da se na poziciji i
  // nalazi najmanji prost cinilac broja i
  vector<int> najmanjiCinilac(n + 1);

  // postavljamo da je najmanji prost cinilac svakog broja sam taj broj
  for (int i = 1; i <= n; i++)
    najmanjiCinilac[i] = i;
  for (int d = 2; d * d <= n; d++)
```

```
    // ako je broj d prost, tj njegov najmanji prost činilac je on sam
    if (najmanjiCinilac[d] == d)
      // prolazimo kroz skup svih umnozaka tog broja
      for (int i = d * d; i <= n; i += d)
        // ako nije ranije azurirana vrednost, odnosno
        // postavljena neka manja vrednost za cinilac
        if (najmanjiCinilac[i] == i)
          // postavljamo da je najmanji prost cinilac broj d
          najmanjiCinilac[i] = d;
  return najmanjiCinilac;
}

// funkcija koja ispisuje sve proste cinioce broja n
void ispisiProsteCinioce(int n, const vector<int>& najmanjiCinilac) {
  while (n != 1) {
    // stampamo najmanji prost cinilac broja n
    cout << najmanjiCinilac[n] << " ";
    // azuriramo vrednost n
    n /= najmanjiCinilac[n];
  }
  cout << endl;
}
```

Сложеност првог корака алгоритма – одређивања најмањег простог чиниоца свих бројева до n једнака је сложености алгоритма Ератостеново сито која је, ако претпоставимо да је сложеност сабирања $O(1)$, једнака суми n/p по простим бројевима p, што је $O(n \log \log n)$.[6] Сложеност исписивања простих чинилаца броја k износи $O(\log k)$ јер је максимални број простих чинилаца броја k једнак $\log_2 k$, а претпостављамо да је дељење константне временске сложености.[7] Пошто треба исписати просте чиниоце свих бројева од 1 до n, укупно време потребно за то једнако је $\log 1 + \log 2 + \ldots + \log n$ што је, присетимо се, једнако $\Theta(n \log n)$. Дакле, сложеност приказаног алгоритма за факторизацију бројева од 1 до n је једнака $O(n \log \log n) + O(n \log n) = O(n \log n)$, што је значајно ефикасније од приступа заснованог на независној факторизацији сваког броја посебно.

[6] Напоменимо да ако би n могло бити произвољно велико (ако би број био представљен репрезентацијом са произвољним бројем битова), онда би сложеност сабирања била $O(\log n)$, међутим у конкретним имплементацијама ослањамо се на сабирање бројева ограниченог опсега које је хардверски подржано и за које сматрамо да је сложености $O(1)$.

[7] Ако би број n могао бити произвољно велики, онда би сложеност дељења била $O(\log^2 n)$.

Задатак: Допуна до пуног квадрата

Напиши програм који за унети природни број n одређује најмањи број m такав да је $n \cdot m$ потпун квадрат.

Опис улаза

Са стандардног улаза се уноси природни број n ($1 \leq n \leq 2 \cdot 10^{12}$).

Опис излаза

На стандардни излаз исписати тражени број m.

Пример

Улаз	*Излаз*
`104`	`26`

Решење

Најмањи број којим се n множи да би се добио потпун квадрат мора бити делилац броја n (он мора да буде производ простих фактора броја n који се јављају са непарном вишеструкошћу). Зато можемо испитати све делиоце броја n (линеарном претрагом) и проверити који је први који када се помножи са n даје потпун квадрат. Проверу да ли је број потпун квадрат можемо вршити уз помоћ функције (реалног) кореновања (додуше, то може некада довести до грешака услед непрецизности у запису реалних бројева).

Пошто се делиоци јављају увек у пару (ако је број d делилац броја n, онда је делилац и број n/d), претрагу можемо вршити само до вредности $\sqrt{n}$.

Ако је број $d \leq \sqrt{n}$ први који задовољава услов, он је најмањи такав и претрага се може прекинути. Са друге стране, када број n/d задовољи услов, само то региструјемо и настављамо претрагу, јер не знамо да је он најмањи такав.

Осим што може бити донекле неефикасна, основни проблем са оваквом имплементацијом је то што број n и његови делиоци могу бити 64-битни бројеви, па се њихов производ не може представити исправно 64-битним бројем, тако да овај приступ може да да гарантовано исправне резултате само ако је $n \leq 10^9$.

Сложеност овог приступа је $O(\sqrt{n})$.

Пример 3.2.5

Размотримо број 234. *Он се може растaвити на просте чиниоце као* $2 \cdot 3 \cdot 3 \cdot 13$. *Да би овај број био потпун квадрат потребно је да се сваки чинилац понови паран број пута и зато је тражена допуна* $2 \cdot 13$. *Заиста, множењем са* $2 \cdot 13 = 26$ *добија се број* $2^2 \cdot 3^2 \cdot 13^2$ *који је квадрат броја* $2 \cdot 3 \cdot 13 = 78$.

Када се пун квадрат растави на просте чиниоце, сваки чинилац се јавља паран број пута. Ако је број $n = p_1^{k_1} \cdot \ldots \cdot p_m^{k_m}$, тада је тражена допуна једнака производу оних чинилаца p_i за који је вишеструкост k_i непаран број. Дакле, задатак се једноставно решава применом алгоритма растављања на просте чиниоце. Издвајамо чинилац по чинилац броја и за сваки од њих бројимо вишеструкост (тако што сваки пут када поделимо број n са неким простим чиниоцем p увећамо вредност бројача k). Када утврдимо да неки чинилац има непарну вишеструкост, тада њиме множимо тренутну вредност броја m (коју иницијализујемо на 1).

Приметимо да нема потребе да израчунавамо производ бројева n и m, па је исправно све време користити 64-битни тип података (који је довољан за запис бројева n и m, али не и њиховог производа).

Сложеност овог алгоритма одговара сложености растављања на просте чиниоце, а то је $O(\sqrt{n})$.

```
long long dopunaDoPunogKvadrata(long long n) {
  // dopuna
  long long m = 1;

  // rastavljamo broj n na proste cinioce

  // kandidat za prost cinilac
  long long p = 2;
  while (p * p <= n) {
    // racunamo visestrukost cinioca p
    int k = 0;
    while (n % p == 0) {
      n /= p;
      k++;
    }
    // ako se cinilac p javlja neparan broj puta tada se on mora
    // javiti u dopuni m
    if (k % 2 != 0)
      m *= p;
    // prelazimo na sledeceg kandidata
    p++;
  }
  // n se sveo na svoj najveci prost cinilac
  if (n > 1)
```

```
    // i on mora da ucestvuje u dopuni
    m *= n;

  return m;
}
```

Задатак: Највећи НЗД бројева чији је производ познат

Напиши програм који на основу познатог производа два позитивна природна броја a и b одређује највећу могућу вредност њиховог највећег заједничког делиоца.

Опис улаза

Са стандардног улаза се учитава број $p = a \cdot b$ ($1 \leq p \leq 10^{19}$).

Опис излаза

На стандардни излаз исписати максималну могућу вредност за НЗД.

Пример

Улаз	*Излаз*	*Објашњење*
`600`	`10`	Највећи НЗД се добија када се број 600 представи као производ бројева 20 и 30.

Решење

Рачунање НЗД свих производа

Решење грубом силом подразумева да се број p на све могуће начине разложи на производ $p = a \cdot b$ и да се израчуна НЗД за све тако добијене парове. Без губитка на општости се може претпоставити да је $a \leq b$, па се зато проналазе сви делиоци $a \leq \sqrt{p}$. НЗД два броја се може израчунати Еуклидовим алгоритмом.

Проналажење делилаца вршимо у сложености $O(\sqrt{p})$, а рачунање њиховог НЗД-а у сложености $O(\log p)$. Стога укупну служеност можемо оценити са $O(\sqrt{p} \log p)$.

Растављање на просте чиниоце

Раставимо p на просте чиниоце, тј. претпоставимо да је $p = p_1^{\alpha_1} \dots p_k^{\alpha_k}$.

Два чиниоца a и b треба одредити тако да им је НЗД што већи, тј. да имају што више заједничких простих фактора. Стога је најбоље да први број буде једнак

$$a = p_1^{\lfloor \frac{\alpha_1}{2} \rfloor} \dots p_k^{\lfloor \frac{\alpha_k}{2} \rfloor}$$

а други број буде једнак $b = \frac{p}{a}$. Број b је дељив бројем a и НЗД та два броја једнак је a. То је уједно највећи НЗД који се може добити.

Сложеност потиче од сложености факторизације и износи $O(\sqrt{p})$. Пошто се број дели својим факторима, алгоритам се често извршава само до корена највећег простог фактора броја p, што може значајно утицати на време извршавања у таквим примерима.

3.3 Мултипликативне функције

Једна од важних примена факторизације броја је ефикасно израчунавање неких функција над природним бројевима. За функцију $f : \mathbb{N} \to \mathbb{N}_0$ кажемо да је *мултипликативна* (енгл. multiplicative function) ако и само ако за свака два узајамно проста броја a и b (два броја за које је nzd$(a, b) = 1$) важи да је $f(a \cdot b) = f(a) \cdot f(b)$. Када се број n факторише у производ $p_1^{k_1} p_2^{k_2} \dots p_m^{k_m}$, чиниоци $p_1^{k_1}, p_2^{k_2}, \dots, p_m^{k_m}$ су узајамно прости па, ако је f мултипликативна функција, онда важи:

$$f(n) = f(p_1^{k_1}) \cdot f(p_2^{k_2}) \cdot \dots \cdot f(p_m^{k_m}).$$

Дакле, да би се могла одредити вредност мултипликативне функције за произвољни број n потребно је одредити вредност функције за произвољни степен простог броја тј. за бројеве обилка p^k. То је често много једноставније него у општем случају и може се решити аналитички, проналажењем израза којим се та вредност израчунава. Нагласимо да је у општем случају $f(p^k) \neq f(p)^k$, јер степени простог броја p нису међусобно узајамно прости (имају заједнички делилац p).

Ако је временска сложеност израчунавања вредности $f(p^k)$ једнака $O(1)$, што је често случај, онда временска сложеност израчунавања вредности $f(n)$ одговара времену потребном за факторизацију броја n што је $O(\sqrt{n})$.

Ако је потребно израчунати вредности неких мултипликативних функција за више улазних параметара (на пример за све вредности од 1 до n), тада је могуће употребити факторизацију помоћу Ератостеновог сита, која је ефикаснија од факторизације сваког броја појединачно.

У наставку ће бити приказано израчунавање неколико важних мултипликативних функција.

3.3.1 Ојлерова функција

Размотримо наредни проблем: потребно је одредити број несводљивих разломака у интервалу $[0, 1]$ чији је именилац једнак датом броју n. То су сви они разломци $\frac{m}{n}$, $m \leq n$ за које важи да су m и n узајамно прости. Задатак се дакле своди на пребројавање колико има природних бројева мањих или једнаких n који су узајамно прости са n.

Ојлерова функција (енгл. Euler's totient function) φ броја n означава број природних бројева мањих или једнаких од n који су узајамно прости са n.

Пример 3.3.1

Важи $\varphi(9) = 6$*, јер постоји шест бројева* $1, 2, 4, 5, 7, 8$ *мањих од* 9 *који су узајамно прости са* 9*, а* $\varphi(17) = 16$*, јер је број* 17 *прост и узајамно је прост са свим бројевима строго мањим од њега.*

По дефиницији је $\varphi(1) = 1$[8]. Означимо са Φ_n скуп свих бројева који су мањи од n и узајамно су прости са n. На пример, $\Phi_9 = \{1, 2, 4, 5, 7, 8\}$. Важи да је $\varphi(n) = |\Phi_n|$.

Ојлерова функција има примену у теорији бројева, у алгебри, али, као што ћемо видети, игра и кључну улогу у систему за шифровање RSA.

Проблем

За дати природан број n *израчунати вредност Ојлерове функције* $\varphi(n)$.

3.3.1.1 Директан алгоритам

Директан начин да се реши проблем био би да се редом прође кроз све бројеве од 1 до $n - 1$ и да се изброји колико је од њих узајамно просто са n.

```
// funkcija koja racuna vrednost Ojlerove funkcije
int ojlerovaFunkcija(int n) {
  // 1 je uvek uzajamno prosto sa n
  int phi = 1;
  for (int i = 2; i < n; i++)
    if (nzd(i, n) == 1)
      phi++;
  return phi;
}
```

[8] Некад се Ојлерова функција броја n дефинише као број природних бројева строго мањих од n, што једино утиче на вредност Ојлерове функције броја 1.

Приликом рачунања вредности Ојлерове функције, функција за одређивање највећег заједничког делиоца два броја се позива $O(n)$ пута. Како знамо да је сложеност алгоритма за израчунавање највећег заједничког делиоца бројева i и n једнака $O(\log(i+n))$ и како је овде увек $i < n$, важи да је сложеност рачунања вредности $\text{nzd}(i, n)$ једнака $O(\log n)$, те је укупна сложеност приказаног алгоритма за израчунавање вредности Ојлерове функције броја n једнака $O(n \log n)$.

3.3.1.2 Рачунање Ојлерове функције броја свођењем на факторизацију

Доказаћемо да је Ојлерова функција мултипликативна, што омогућава њено израчунавање свођењем на факторизацију.

Лема 3.3.1 **[Мултипликативност Ојлерове функције φ]**

Ојлерова функција φ је мултипликативна, тј. ако су M и N узајамно прости бројеви, тада је $\varphi(M \cdot N) = \varphi(M) \cdot \varphi(N)$.

Доказ. Постоји бијекција између скупова $\Phi_M \times \Phi_N$ и Φ_{MN}. Бијекција се успоставља тако што се сваком пару $(m, n) \in \Phi_M \times \Phi_N$ додели јединствен број $k \leq M \cdot N$ који при дељењу са M даје остатак m (тј. важи $k \bmod M = m$), а при дељењу са N даје остатак n (тј. важи $k \bmod N = n$). Тај број постоји и јединствен је на основу тзв. Кинеске теореме о остацима (видети поглавље 3.4.5). Он је узајамно прост са $M \cdot N$, па припада скупу Φ_{MN}. Заиста, на основу Еуклидовог алгоритма важи $\text{nzd}(k, M) = \text{nzd}(M, k \bmod M) = \text{nzd}(M, m) = 1$ и $\text{nzd}(k, N) = \text{nzd}(N, k \bmod N) = \text{nzd}(N, n) = 1$. Пошто је $\text{nzd}(k, M) = 1$ и $\text{nzd}(k, N) = 1$, мора да важи и $\text{nzd}(k, MN) = 1$.

Обратно, сваком броју $k \in \Phi_{MN}$ се може придружити пар $(k \bmod M, k \bmod N) \in \Phi_M \times \Phi_N$. Заиста, остаци су увек мањи од делиоца, а $k \bmod M$ и $k \bmod N$ морају бити узајамно прости са M односно N (што се доказује аналогно претходном смеру). Дакле, пошто је између два коначна скупа успостављена бијекција, они су истобројни:

$$\varphi(MN) = |\Phi_{MN}| = |\Phi_M \times \Phi_N| = |\Phi_M| \cdot |\Phi_N| = \varphi(M) \cdot \varphi(N). \qquad \square$$

Пример 3.3.2

Нека је $M = 5$ и $N = 6$. Узајамно прости са $M = 5$ су елементи скупа $\Phi_5 = \{1, 2, 3, 4\}$, а са N елементи скупа $\Phi_6 = \{1, 5\}$. Узајамно прости са $MN = 30$ су елементи скупа $\Phi_{30} = \{1, 7, 11, 13, 17, 19, 23, 29\}$. Остаци при дељењу елемената скупа Φ_{30} бројевима $M = 5$ и $N = 6$ су редом следећи парови: $\{(1, 1), (2, 1), (1, 5), (3, 1), (2, 5), (4, 1), (3, 5), (4, 5)\}$ и јасно се види да су то тачно сви парови скупа $\Phi_5 \times \Phi_6$.

Пошто је Ојлерова функција мултипликативна, довољно је одредити само њене вредности за степене простих бројева тј. за бројеве облика p^k.

Лема 3.3.2 **[Рачунање Ојлерове функције $\varphi(p^k)$]**

Нека је p прост број. Тада важи:

$$\varphi(p^k) = p^k - p^{k-1} = p^k \left(1 - \frac{1}{p}\right)$$

Доказ. Од свих бројева мањих или једнаких од p^k (којих има p^k) са бројем p^k нису узајамно прости само умношци броја p, односно бројеви $p, 2p, 3p, \ldots, p^{k-1}p$, којих има укупно p^{k-1}. □

Лема 3.3.3 **[Рачунање Ојлерове функције $\varphi(n)$]**

$$\varphi(n) = n \cdot \prod_{\substack{p|n \\ p \text{ prost}}} \left(1 - \frac{1}{p}\right) \tag{3.4}$$

На основу овога, могуће је израчунати вредност Ојлерове функције за произвољно n.

Доказ. У општем случају је $n = p_1^{k_1} \cdot \ldots \cdot p_m^{k_m}$, где су $p_1, p_2, \ldots, p_m$ различити прости чиниоци броја n, па је

$$\begin{aligned}
\varphi(n) &= \varphi(p_1^{k_1})\varphi(p_2^{k_2}) \cdot \ldots \cdot \varphi(p_m^{k_m}) \\
&= p_1^{k_1}\left(1 - \frac{1}{p_1}\right) p_2^{k_2}\left(1 - \frac{1}{p_2}\right) \cdot \ldots \cdot p_m^{k_m}\left(1 - \frac{1}{p_m}\right) \\
&= p_1^{k_1} p_2^{k_2} \cdot \ldots \cdot p_m^{k_m}\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right) \cdot \ldots \cdot \left(1 - \frac{1}{p_m}\right) \\
&= n \cdot \prod_{\substack{p|n \\ p \text{ prost}}} \left(1 - \frac{1}{p}\right)
\end{aligned}$$

□

Пример 3.3.3

- $\varphi(30) = \varphi(2^1 \cdot 3^1 \cdot 5^1) = 30(1 - \frac{1}{2})(1 - \frac{1}{3})(1 - \frac{1}{5}) = 30 \cdot \frac{1}{2} \cdot \frac{2}{3} \cdot \frac{4}{5} = 8$
- $\varphi(36) = \varphi(2^2 \cdot 3^2) = 36(1 - \frac{1}{2})(1 - \frac{1}{3}) = 36 \cdot \frac{1}{2} \cdot \frac{2}{3} = 12$

Често је у применама (на пример у алгоритму RSA) потребно израчунати $\varphi(n)$, где је n производ два велика проста броја p и q; тада је $\varphi(n) = \varphi(p \cdot q) = \varphi(p) \cdot \varphi(q) = (p-1) \cdot (q-1)$.

Имплементација рачунања Ојлерове функције броја своди се на прилагођавање алгоритма факторизације. Вредност функције φ чувамо у променљивој `phi`. Њено множење фактором `(1 - 1/p)` у имплементацији не би било коректно, јер тај фактор није целобројан. Ако `phi` иницијализујемо на `n`, можемо га ажурирати наредбама облика `phi = (phi / p) * (p - 1)`, јер је број `phi` увек дељив бројем `p` (`phi` се иницијализује на вредност `n`, а `p` је увек делилац броја `n`). Пошто је `(phi / p) * (p - 1) = phi - phi / p`, можемо избећи множење тј. можемо ажурирати `phi` наредбама облика `phi -= phi / p`.

```
// funkcija koja racuna vrednost Ojlerove funkcije
// svodjenjem na problem faktorizacije
int ojlerovaFunkcija(int n) {
  // rezultat inicijalizujemo na n
  int phi = n;
  // za svaki prost cinilac p broja n
  // rezultat mnozimo sa 1-1/p = (p-1)/p
  // usput azuriramo vrednost broja n
  for (int p = 2; p * p <= n; p++) {
    if (n % p == 0) {
      while (n % p == 0)
        n /= p;
      phi -= phi / p;
    }
  }
  // ako je n prost broj
  if (n > 1)
    phi -= phi / n;
  return fi;
}
```

Сложеност овог алгоритма одговара сложености одговарајућег алгоритма за факторизацију, односно једнака је $O(\sqrt{n})$. Иако је овај алгоритам неупоредиво ефикаснији него

директан алгоритам, израчунавање вредности Ојлерове функције када није позната факторизација броја n се сматра рачунски изузетно тешким проблемом и није га могуће извршити за велике бројеве n (са неколико стотина цифара). На овоме се заснива сигурност неких криптографских алгоритама (на пример, RSA описаног у поглављу 3.5).

Као што је то случај када се ради факторизација, уместо да се у петљи пролази редом кроз све природне бројеве од 2 до $\sqrt{n}$, посебно се може размотрити вредност $p = 2$, а затим се у петљи може пролазити само кроз скуп непарних бројева од 3 до $\sqrt{n}$, чиме би се добило убрзање за фактор 2.

3.3.1.3 Рачунање Ојлерове функције свих бројева до n

Уколико би задатак био да се израчуна вредност Ојлерове функције за све бројеве мање или једнаке n, претходна имплементација била би сложености $O(n\sqrt{n})$.

Као и раније, размотримо варијанту алгоритма засновану на Ератостеновом ситу, којом се ангажује додатни меморијски простор величине $O(n)$. На основу једнакости (3.4) можемо закључити да се за све бројеве дељиве неким простим бројем p у изразу за вредност Ојлерове функције јавља чинилац $1 - \frac{1}{p}$. Дакле, можемо редом пролазити кроз све просте бројеве и вредности Ојлерове функције свих умножака текућег простог броја p помножити изразом $1 - \frac{1}{p}$.

Пошто у изразу (3.4) за $\varphi(n)$ као чинилац фигурише n, вредности елемената низа $\bar{\varphi}(i)$, $1 \leq i \leq n$, иницијализоваћемо на вредност i. Разматраћемо редом све вредности за p од 2 до n: наиме, за разлику од основне варијанте Ератостеновог сита где је вредност за p ишла до $\sqrt{n}$, овде је неопходно да p прође скупом свих вредности до n, јер је потребно обрадити вредности свих простих чинилаца (а не само најмањих) свих бројева. Ако приликом обраде броја p и даље важи $\bar{\varphi}(p) = p$, број p је прост, па вредност $\bar{\varphi}$ свих умножака броја p (укључујући и њега) множимо са $\frac{p-1}{p}$. Притом не можемо као у оригиналној верзији Ератостеновог сита кренути од вредности p^2, јер морамо све умношке броја p (и бројеве $p, 2p, 3p, \ldots, (p-1)p$) помножити са $\frac{p-1}{p}$. Ако приликом обраде броја p важи $\bar{\varphi}(p) \neq p$, број p је сложен и прескаче се. На крају алгоритма у низу $\bar{\varphi}(i)$ налазе се вредности Ојлерове функције за све вредности i од 1 до n.

```
vector<int> OjlerovaFunkcijaDoN(int n) {
  // niz u kome na poziciji i formiramo vrednost phi(i)
  vector<int> phi(n + 1);
  // inicijalizujemo sve vrednosti
  for (int i = 1; i <= n; i++)
    phi[i] = i;
```

```
  // za sve vrednosti od 2 do n
  for (int p = 2; p <= n; p++) {
    // ako vrednost nije menjana, to znaci da je p prosto
    if (phi[p] == p) {
      // azuriramo vrednosti Ojlerove funkcije
      // za sve umnoske broja p
      for (int i = p; i <= n; i += p)
         phi[i] -= phi[i] / p;
    }
  }
  return phi;
}
```

Пример 3.3.4

Илуструјмо одређивање вредности Ојлерове функције свих бројева који су мањи или једнаки од броја 20.

- *Крећемо од низа у коме је на позицији* i *уписана вредност* i.

```
 1  2  3  4  5  6  7  8  9 10
11 12 13 14 15 16 17 18 19 20
```

- *У првом кораку се за све бројеве дељиве са* 2 *текућа вредност Ојлерове функције множи са* $1 - 1/2 = 1/2$.

```
 1  1  3  2  5  3  7  4  9  5
11  6 13  7 15  8 17  9 19 10
```

- *Након тога се разматра прост број* 3 *и за све бројеве дељиве са* 3 *текућа вредност се множи са* $1 - 1/3 = 2/3$.

```
 1  1  2  2  5  2  7  4  6  5
11  4 13  7 10  8 17  6 19 10
```

- *Број* 4 *се прескаче као сложен.*
- *За све бројеве дељиве са* 5 *текућа вредност се множи са* $1 - 1/5 = 4/5$.

```
 1  1  2  2  4  2  7  4  6  4
11  4 13  7  8  8 17  6 19  8
```

- *Број* 6 *се прескаче као сложен.*
- *За све бројеве дељиве са* 7 *текућа вредност се множи са* $1 - 1/7 = 6/7$.

```
 1  1  2  2  4  2  6  4  6  4
11  4 13  6  8  8 17  6 19  8
```

- *Бројеви* 8, 9 *и* 10 *се прескачу као сложени.*
- *Бројевима* 11, 13, 17 *и* 19 *се (као простим бројевима) вредност Ојлерове функције множи одговарајућим коефицијентима и тиме се ове вредности заправо постављају на* 10, 12, 16 *и* 18. *Не постоје бројеви мањи или једнаки* 20 *који су њима дељиви, те немамо додатних ажурирања.*

```
 1  1  2  2  4  2  6  4  6  4
10  4 12  6  8  8 16  6 18  8
```

Иако се са пролазом кроз делиоце не иде до $\sqrt{n}$ већ до n и иако се прецртавају сви умношци од p до n, уместо од p^2 до n, сложеност приказаног алгоритма и даље износи $O(n \log \log n)$, што је и сложеност основне варијанте Ератостеновог сита у коју су укључена сва ова одсецања.

Збир Ојлерових функција делилаца

Наредна лема даје једно интересантно својство Ојлерове функције које нам омогућава да на други начин одредимо вредности Ојлерове функције за све бројеве од 1 до n.

Лема 3.3.4 **[Збир Ојлерових функција $\varphi(d)$ делилаца d броја n]**

За сваки природан број n *важи:*

$$\sum_{d|n} \varphi(d) = n$$

Пре него што пређемо на доказ, размотримо наредни пример.

Пример 3.3.5

Нека је $n = 10$. *Напишимо све разломке* k/n *за* k *од* 1 *до* n *у скраћеном облику.*

$$\frac{1}{10}, \frac{1}{5}, \frac{3}{10}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{7}{10}, \frac{4}{5}, \frac{9}{10}, \frac{1}{1}.$$

Лако се примећује да су имениоци делиоци броја 10 *(сваки је добијен дељењем броја* 10 *неким бројем током скраћивања разломка) и да се за сваки такав именилац јављају сви бројиоци који су узајамно прости са њим. Тако се за именилац* 10 *јављају бројиоци*

$1, 3, 7, 9$ *и има их* $\varphi(10) = 4$*, за именилац* 5 *се јављају бројиоци* $1, 2, 3, 4$ *и има их* $\varphi(5) = 4$*, за именилац* 2 *се јавља само бројилац* 1 *и важи* $\varphi(2) = 1$ *и за именилац* 1 *се јавља само бројилац* 1 *и важи* $\varphi(1) = 1$*. У низу постоји* 10 *разломака и важи да је* $\varphi(10) + \varphi(5) + \varphi(2) + \varphi(1) = 10$*.*

Доказ. Закључак из претходног примера се лако уопштава на произвољно n тако што се размотре сви разломци од $\frac{1}{n}$ до $\frac{n}{n}$ у скраћеном облику. Њих има n. Са друге стране, њихови имениоци су сви делиоци $d|n$, и за сваки именилац d у овом низу постоји тачно $\varphi(d)$ разломака (сваком одговара бројилац узајамно прост са d). Зато у низу постоји тачно $\sum_{d|n} \varphi(d)$ елемената и тврђење је доказано. □

Претходна лема нам даје начин да коришћењем наредне формуле израчунамо $\varphi(n)$.

$$\varphi(n) = n - \sum_{d|n, d<n} \varphi(d).$$

Поступак је сличан Ератостеновом ситу. Иницијализују се све вредности у низу `fi` тако да се на позицији i налази вредност i. Затим пролазимо кроз све вредности d од 1 до n. Инваријанта алгоритма је то да се у тренутку обраде вредности d у низу на позицији d већ налази израчуната вредност $\varphi(d)$. Тада ту вредност одузимамо од свих умножака броја d. Инваријанта заиста важи, јер када се стигне до броја d од почетне вредности d су одузете све вредности функције φ за делиоце броја d (они су сви мањи од d па су тада већ обрађени).

Имплементација је једноставна.

```
vector<int> OjlerovaFunkcijaDoN(int n) {
  vector<int> phi(n+1);
  for (int i = 1; i <= n; i++)
     phi[i] = i;
  for (int d = 1; d <= n; d++)
     for (i = d+d; i <= n; i += d)
         phi[i] -= phi[d];
  return phi;
}
```

Пошто се у овом алгоритму унутрашња петља извршава за све вредности броја d, а не само за просте, сложеност алгоритма је $O(n \log n)$.

Пример 3.3.6

За $n = 10$*, алгоритам ради на следећи начин.*

- *Најпре иницијализујемо све вредности.*

```
i:    0  1  2  3  4  5  6  7  8  9  10
phi:     1  2  3  4  5  6  7  8  9  10
```

- *У том тренутку смо сигурни једино да је исправно израчуната вредност* $\varphi(1) = 1$*. Сви бројеви већи од* 1 *су умношци броја* 1*, па вредности у низу умањујемо за* $\varphi(1) = 1$*.*

```
i:    0  1  2  3  4  5  6  7  8  9  10
phi:     1  1  2  3  4  5  6  7  8  9
```

- *Сада смо сигурни да је* $\varphi(2) = 1$*, па вредности у низу на позицијама умножака броја* 2 *већих од* 2 *умањујемо за* $\varphi(2) = 1$*.*

```
i:    0  1  2  3  4  5  6  7  8  9  10
phi:     1  1  2  2  4  4  6  6  8  8
```

- *Сада смо сигурни да је* $\varphi(3) = 2$*, па вредности у низу на позицијама умножака броја* 3 *већих од* 3 *умањујемо за* $\varphi(3) = 2$*.*

```
i:    0  1  2  3  4  5  6  7  8  9  10
phi:     1  1  2  2  4  2  6  6  6  8
```

- *Сада смо сигурни да је* $\varphi(4) = 2$*, па вредности у низу на позицијама умножака броја* 4 *већих од* 4 *умањујемо за* $\varphi(4) = 2$*.*

```
i:    0  1  2  3  4  5  6  7  8  9  10
phi:     1  1  2  2  4  2  6  4  6  8
```

- *Сада смо сигурни да је* $\varphi(5) = 4$*, па вредности у низу на позицијама умножака броја* 5 *већих од* 5 *умањујемо за* $\varphi(5) = 4$*.*

```
i:    0  1  2  3  4  5  6  7  8  9  10
phi:     1  1  2  2  4  2  6  4  6  4
```

- *Пошто наредни бројеви немају умношке који су већи од њих а и даље су мањи од* $n = 10$*, поступак је завршен и у низу се налазе вредности Ојлерове функције од* 1 *до* n*.*

3.3.2 Функције делилаца

Функција делилаца (енгл. divisor function) је нека функција над скупом делилаца датог целог броја. У функције делилаца спадају број делилаца и збир делилаца датог броја које ћемо у даљем тексту детаљније разматрати. Уопштење ове две функције је функција збира степена делилаца броја:

$$\sigma_l(n) = \sum_{d|n} d^l$$

Тада је збир делилаца $\sigma_1(n)$, а број делилаца $\sigma_0(n)$.

Понекад се разматра и збир правих делилаца, у које спадају сви делиоци осим n. У зависности од тога да ли је збир правих делилаца броја мањи, једнак или већи од њега самог, бројеви се могу класификовати на дефицитарне (енгл. deficit), савршене (енгл. perfect) или обилне (енгл. abundant). Збир правих делилаца броја се назива и *аликвотна сума* (енгл. aliquot sum) и лако израчунава када је познат збир свих делилаца.

3.3.2.1 Број делилаца

Проблем

За дати број n израчунати укупан број његових делилаца $\sigma_0(n)$.

Пример 3.3.7

Делиоци броја $n = 24$ су $1, 2, 3, 4, 6, 8, 12$ и 24 и укупно их има 8, па је $\sigma_0(24) = 8$.

Директан алгоритам

Приметимо да је сваки број дељив бројем 1 и да број n увек дели сам себе те је број делилаца броја n увек већи или једнак 2 (осим када је $n = 1$). Делиоци 1 и n су тривијални делиоци броја n. Једноставни алгоритам за одређивање броја делилаца броја n пролази скупом свих бројева од 1 до n и за сваку вредност која дели број n се укупан број делилаца увећава за 1. Пошто нема делилаца између $n/2$ и n, алгоритам се може мало усавршити тако што се делилац n обради засебно (на пример, пре петље), док се у петљи пролази скупом бројева од 1 до $n/2$. Сложеност овог алгоритма је $O(n)$.

Тражење парова делилаца

Приметимо да се делиоци скоро увек јављају у пару: у претходном примеру имамо да је $24 = 1 \cdot 24 = 2 \cdot 12 = 3 \cdot 8 = 4 \cdot 6$. Ово не важи само када је n квадрат неког броја: на пример делиоци броја 16 су $1, 2, 4, 8$ и 16, где средишњи делилац 4 нема свог пара (тј. сам је свој пар). Приметимо да је мањи од бројева који чине пар мањи од $\sqrt{n}$ (а ако

је број потпун квадрат, тада је број $\sqrt{n}$ сам себи пар). Стога је могуће формулисати ефикаснији алгоритам који пролази скупом свих бројева мањих или једнаких $\sqrt{n}$ и за сваки број d који дели број n ако је $d \neq \frac{n}{d}$ укупан број делилаца увећава за 2, а ако је $d = \frac{n}{d}$ за 1.

```
// funkcija koja racuna broj delilaca broja n
// razmatrajuci parove delilaca
int brojDelilaca(int n) {
  int br = 0;
  for (int i = 1; i * i <= n; i++)
    if (n % i == 0)
      if (i * i != n)
        br += 2;
      else
        br++;
  return br;
}
```

Приметимо да смо овом малом модификацијом претходног алгоритма, смањили асимптотску сложеност алгоритма на $O(\sqrt{n})$.

Свођење на проблем факторизације

Функција броја делилаца σ_0 је мултипликативна.

Лема 3.3.5 **[Мултипликативност функције броја делилаца σ_0]**

Функција σ_0 је мултипликативна, тј. за свака два узајамно проста броја M и N важи $\sigma_0(MN) = \sigma_0(M) \cdot \sigma_0(N)$.

Доказ. Означимо са D_K скуп делилаца броја K. Функција $(m, n) \mapsto mn$ је бијекција између скупова $D_M \times D_N$ и D_{MN}. Заиста, тривијална чињеница је да ако $m|M$ и $n|N$, тада $mn|MN$. Међутим, важи и обратно: сваки делилац броја MN је производ нека два делиоца $m|M$ и $n|N$. Заиста, ако је d неки делилац броја MN он се може факторисати. Нека је m производ његових простих чинилаца који деле M, а n производ његових простих чинилаца који деле N. Пошто су M и N узајамно прости ниједан делилац броја m не дели N и ниједан делилац броја n не дели M. Зато m дели M, а n дели N. Пошто је успостављена бијекција између два коначна скупа, важи:

$$\sigma_0(MN) = |D_{MN}| = |D_M \times D_N| = |D_M| \cdot |D_N| = \sigma_0(M) \cdot \sigma_0(N) \qquad \square$$

Дакле, довољно је само да умемо да израчунамо вредности $\sigma_0(p^k)$.

Лема 3.3.6 **[Рачунање $\sigma_0(p^k)$]**

Нека је p прост број. Тада је $\sigma_0(p^k) = k + 1$.

Доказ. Сви делиоци броја p^k су $1, p, \ldots, p^k$, којих има $k+1$, па је $\sigma_0(p_k) = k+1$. □

Лема 3.3.7 **[Рачунање $\sigma_0(n)$]**

Ако је $n = p_1^{k_1} \cdot \ldots \cdot p_m^{k_m}$, разлагање броја n на просте чиниоце, тада је $\sigma_0(n) = (k_1 + 1) \cdot \ldots \cdot (k_m + 1)$.

Доказ. $\sigma_0(n) = \sigma_0(p_1^{k_1} \cdot \ldots \cdot p_m^{k_m}) = \sigma_0(p_1^{k_1}) \cdot \ldots \cdot \sigma_0(p_m^{k_m}) = (k_1+1) \cdot \ldots \cdot (k_m+1)$. □

Ово је заправо прилично очигледно, јер је општи облик сваког делиоца броја n једнак $d = p_1^{l_1} \cdot p_2^{l_2} \cdot \ldots \cdot p_m^{l_m}$, где за свако $1 \leq i \leq m$ важи $0 \leq l_i \leq k_i$. Дакле, за експонент простог чиниоца p_i у делиоцу d постоји $k_i + 1$ различитих могућности, те је укупан број делилаца броја n једнак $(k_1 + 1) \cdot (k_2 + 1) \cdot \ldots \cdot (k_r + 1)$. Приметимо да у овој формули фигуришу само експоненти простих чинилаца, а не и сами чиниоци.

Пример 3.3.8

Број 24 можемо представити на следећи начин $24 = 2^3 \cdot 3^1$. Сви његови делиоци су $2^0 \cdot 3^0$, $2^0 \cdot 3^1$, $2^1 \cdot 3^0$, $2^1 \cdot 3^1$, $2^2 \cdot 3^0$, $2^2 \cdot 3^1$, $2^3 \cdot 3^0$ и $2^3 \cdot 3^1$ и има их $(3+1) \cdot (1+1) = 8$.

Имплементација се добија једноставном модификацијом алгоритма факторизације.

```
int brojDelilaca(int n) {
  // ukupan broj delilaca broja n
  int broj = 1;
  // prolazimo skupom svih brojeva od 2 do sqrt(n)
  for (int i = 2; i * i <= n; i++) {
    // broj puta koliko broj i deli n
    int k = 0;
    while (n % i == 0) {
      n /= i;
      k++;
    }
    // ukupan broj delilaca mnozimo sa k+1
    broj *= (k + 1);
```

```
  }
  // ako je preostali broj prost, njegova vrednost za k je 1,
  // te je prethodni rezultat potrebno pomnoziti sa k + 1 = 2
  if (n > 1)
    broj *= 2;

  return broj;
}
```

Сложеност претходног алгоритма за рачунање броја делилаца броја n одговара сложености факторизације броја n, те износи $O(\sqrt{n})$. Приметимо да ако нам је унапред позната факторизација броја n, број делилаца можемо лако ефективно одредити.

Одређивање броја делилаца бројева од 1 до n

По узору на Ератостеново сито, лако је одредити број делилаца сваког броја од 1 до n. Број 1 је делилац сваког броја, па број делилаца сваког броја иницијализујемо на 1. Затим пролазимо редом кроз све могуће делиоце d и број делилаца њихових умножака (кренувши од самог d) увећавамо за 1.

```
vector<int> brojDelilaca(n+1, 1);
for (int d = 2; d <= n; d++)
   for (int k = d; k <= n; k+=d)
      brojDelilaca[k]++;
```

Сложеност овог алгоритма је $O(n \log n)$, што је ефикасније од израчунавања броја делилаца сваког броја појединачно, које би било сложености $O(n\sqrt{n})$.

3.3.2.2 Збир делилаца

Проблем

За дати број n израчунати збир свих његових делилаца.

Пример 3.3.9

Ако је $n = 24$, његови делиоци су $1, 2, 3, 4, 6, 8, 12$ и 24 и њихов збир једнак је 60.

Директан алгоритам

Наивни приступ за решавање овог проблема је проћи скупом свих бројева мањих од n и увећавати текућу суму делилаца за вредност сваког од бројева који дели број n. Сложеност овог приступа је $O(n)$.

Тражење парова делилаца

Ефикаснији приступ је да се делиоци откривају у пару, тако што се пролази скупом свих бројева мањих или једнаких $\sqrt{n}$ и за сваки број i који дели број n вредност i и вредност њему одговарајућег делиоца n/i се додаје на текућу суму, осим када је баш $i = n/i$, односно када је $i = \sqrt{n}$, када се на суму додаје само вредност i.

```
// funkcija koja racuna sumu delilaca broja n
// razmatrajuci parove delilaca
int zbirDelilaca(int n) {
  int zbir = 0;
  for (int i = 1; i * i <= n; i++)
    if (n % i == 0)
      if (i * i != n)
        zbir += (i + n/i);
      else
        zbir += i;
  return zbir;
}
```

Сложеност овог алгоритма је $O(\sqrt{n})$.

Свођење на проблем факторизације

И функција збира делилаца је мултипликативна. Докажимо још општије да су све функције σ_l мултипликативне (подсетимо се, σ_l израчунава збир l-тих степена делилаца).

Лема 3.3.8 **[Мултипликативност функције збира делилаца σ_l]**

Функција σ_l, $l \in \mathbb{N}_0$, је мултипликативна, тј. ако су M и N узајамно прости бројеви, тада је $\sigma_l(MN) = \sigma_l(M) \cdot \sigma_l(N)$.

Доказ. Нека су $D_M = \{a_1, \ldots, a_m\}$ сви делиоци броја M, $D_N = \{b_1, \ldots, b_n\}$ сви делиоци броја n, а $D_{MN} = \{c_1, \ldots, c_{mn}\}$ сви делиоци броја MN. Тада је

$$\sigma_l(M)\cdot\sigma_l(N) = (a_1^l + \ldots + a_m^l)\cdot(b_1^l + \ldots + b_n^l) = \left(\sum_{i=1}^{m} a_i^l\right)\cdot\left(\sum_{i=1}^{n} b_i^l\right) = \sum_{\substack{1\le i\le m,\\ 1\le j\le n}} a_i^l b_j^l.$$

Већ смо у леми 3.3.5 доказали да је $(x, y) \mapsto xy$ бијекција између скупова $D_M \times D_N$ и D_{MN}. Зато сваком производу $a_i b_j$ за $1 \le i \le m$ и $1 \le j \le n$ одговара неки $c_k \in D_{MN}$ за $1 \le k \le mn$, и обратно. Зато је

$$\sigma_l(M)\cdot\sigma_l(N)=\sum_{\substack{1\le i\le m,\\ 1\le j\le n}} a_i^l b_j^l=\sum_{\substack{1\le i\le m,\\ 1\le j\le n}} (a_ib_j)^l=\sum_{k=1}^{mn} c_k^l=\sigma_l(MN). \qquad \square$$

Дакле, довољно је да знамо вредност $\sigma_1(p^k)$.

Лема 3.3.9 **[Рачунање $\sigma_1(p^k)$]**

Нека је p прост број. Тада је $\sigma_1(p^k)=\frac{p^{k+1}-1}{p-1}$.

Доказ. Сви делиоци броја p^k су бројеви $1, p, \dots, p^k$. Зато важи да је

$$\sigma_1(p^k)=1+p+\dots+p^k=\frac{p^{k+1}-1}{p-1}. \qquad \square$$

Лема 3.3.10 **[Рачунање $\sigma_1(n)$]**

Ако је $n=p_1^{k_1}\cdot\dots\cdot p_m^{k_m}$ разлагање броја n на просте чиниоце, тада је

$$\sigma_1(n)=\frac{p_1^{k_1+1}-1}{p_1-1}\cdot\dots\cdot\frac{p_m^{k_m+1}-1}{p_m-1}.$$

Доказ. Важи:

$$\sigma_1(n)=\sigma_1(p_1^{k_1}\cdot\dots\cdot p_m^{k_m})=\sigma_1(p_1^{k_1})\cdot\dots\cdot\sigma_1(p_m^{k_m})=\frac{p_1^{k_1+1}-1}{p_1-1}\cdot\dots\cdot\frac{p_m^{k_m+1}-1}{p_m-1}. \qquad \square$$

До овог резултата смо могли доћи и директно. Општи облик делиоца броја n једнак је $d=p_1^{l_1}\cdot p_2^{l_2}\cdot\dots\cdot p_m^{l_m}$, где је $0\le l_i\le k_i$. Приметимо да се у наредном изразу после ослобађања од заграда сваки делилац броја n појављује у суми тачно једном:

$$(1+p_1+p_1^2+\dots+p_1^{k_1})\cdot(1+p_2+p_2^2+\dots+p_2^{k_2})\cdot\dots\cdot(1+p_m+p_m^2+\dots+p_m^{k_m})$$

односно израз је једнак суми делилаца броја n. Дакле, проблем се своди на идентификовање простих чиналаца броја n, односно на факторизацију броја n. Иако је на

папиру једноставније да уместо суме $1 + p_i + p_i^2 + ... + p_i^{k_i}$ применимо формулу за збир геометријског низа и добијемо $\frac{p_i^{k_i+1}-1}{p_i-1}$, у имплементацији степеновање можемо избећи ако тражени збир рачунамо инкрементално, чиме добијамо и бржи и једноставнији код. Наиме, познати збир $S = 1 + p_i + ... + p_i^k$ можемо ажурирати додавањем сабирка p_i^{k+1}, али и множењем са p_i и сабирањем са 1.

```
int zbirDelilaca(int n) {
  // ukupan zbir delilaca broja n
  int zbir = 1;
  // prolazimo skupom svih brojeva od 2 do sqrt(n)
  for (int p = 2; p * p <= n; p++) {
     // 1 + p + ... + p^k
    int S = 1;
    while (n % p == 0) {
      n /= p;
      S = S * p + 1;
    }
    // azuriramo vrednost zbira delilaca
    zbir *= S;
  }
  // ako je preostali broj n prost, njegovi delioci su 1 i n
  if (n > 1)
    zbir *= (1 + n);
  return zbir;
}
```

Сложеност алгоритма за рачунање суме свих делилаца заснованог на факторизацији је $O(\sqrt{n})$.

Одређивање збира делилаца бројева од 1 до n

Веома слично броју делилаца, по узору на Ератостеново сито, алгоритмом сложености $O(n \log n)$ можемо одредити и збирове делилаца свих бројева од 1 до n.

```
vector<int> zbirDelilaca(n+1, 1);
for (int d = 2; d <= n; d++)
   for (int k = d; k <= n; k+=d)
      zbirDelilaca[k] += d;
```

Задатак: Број делилаца производа

Напиши програм који израчунава број делилаца производа унетог низа природних бројева.

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 10^6$), а затим низ од n бројева између 1 и 10^6 раздвојених са по једним размаком.

Опис излаза

На стандардни излаз исписати остатак при дељењу траженог броја делилаца при дељењу са $10^9 + 7$.

Пример 1

Улаз	*Излаз*	*Објашњење*
`2`	`8`	$4 \cdot 14 = 56$, а делиоци броја 56 су $1, 2, 4, 7, 8, 14, 28, 56$ и има их осам.
`4 14`		

Пример 2

Улаз	*Излаз*
`3`	`40`
`10 20 30`	

Решење

Разлагање сваког броја појединачно на просте чиниоце

Број делилаца се може израчунати, коришћењем мултипликативности функције σ_0, која израчунава број делилаца датог броја. За ово је потребно да се производ свих бројева растави на просте чиниоце. Довољно је одредити експонент сваког простог чиниоца у том разлагању. Сваки број можемо независно разложити на просте чиноице. Можемо одржавати низ експонената свих потенцијалних чинилаца производа (они су сви мањи или једнаки од највећег броја у низу) и чим елемент низа разложимо на просте чиниоце, можемо ажурирати низ експонената простих чинилаца производа.

Сложеност растављања сваког броја N на просте чиниоце врши се у сложености $O(\sqrt{N})$. Нека је M горње ограничење вредности елемената низа. Најгори случај наступа када су сви елементи низа велики прости бројеви (они могу бити и једнаки) или квадрати простих бројева, блиски вредности M. Сложеност растављања свих бројева на просте чиниоце је $O(n\sqrt{M})$. Број чинилаца није већи од $\log M$, а можемо га и сматрати константним (сваки број који стаје у опсег типа `int` има највише тридесетак простих чинилаца). Израчунавање крајњег производа извршава се у сложености $O(M)$ (што се може смањити на број различитих простих чинилаца који се јављају у производу, ако се уместо низа употреби мапа).

Ератостеново сито

Ако је број елемената велики, тада се уместо независног растављања сваког броја на просте чиниоце више исплати примена модификације Ератостеновог сита.

Нека је M максималан број у низу. Да бисмо могли брзо да факторишемо велики број бројева из интервала $[1, M]$, довољно је да за сваки број из тог интервала одредимо најмањи прост чинилац, што се ефикасно може урадити модификацијом Ератостеновог сита. Након тога можемо растављати на чиниоце један по један елемент низа, чувајући глобални низ у ком ћемо памтити експоненте свих простих чинилаца производа. Тај низ једнозначно одређује факторизацију производа свих бројева.

Када је факторизација производа позната, број делилаца се одређује коришћењем својства мултипликативности функције σ_0 која израчунава број делилаца.

Ако се у низу бројеви често понављају, да их не бисмо изнова растављали на чиниоце, можемо пребројати појављивања сваког елемента у низу, па приликом његове факторизације (коју ћемо вршити само једном), тј. дељења са неким простим чиниоцем p_i експонент тог простог чиниоца увећавати за број појављивања тог елемента (а не за 1).

За сваки елемент из интервала $[1, M]$ најмањи прост чинилац се одређује модификацијом Ератостеновог сита у сложености $O(M \log \log M)$. Након тога се факторизација сваког броја врши у сложености $O(\log M)$ (јер сваки број из интервала $[1, M]$ може имати највише $\log M$ простих чинилаца). Укупна сложеност је зато $O(M \log \log M + n \log M)$.

```
// broj delilaca proizvoda elemenata niza a
long long brojDelilacaProizvoda(const vector<int>& a) {
  // najveci element u nizu a
  int max = *max_element(begin(a), end(a));

  // za svaki broj od 1 do max odredjujemo najmanji prost cinilac
  vector<int> najmanjiCinilac(max+1);
  for (int i = 1; i <= max; i++)
    najmanjiCinilac[i] = i;
  for (int i = 2; i * i <= max; i++)
    if (najmanjiCinilac[i] == i)
      for (int j = i*i; j <= max; j += i)
        najmanjiCinilac[j] = i;

  // za svaki prosti cinilac proizvoda pamtimo eksponent u sklopu
  // jedinstvenog razlaganja proizvoda na proste cinioce - cinilac se
```

```
  // preslikava u eksponent
  vector<int> eksponentiProstihCinilaca(max+1, 0);

  // rastavljamo svaki broj na proste cinioce
  for (int x : a) {
    while (x != 1) {
      eksponentiProstihCinilaca[najmanjiCinilac[x]]++;
      x /= najmanjiCinilac[x];
    }
  }

  // izracunavamo broj delilaca proizvoda koriscenjem multiplikativnosti
  // funkcije broja delilaca
  long long broj = 1;
  for (int e : eksponentiProstihCinilaca)
    broj = (broj * (e + 1)) % MOD;

  return broj;
}
```

Задатак: Збир НЗД свих парова делилаца

Напиши програм који израчунава збир НЗД свих парова делилаца броја N тј. израчунава

$$\sum_{i|N} \sum_{j|N} \text{nzd}(i, j).$$

Опис улаза

Са стандардног улаза се учитава један природан број N ($1 \leq N \leq 10^{12}$).

Опис излаза

На стандардни излаз исписати тражени збир.

Пример 1

Улаз	*Излаз*	*Објашњење*
`10`	`40`	Делиоци броја 10 су 1, 2, 5 и 10. НЗД парова су $(1,1)=1$, $(1,2)=1$, $(1,5)=1$, $(1,10)=1$, $(2,1)=1$, $(2,2)=2$, $(2,5)=1$, $(2,10)=2$, $(5,1)=1$, $(5,2)=1$, $(5,5)=5$, $(5,10)=5$, $(10,1)=1$, $(10,2)=2$, $(10,5)=5$, $(10,10)=10$. Њихов збир је 40.

Пример 2

Улаз	*Излаз*
`100`	`675`

Решење

Груба сила

Тражени збир се може израчунати по дефиницији. Прво проналазимо све делиоце броја. Пошто се делиоци јављају у пару (ако је делилац број d, делилац је и број $\frac{N}{d}$), довољно је испитивати само потенцијалне делиоце који су мањи од $\sqrt{N}$. Након тога за сваки пар делилаца Еуклидовим алгоритмом израчунавамо НЗД и добијене вредности сабирамо.

Одређивање свих делилаца врши се у сложености $O(\sqrt{N})$. Затим се анализира сваки пар делилаца. Ако делилаца има D, парова има $O(D^2)$. Имајући у виду ограничења дата у тексту задатка, вредност D може да нарасте до неколико хиљада, па овај квадратни фактор може значајно да деградира ефикасност решења. За сваки пар се одређује НЗД у сложености $O(\log N)$, јер је сваки делилац у пару мањи или једнак од N.

Мултипликативност и растављање на просте чиниоце

Нека је

$$f(N) = \sum_{i|N} \sum_{j|N} \text{nzd}(i,j).$$

Функција f је мултипликативна, што значи да за било која два узајамно проста броја p и q важи $f(p \cdot q) = f(p) \cdot f(q)$. Мултипликативност омогућава да се до решења дође разлагањем броја N на производ простих чинилаца. Наиме, ако је $N = p_1^{k_1} \cdot \ldots \cdot p_m^{k_m}$, тада је $f(N) = f(p_1^{k_1}) \cdot \ldots \cdot f(p_m^{k_m})$.

Вредност $f(p^k)$, где је прост број се може израчунати релативно једноставно. Наиме, делиоци броја p^k су редом p^0, p^1, ..., p^k. НЗД два таква делиоца p^i p^j једнак је $p^{\min(i,j)}$. Зато је

$$f(p^k) = \sum_{i=0}^{k} \sum_{j=0}^{k} p^{\min(i,j)}.$$

Ова се сума може и даље упрошћавати, међутим, пошто је k увек прилично мали број може се израчунавати и директно.

```
long long zbirNZDDelilacaProstog(long long p, int k) {
  vector<long long> stepeni(k+1);
  stepeni[0] = 1;
  for (int i = 1; i <= k; i++)
    stepeni[i] = stepeni[i-1] * p;

  long long zbir = 0;
  for (int i = 0; i <= k; i++)
    for (int j = 0; j <= k; j++)
      zbir += stepeni[min(i, j)];
  return zbir;
}

long long zbirNZDDelilaca(long long n) {
  long long zbir = 1;
  for (int d = 2; d*d <= n; d++) {
    int k = 0;
    while (n % d == 0) {
      n /= d;
      k++;
    }
    if (k > 0)
      zbir *= zbirNZDDelilacaProstog(d, k);
  }
  if (n > 1)
    zbir *= zbirNZDDelilacaProstog(n, 1);
  return zbir;
}
```

Докажимо да је функција мултипликативна. Нека су p и q узајамно прости бројеви.

Израчунавамо

$$f(pq) = \sum_{i|pq} \sum_{j|pq} \text{nzd}(i, j)$$

Сваки делилац броја pq се добија као производ једног делиоца броја p и једног делиоца броја q. Зато је

$$f(pq) = \sum_{a_1|p,b_1|q} \sum_{a_2|p,b_2|q} \text{nzd}(a_1b_1, a_2b_2)$$

Редослед сумирања се може променити

$$f(pq) = \sum_{a_1,a_2|p} \sum_{b_1,b_2|q} \text{nzd}(a_1b_1, a_2b_2)$$

Пошто су бројеви p и q узајамно прости, узајамно прости су и сви парови бројева (a_1, b_1), (a_1, b_2), (a_2, b_1) и (a_2, b_2), па је $\text{nzd}(a_1b_1, a_2b_2) = \text{nzd}(a_1, a_2) \cdot \text{nzd}(b_1, b_2)$.

Зато је

$$\begin{aligned} f(pq) &= \sum_{a_1,a_2|p} \sum_{b_1,b_2|q} (\text{nzd}(a_1, a_2) \cdot \text{nzd}(b_1, b_2)) \\ &= \sum_{a_1,a_2|p} \text{nzd}(a_1, a_2) \cdot \left(\sum_{b_1,b_2|q} \text{nzd}(b_1, b_2) \right) \\ &= \left(\sum_{a_1,a_2|p} \text{nzd}(a_1, a_2) \right) \cdot \left(\sum_{b_1,b_2|q} \text{nzd}(b_1, b_2) \right) \\ &= f(p) \cdot f(q) \end{aligned}$$

Пример 3.3.10

Илуструјмо кораке претходног доказа и на примеру израчунавања $f(2 \cdot 5)$. *Збир НЗД свих парова делилаца броја* 10 *је следећи:*

$$\begin{array}{ccccccccc} (1,1) & + & (1,2) & + & (1,5) & + & (1,10) & + \\ (2,1) & + & (2,2) & + & (2,5) & + & (2,10) & + \\ (5,1) & + & (5,2) & + & (5,5) & + & (5,10) & + \\ (10,1) & + & (10,2) & + & (10,5) & + & (10,10) & \end{array}$$

Сваки делилац броја 10 *је производ једног делиоца броја* 2 *и једног делиоца броја* 5.

$$\begin{array}{ccccccccc} (1\cdot1,1\cdot1) & + & (1\cdot1,2\cdot1) & + & (1\cdot1,1\cdot5) & + & (1\cdot1,2\cdot5) & + \\ (2\cdot1,1\cdot1) & + & (2\cdot1,2\cdot1) & + & (2\cdot1,1\cdot5) & + & (2\cdot1,2\cdot5) & + \\ (1\cdot5,1\cdot1) & + & (1\cdot5,2\cdot1) & + & (1\cdot5,1\cdot5) & + & (1\cdot5,2\cdot5) & + \\ (2\cdot5,1\cdot1) & + & (2\cdot5,2\cdot1) & + & (2\cdot5,1\cdot5) & + & (2\cdot5,2\cdot5) & \end{array}$$

Након реорганизације редоследа тако да се прво фиксира пар делилаца броја 2*, а затим пар делилаца броја* 5 *добија се следећи збир.*

$$\begin{array}{llllllll} (1\cdot 1,1\cdot 1) & + & (1\cdot 1,1\cdot 5) & + & (1\cdot 5,1\cdot 1) & + & (1\cdot 5,1\cdot 5) & + \\ (1\cdot 1,2\cdot 1) & + & (1\cdot 1,2\cdot 5) & + & (1\cdot 5,2\cdot 1) & + & (1\cdot 5,2\cdot 5) & + \\ (2\cdot 1,1\cdot 1) & + & (2\cdot 1,1\cdot 5) & + & (2\cdot 5,1\cdot 1) & + & (2\cdot 5,1\cdot 5) & + \\ (2\cdot 1,2\cdot 1) & + & (2\cdot 1,2\cdot 5) & + & (2\cdot 5,2\cdot 1) & + & (2\cdot 5,2\cdot 5) & \end{array}$$

НЗД производа је производ НЗД-ова.

$$\begin{array}{llllllll} (1,1)\cdot(1,1) & + & (1,1)\cdot(1,5) & + & (1,1)\cdot(5,1) & + & (1,1)\cdot(5,5) & + \\ (1,2)\cdot(1,1) & + & (1,2)\cdot(1,5) & + & (1,2)\cdot(5,1) & + & (1,2)\cdot(5,5) & + \\ (2,1)\cdot(1,1) & + & (2,1)\cdot(1,5) & + & (2,1)\cdot(5,1) & + & (2,1)\cdot(5,5) & + \\ (2,2)\cdot(1,1) & + & (2,2)\cdot(1,5) & + & (2,2)\cdot(5,1) & + & (2,2)\cdot(5,5) & \end{array}$$

Извлачимо први НЗД испред заграде

$$\begin{array}{lllllllll} (1,1) & \cdot & ((1,1) & + & (1,5) & + & (5,1) & + & (5,5)) & + \\ (1,2) & \cdot & ((1,1) & + & (1,5) & + & (5,1) & + & (5,5)) & + \\ (2,1) & \cdot & ((1,1) & + & (1,5) & + & (5,1) & + & (5,5)) & + \\ (2,2) & \cdot & ((1,1) & + & (1,5) & + & (5,1) & + & (5,5)) & + \end{array}$$

На крају, извлачимо заграду испред заграде (десно):

$$((1,1)+(1,2)+(2,1)+(2,2))\cdot((1,1)+(1,5)+(5,1)+(5,5))$$

Израчунавањем НЗД добијамо $(1+1+1+2)\cdot(1+1+1+5)=5\cdot 8=40$.

Растављање броја N на просте чиниоце се врши у сложености $O(\sqrt{N})$. За сваки прост фактор p^k збир НЗД-ова се рачуна у сложености $O(k^2)$, што се може сматрати практично константним, с обзиром на јако мале могуће вредности k.

3.4 Модуларна аритметика

У основи модуларне аритметике лежи операција одређивања целобројног количника и остатка. Подсетимо се, број је *q количник*, а број r *остатак* при дељењу броја x бројем y ако и само ако постоје бројеви q и r такви да важи $x = q \cdot y + r$ и $0 \leq r < y$. Бројеви q и r су овим условом јединствено одређени. Писаћемо $q = a \operatorname{div} b$ и $r = a \bmod b$.

Поред тога што са mod означавамо бинарну операцију, mod се користи и као ознака релације у скупу целих бројева. Наиме, писаћемо $a \equiv b \pmod{m}$ и рећи да су a и b *конгруентни по модулу* m ако $m|a-b$ тј. ако је $a \bmod m = b \bmod m$, односно ако a и b дају исти остатак при дељењу са m. На пример, $12 \equiv 2 \pmod{5}$ јер $5|(12-2)$.

Релацију mod користимо у разним свакодневним ситуацијама, а да тога често нисмо ни свесни. Један од таквих примера је рад са временом. Наиме, за време које је 15 часова након 11 сати рећи ћемо да је 2 сата (што одговара томе да је $11 + 15 \equiv 2 \pmod{24}$). Слично важи и за дане у недељи које можемо обележити бројевима од 0 до 6 (на пример, од недеље, до суботе). Операције вршимо по модулу 7. На пример, ако је данас четвртак (дан обележен бројем 4), за шест дана биће среда (дан обележен бројем 3), јер је $4 + 6 \equiv 3 \pmod{7}$. Аналогно се рачуна и месец или редни број недеље у години. Модуларна аритметика има још пуно практичних примена, поменимо само неке занимљиве: користи се за израчунавање контролних сума за међународне стандардне идентификаторе књига (ISBN бројеве), међународне бројеве банковних рачуна (IBAN), као и јединствене идентификаторе хемијских једињења (CAS регистарски број). Модуларна аритметика је и у основи неких савремених криптографских система.

Неозначени цели бројеви се у рачунарима представљају са k битова, а операције над њима се изводе по модулу 2^k. Рецимо, у језику C++ бројеви типа `unsigned int` су ширине 32 бита, па се операције над њима изводе по модулу 2^{32}.

Пример 3.4.1

У наредном коду као резултат квадрирања броја 123456789 *добија се вредност* $123456789^2 \bmod 2^{32} = 2537071545$.

```
unsigned int x = 123456789;
unsigned int y = x * x;
cout << y << endl;
```

Ако је $a_1 \equiv b_1 \pmod{m}$ и $a_2 \equiv b_2 \pmod{m}$, онда је $a_1 \pm a_2 \equiv b_1 \pm b_2 \pmod{m}$ и $a_1 \cdot a_2 \equiv b_1 \cdot b_2 \pmod{m}$; на пример,

$$\begin{array}{ccc} 12,14 & \overset{\text{mod } 5}{\longrightarrow} & 2,4 \\ +\downarrow & & \downarrow+ \\ 26 & \overset{\text{mod } 5}{\longrightarrow} & 1 \end{array}$$

Другим речима, релација mod је *сагласна* (конгруентна) са операцијама сабирања, одузимања и множења. Резимирајмо основне особине когруенција у наредној леми.

Лема 3.4.1 **[Основне особине конгруенције]**

Конгруенција по модулу задовољава следеће особине:

1. *Важи* $a \equiv b \pmod m$ *ако и само ако* $m|(a-b)$. *Важи* $a \equiv 0 \pmod m$ *ако и само ако* $m|a$.
2. *Конгруенција по модулу је релација еквиваленције (важи* $a \equiv a \pmod m$*, ако важи* $a \equiv b \pmod m$ *тада важи и* $b \equiv a \pmod m$ *и ако важи* $a \equiv b \pmod m$ *и* $b \equiv c \pmod m$ *тада важи и* $a \equiv c \pmod m$*).*
3. *Ако важи* $a_1 \equiv b_1 \pmod m$ *и* $a_2 \equiv b_2 \pmod m$ *тада важи* $a_1 + a_2 \equiv b_1 + b_2 \pmod m$, $a_1 - a_2 \equiv b_1 - b_2 \pmod m$ *и* $a_1 \cdot a_2 \equiv b_1 \cdot b_2 \pmod m$.
4. *Ако важи* $a \equiv b \pmod m$*, онда за сваки природан број* k *важи* $a^k \equiv b^k \pmod m$.
5. *Ако су* c *и* m *узајамно прости (важи* $\text{nzd}(c,m) = 1$*), тада из* $c \cdot a \equiv c \cdot b \pmod m$ *следи* $a \equiv b \pmod m$ *(конгруенција се може скратити са* c*).*
6. *Ако важи* $c \cdot a \equiv c \cdot b \pmod{c \cdot m}$ *тада важи и* $a \equiv b \pmod m$ *(конгруенција се може скратити са* c*).*

Доказ. Докажимо само последње две ставке које ћемо звати први и други закон скраћивања (остале се доказују директно на основу дефиниције).

Претпоставимо да је $\text{nzd}(c,m) = 1$ и да важи $c \cdot a \equiv c \cdot b \pmod m$ тј. да важи $m|(c \cdot a - c \cdot b)$. Зато постоји неки природан број k такав да је $c \cdot (a-b) = km$. Пошто су c и m узајмно прости, сви прости фактори броја m морају бити садржани у броју $a-b$, па постоји неки природан број j такав да је $a - b = jm$, одакле следи $m|(a-b)$ тј. $a \equiv b \pmod m$.

Ако важи $c \cdot a \equiv c \cdot b \pmod{c \cdot m}$, тада постоји природан број k такав да је $(c \cdot a - c \cdot b) = k \cdot c \cdot m$. Скраћивањем са c добија се $a - b = k \cdot m$, из чега тврђење следи.

Важи и обратан смер, међутим, ову лему ћемо користити увек за скраћивање конгруенција, а не за њихово проширивање. □

Пример 3.4.2

Први закон скраћивања је могуће применити само када су бројеви c и m узајамно прости. Из $4 \equiv 10 \pmod 6$ не следи $2 \equiv 5 \pmod 6$, међутим, на основу другог закона скраћивања следи $2 \equiv 5 \pmod 3$.

Ово инспирише следеће једнакости које се користе за рачунање вредности збира или производа бројева по модулу m.

Лема 3.4.2 **[Сабирање и множење по модулу]**

$$\begin{aligned}(a+b) \bmod m &= (a \bmod m + b \bmod m) \bmod m \\ (a \cdot b) \bmod m &= (a \bmod m \cdot b \bmod m) \bmod m\end{aligned}$$

Доказ. Докажимо другу једнакост (прва је једноставнија за доказивање).

Претпоставимо да је $a = q_a \cdot m + r_a$ и $b = q_b \cdot m + r_b$ за $0 \leq r_a, r_b < m$. Тада је:

$$a \cdot b = (q_a \cdot m + r_a) \cdot (q_b \cdot m + r_b) = (q_a \cdot q_b \cdot m + q_a \cdot r_b + r_a \cdot q_b)m + r_a \cdot r_b.$$

Ако је $r_a \cdot r_b = q \cdot m + r,\ 0 \leq r < m$, тада је:

$$a \cdot b = (q_a \cdot q_b \cdot m + q_a \cdot r_b + r_a \cdot q_b + q)m + r,\ 0 \leq r < m$$

па је $(a \cdot b) \bmod m = r$.

Важи, такође, да је $(a \bmod m \cdot b \bmod m) \bmod m = (r_a \cdot r_b) \bmod m = r$, чиме је тврђење доказано. □

Рачунање на основу претходних једнакости је важно, јер често омогућава да се избегну грешке настале услед прекорачења. Наиме, ако су бројеви a и b релативно велики, а m мали, тада изрази $a + b$ и $a \cdot b$ могу довести до прекорачења, па се самим тим могу добити нетачни резултати приликом израчунавања вредности израза $(a + b) \bmod m$ и $(a \cdot b) \bmod m$. Са друге стране, у изразима $(a \bmod m + b \bmod m) \bmod m$ и $(a \bmod m \cdot b \bmod m) \bmod m$ се рачуна са мањим бројевима и они најчешће не доводе до прекорачења.

Пример 3.4.3

На пример, ако је $a = b = 3 \cdot 10^9 + 9$, *а* $m = 10$, *тада је* $a + b = 6 \cdot 10^9 + 18$ *и не може се исправно представити помоћу* 32 *бита. Важи да је* $a \bmod m = b \bmod m = 9$, *па се збир* $a \bmod m + b \bmod m = 18$ *може исправно представити са* 32 *бита, пре него што се израчуна коначан резултат* $(a \bmod m + b \bmod m) \bmod m = 8$.

Ипак, треба бити обазрив, јер је могуће да се деси да су и операнди ($a \bmod m$ и $b \bmod m$) и крајњи резултат довољно мали да се могу исправно представити одређеним типом, али да су међурезултати ($a \bmod m + b \bmod m$ и нарочито $a \bmod m \cdot b \bmod m$) превелики да би могли да се исправно представе.

Пример 3.4.4

Ако је $a = b = 10^5 + 1$ *и* $m = 10^6$, *тада је* $a \bmod m = b \bmod m = 10^5 + 1$, *па је* $a \bmod m \cdot a \bmod m = 10^{10} + 2 \cdot 10^5 + 1$, *што се не може исправно записати помоћу* 32 *бита. Коначан резултат је* $(a \bmod m \cdot b \bmod m) \bmod m = 2 \cdot 10^5 + 1$ *и он се може исправно записати помоћу* 32 *бита.*

Зато је у случају великих вредности броја m пожељно међурезултате израчунати у ширем типу (на пример, ако су $a \bmod m$ и $b \bmod m$ представљени са 32 бита, тада је пожељно конвертовати их и помножити као 64-битне бројеве, пре него што се израчунавањем остатка добије коначан резултат који се поново исправно може представити помоћу 32 бита).

Претходна својства нам омогућавају да гарантујемо да ће се уз сва прекорачења међурезултата која долазе током рада са неозначеним бројевима, на крају израчунавања добити резултат који је једнак остатку при дељењу бројем 2^{32} (тј. 2 на број битова употребљених за запис) резултата који би се добио без прекорачења.

Размотримо сада проблем одређивања разлике ненегативних бројева b и a по модулу m. На пример, потребно је одредити вредност $(b - a) \bmod m$ за $a, b \geq 0$. Претпоставимо за почетак да су бројеви a и b мањи од m. Поновимо да не постоји сагласност између различитих програмских језика у рачунању вредности `a % m` када је вредност броја `a` негативна. Наиме, у језику C++ се за вредност остатка при дељењу може добити негативан број: на пример, вредност израза `(2 - 7) % 3` је -2, док се у језику Python као резултат истог овог израза добија позитиван број 1. Често желимо да као резултат рачунања вредности по модулу добијемо ненегативну вредност. Уместо да вршимо анализу случајева, решење је могуће добити израчунавањем вредности израза $(b \bmod m - a \bmod m + m) \bmod m$.

Лема 3.4.3 **[Одузимање по модулу]**

За произвољне бројеве a и b важи:

$$(b-a) \bmod m = (b \bmod m - a \bmod m + m) \bmod m$$

Притом је вредност израза $b \bmod m - a \bmod m + m$ увек ненегативна.

Доказ. Нека је $a = q_a \cdot m + r_a$ и $b = q_b \cdot m + r_b$, за $0 \leq r_a, r_b < m$. Тада је $a \bmod m = r_a$ и $b \bmod m = r_b$. Нека је: $r_b - r_a + m = p \cdot m + r,\ 0 \leq r < m$. Зато је: $(b \bmod m - a \bmod m + m) \bmod m = (r_b - r_a + m) \bmod m = r$. Такође, важи и да је: $b-a = (q_b-q_a)\cdot m+(r_b-r_a) = (q_b-q_a-1)\cdot m+(r_b-r_a+m) = (q_b-q_a-1+p)\cdot m+r$, па је и $(b-a) \bmod m = r$.

Бројеви $a \bmod m = r_a$ и $b \bmod m = r_b$ припадају интервалу $[0, m)$, па њихова разлика припада интервалу $(-m, m)$. Додавањем вредности m, добија се ненегативна вредност (у интервалу $(0, 2m)$). □

Наравно, ако се унапред зна да су аргументи a и b већ сведени по модулу m (тј. важи $0 \leq a < m$ и $0 \leq b < m$), онда је довољно само користити израз $(b - a + m) \bmod m$.

Претходно тврђење нам омогућава да у било ком програмском језику разлику по модулу можемо израчунати једним изразом, тако да се добије ненегативан резултат.

Важи сличан резултат и за степеновање по модулу.

Лема 3.4.4 **[Степеновање по модулу]**

$a^n \bmod m = (a \bmod m)^n \bmod m$

Ова лема се једноставно доказује применом правила за множење бројева по модулу. Пошто и вредност $(a \bmod m)^n$ веома брзо расте и може довести до прекорачења, приликом рачунања степена, свођење по модулу треба вршити приликом сваког множења. Овакво модуларно степеновање се најбоље изводи алгоритмом брзог степеновања (при чему је пожељно у првом позиву свести вредност a по модулу m тј. као први параметар проследити вредност `a % m`).

```
int stepen(int a, int n, int m) {
    if (n == 0)
       return 1;
    if (n % 2 == 0)
        return stepen((a * a) % m, n / 2, m);
```

```
    else
        return (a * stepen(a, n-1, m)) % m;
}
```

Пример 3.4.5

Илуструјмо процедуру рачунања тринаестог степена броја 2 *по модулу* 100*:*

$$\begin{aligned}
\texttt{stepen}(2,13,100) &= \\
(2\cdot\texttt{stepen}(2,12,100)) \bmod 100 &= \\
(2\cdot\texttt{stepen}((2\cdot 2)\bmod 100,6,100)) \bmod 100 &= \\
(2\cdot\texttt{stepen}(4,6,100)) \bmod 100 &= \\
(2\cdot\texttt{stepen}((4\cdot 4)\bmod 100,3,100)) \bmod 100 &= \\
(2\cdot\texttt{stepen}(16,3,100)) \bmod 100 &= \\
(2\cdot((16\cdot\texttt{stepen}(16,2,100))\bmod 100)) \bmod 100 &= \\
(2\cdot((16\cdot\texttt{stepen}((16\cdot 16)\bmod 100,1,100))\bmod 100)) \bmod 100 &= \\
(2\cdot((16\cdot\texttt{stepen}(56,1,100))\bmod 100)) \bmod 100 &= \\
(2\cdot((16\cdot(56\cdot\texttt{stepen}(56,0,100))\bmod 100)\bmod 100)) \bmod 100 &= \\
(2\cdot((16\cdot(56\cdot 1)\bmod 100)\bmod 100)) \bmod 100 &= \\
(2\cdot((16\cdot 56)\bmod 100)\bmod 100) = (2\cdot 96)\bmod 100 &= 92
\end{aligned}$$

Пошто је $2^{13} = 8192$*, резултат је исправно израчунат.*

Аритметичке операције по модулу m се често користе, па ћемо увести наредне скраћене ознаке: са $+_m$ ћемо обележавати сабирање по модулу тј. $a +_m b$ је ознака за $(a + b) \bmod m$, са $\times_m$ множење по модулу тј. $(a \cdot b) \bmod m$ итд.

У криптографији се обично разматрају јако велики бројеви (са по неколико хиљада цифара) који се не могу представити машинским типовима података и за које се ни основне аритметичке операције не могу извршити у константном времену. Подсетимо се да је број цифара броја n једнак $O(\log n)$, одакле следе наредна тврђења.

- Сабирање и одузимање два велика броја (реда величине броја n) може се извршити у времену $O(\log n)$.
- Множење таква два броја се основним алгоритмом може извршити у времену $O(\log^2 n)$ (а може и ефикасније, на пример, применом брзе Фуријеове трансформације описане у поглављу 3.6).

- И дељење се може основним алгоритмом извршити у времену $O(\log^2 n)$ (а постоје и ефикаснији алгоритми).

Дакле, све основне аритметичке операције су довољно ефикасне и када се спроводе над бројевима са по неколико хиљада цифара.

Ствар се не мења значајно ни када се операције извршавају по модулу. Одређивање остатка захтева додатно дељење, међутим, смањује број цифара резултата, па се сложени изрази понекад ефикасније израчунавају када су операције по модулу.

- У поглављу 3.4.1 видећемо да дељење по модулу (тј. одређивање модуларног инверза) захтева спровођење проширеног Еуклидовог алгоритма, па је спорије од множења бројева по модулу. Међутим, проширени Еуклидов алгоритам примењен на бројеве a и n захтева $O(\log(a+n))$ корака који укључују основне аритметичке операције по модулу. Зато су у резултујућој сложености дељења по модулу сви изрази под логаритмом, па се и ова операција може извршити релативно ефикасно (у веома разумном времену, чак и за бројеве који имају по неколико хиљада цифара).
- Модуларно степеновање $a^k \bmod n$ захтева $O(\log k)$ операција дељења и множења по модулу, па се и оно може извршити у разумном времену, чак и када вредности бројева a, k и n имају по неколико хиљада цифара.

Са друге стране, у случају јако великих вредности модула n, неке операције које се ефикасно спроводе над машинским типовима података су јако неефикасне и не могу се практично извршити. Код овако великих бројева постоји огромна разлика између алгоритама сложености $O(\sqrt{n})$ и $O(\log n)$. Ако је n реда величине 10^{1000}, онда алгоритам који извршава $\sqrt{n}$ операција извршава 10^{500} операција, што је неизводиво, док алгоритам који извршава $\log n$ операција извршава само око 1000 операција, што је веома брзо. Сигурност многих криптографских протокола заснована је на тежини извођења операција сложености $O(\sqrt{n})$.

- Операција факторизације броја n захтева $O(\sqrt{n})$ корака ако се користи основни алгоритам описан у поглављу 3.2 и не може се спровести за бројеве који имају велике просте факторе.
- Израчунавање Ојлерове функције (и сличних мултипликативних функција) обично се своди на факторизацију, па је и оно практично неизводиво.
- Операција инверзна модуларном степеновању, тј. операција проналажења вредности k такве да је $a^k \equiv b \pmod n$ назива се *дискретни логаритам* и до сада није пронађен ни један алгоритам који би омогућио њено ефикасно извршавање (за бројеве од по неколико хиљада цифара и најсавременијим рачунарима би требали билиони година за решавање овог проблема).

3.4.1 Модуларне групе

За сваки природан број n остаци при дељењу са n чине скуп $Z_n = \{0, 1, \ldots, n-1\}$. Када се они комбинују операцијом $+_n$ сабирања по модулу n и операцијом $\times_n$ множења по модулу n добијају се вредности које припадају скупу Z_n. Пошто су обе операције асоцијативне и пошто неутрал за сабирање 0 и неутрал за множење 1 припадају скупу Z_n, структуре $(Z_n, +_n)$ и $(Z_n \setminus \{0\}, \times_n)$ су моноиди[9] (за било које n).

Сваки елемент у структури $(Z_n, +_n)$ има инверзни елемент јер је за свако $x \in Z_n$ и елемент $n - x \in Z_n$ и важи $x +_n (n - x) = 0$, па је структура $(Z_n, +_n)$ група[10] (за било коју вредност n). Ову групу називамо *адитивна група целих бројева по модулу* n или *модуларна адитивна група* и обележавамо са $\mathbb{Z}_n^+$.[11] Инверзни елемент у односу на $+_n$ називамо *модуларни адитивни инверз*.

У наредној таблици приказана је таблица сабирања модуларне адитивне групе $\mathbb{Z}_5^+$.

$+_5$	**0**	**1**	**2**	**3**	**4**
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

Да ли, аналогно адитивним, можемо да разматрамо *модуларне мултипликативне групе*? Структура $(Z_n \setminus \{0\}, \times_n)$ не мора бити група, јер не морају сви елементи да имају инверз (који у овом случају називамо *модуларни мултипликативни инверз*). На пример, размотримо таблицу за $n = 6$.

$\times_6$	**1**	**2**	**3**	**4**	**5**
1	1	2	3	4	5
2	2	4	0	2	4
3	3	0	3	0	3
4	4	2	0	4	2
5	5	4	3	2	1

[9] Моноид је алгебарска структура коју чине скуп и бинарна операција која је затворена на том скупу, асоцијативна и има неутрал.

[10] Група је алгебарска структура коју чине скуп и бинарна операција која је затворена на том скупу, асоцијативна, има неутрал и сваки елемент има инверзни елемент.

[11] Алгебарске структуре се обично обележавају подебљаним фонтом (на пример, $\mathbb{Z}$ означава прстен целих бројева), а скупови њихових елемената обичним фонтом (на пример, Z означава скуп целих бројева).

Када се број 2 помножи било којим елементом скупа $Z_6 \setminus \{0\}$, добија се паран број и остатак тог броја при дељењу са 6 може бити само 0, 2 или 4. Зато 2 нема инверзни елемент. Слично се дешава и са елементима 3 и 4. Елемент 1 је сам себи инверзан, а исто важи и за елемент 5 (јер је $5 \cdot 5 = 25 \equiv 1 \pmod 6$).

Са друге стране, размотримо таблицу за $n = 5$.

$\times_5$	**1**	**2**	**3**	**4**
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

Јасно је да сваки елемент има себи инверзни (у свакој врсти и свакој колони се тачно једном јавља вредност 1) и ова структура је група. Инверз броја 1 је увек 1 (јер је $1 \cdot 1 = 1$), инверз броја 2 је 3 (јер је $2 \cdot 3 = 6 \equiv 1 \pmod 5$), инверз броја 3 је 2, а број 4 је сам свој инверз (јер је $4 \cdot 4 = 16 \equiv 1 \pmod 5$). Ова структура је, дакле, група.

Дакле, за неке вредности n, структура $(Z_n \setminus \{0\}, \times_n)$ јесте група, а за неке није.

Приметимо да су у случају $n = 6$ инверз имали бројеви 1 и 5 који су узајамно прости са 6, а у случају $n = 5$ сви бројеви 1, 2, 3 и 4, јер су сви узајамно прости са 5. Размотримо сада општи случај и формулишимо потребан и довољан услов да елемент скупа $Z_n \setminus \{0\}$ има инверз у односу на операцију $\times_n$.

Теорема 3.4.1 **[Постојање модуларног мултипликативног инверза]**

Ако су бројеви a и n узајамно прости природни бројеви (ако је $\operatorname{nzd}(a, n) = 1$*, тј. ако бројеви a и n немају заједничких простих чинилаца), онда постоји јединствен мултипликативни инверз броја a по модулу n, тј. једначина $ax \equiv 1 \pmod n$ има јединствено решење у скупу $Z_n \setminus \{0\}$. Ако бројеви a и n нису узајамно прости, онда модуларни мултипликативни инверз броја a не постоји.*

Доказ. Једначина $ax \equiv 1 \pmod n$ има решење ако и само ако постоји природан број q тако да је $ax = qn + 1$, тј. ако је $ax - qn = 1$. Зато тврђење директно следи из теореме 3.1.4.

Илустрације ради, докажимо тврђење и директно.

Ако би постојао заједнички делилац $d > 1$ бројева a и n он би делио леву страну ове једначине, па би морао да дели и десну тј. број 1, што је немогуће. Дакле, ако бројеви a и n нису узајамно прости, онда a нема инверз.

Претпоставимо сада да су бројеви a и n узајамно прости. Размотримо низ од n бројева $a{\cdot}0, a{\cdot}1, a{\cdot}2, \ldots, a{\cdot}(n{-}1)$. Показаћемо да су остаци свих ових вредности при дељењу са n различити. С обзиром на то да укупно постоји тачно n различитих вредности у скупу Z_n, одатле следи да је $a \cdot x \equiv 1 \pmod{n}$ за тачно једно x из скупа Z_n. Пошто то не може бити $x = 0$, x припада скупу $Z_n \setminus \{0\}$ и он је јединствени модуларни мултипликативни инверз броја a по модулу n.

Да бисмо показали да су остаци различити, претпоставимо супротно, тј. претпоставимо да је $a \cdot x_1 \equiv a \cdot x_2 \pmod{n}$ за две различите вредности x_1 и x_2 из скупа Z_n. Пошто су a и n узајамно прости, на основу првог закона скраћивања конгруенција (лема 3.4.1) важи $x_1 \equiv x_2 \pmod{n}$, тј. $x_1 - x_2$ мора бити неки целобројни умножак броја n. Пошто су x_1 и x_2 два различита ненегативна цела броја мања од n, то је могуће само ако је $x_1 - x_2 = 0$, што је контрадикција јер смо претпоставили да важи $x_1 \neq x_2$

Дакле, све вредности $a \times_n 0, a \times_n 1, \ldots, a \times_n (n-1)$ су међусобно различите и за тачно једну вредност из скупа $\{1, \ldots, n-1\}$ важи да је мултипликативни инверз броја a по модулу n. □

Број x је инверз броја a, али је и број a уједно инверз броја x, па је и x узајамно прост са n.

Ако је p прост број, сви елементи скупа $Z_p \setminus \{0\}$ су узајамно прости са p, па је структура $\mathbb{Z}_p^{\times} = (Z_p \setminus \{0\}, \times_p) = (\{1, 2, \ldots, p-1\}, \times_p)$ група и означавамо је са $\mathbb{Z}_p^{\times}$. Општије, ако посматрамо само скуп Φ_n бројева између 1 и $n-1$ који су узајамно прости са n, структура $(\Phi_n, \times_n)$ је група. Приметимо да за прост број p важи да је скуп $Z_p \setminus \{0\} = \Phi_p$, па је група $\mathbb{Z}_p^{\times}$ само специјални случај групе $\Phi_n^{\times} = (\Phi_n, \times_n)$, када је n прост број. Група $(\Phi_n, \times_n)$ назива се *мултипликативна група целих бројева по модулу* n или *модуларна мултипликативна група* и обележава се са $\Phi_n^{\times}$.

Пример 3.4.6

У наставку је дата табела групе $\Phi_9^{\times}$.

$\times_9$	***1***	***2***	***4***	***5***	***7***	***8***
1	1	2	4	5	7	8
2	2	4	8	1	5	7
4	4	8	7	2	1	5
5	5	1	2	7	8	4
7	7	5	1	8	4	2
8	8	7	5	4	2	1

Операција дељења бројем a по модулу n се своди на множење модуларним мултипликативним инверзом броја a по модулу n. Ово се у неким ситуацијама може употребити за оптимизацију дељења. Претпоставимо да је потребно извршити дељење већег броја неозначених бројева неким бројем k, да су ти бројеви записани са w битова и сви дељиви са k. Уместо да се сви бројеви деле са k могуће је (једном) израчунати модуларни мултипликативни инверз броја k по модулу 2^w, а затим све бројеве помножити тим инверзом. На системима на којима се множење извршава брже од дељења, ово представља оптимизацију. Наравно, да би модуларни мултипликативни инверз постојао, потребно је да су бројеви k и 2^w узајамно прости, што је случај ако и само ако је број k непаран. Сличан поступак можемо употребити и ако је k паран, тако што ћемо k записати у облику $2^p \cdot k'$ за непаран број k', а затим сваки број поделити са 2^p (померањем битова за p места удесно, што је веома брза операција), пре множења са модуларним мултипликативним инверзом броја k' по модулу 2^w (он сигурно постоји јер је k' непаран број).

3.4.2 Ојлерова и мала Фермаова теорема

Кардиналност скупа Φ_n, тј. број бројева између 1 и $n-1$ који су узајамно прости са n израчунава се Ојлеровом функцијом φ, чије је израчунавање описано у поглављу о мултипликативним функцијама 3.3.1. Важи $|\Phi_n| = \varphi(n)$. Ојлерова теорема нам суштински говори о одређеним степенима елемената модуларне мултипликативне групе, али се уопштава и на бројеве који су већи од n.

Теорема 3.4.2 **[Ојлерова теорема]**

Ако су бројеви a и n узајамно прости, тада је $a^{\varphi(n)} \equiv 1 \pmod{n}$ тј. број $a^{\varphi(n)} - 1$ је дељив са n.

Пример 3.4.7

Видели смо да је $\Phi_9 = \{1, 2, 4, 5, 7, 8\}$, па је $\varphi(9) = 6$. Нека је, на пример, $a = 5$: бројеви $a = 5$ и $n = 9$ су узајамно прости. Тада је $5^6 = 15625 = 1736 \cdot 9 + 1$, па је заиста $a^{\varphi(n)} = 5^6 \equiv 1 \pmod{9}$. Тврђење важи и за веће бројеве a, који не припадају скупу Φ_9. На пример, број $a = 14$ је узајамно прост са 9 и важи да је $a^{\varphi(n)} = 14^6 = 7529536 = 836615 \cdot 9 + 1 \equiv 1 \pmod{9}$.

Основни случај Ојлерове теореме се односи на елементе скупа Φ_n, односно бројеве $1 \leq a < n$ узајамно просте са n. Ако претпоставимо да теорема важи за њих, лако је доказати и да важи за све остале бројеве који задовољавају услове теореме. Рецимо да је $a > n$ и да је a узајамно просто са n. Тада се a може поделити са n, тј. постоје бројеви q и $1 \leq r < n$ такви да је $a = qn + r$. Ако је a узајамно прост са n, такав мора

бити и r (јер ако би r имао неки заједнички делилац са n десна страна би била дељива њиме, па би дељив њиме морао да буде и број a на левој страни једнакости). Тада је $a^{\varphi(n)} = (qn + r)^{\varphi(n)}$, међутим, $(qn + r)^{\varphi(n)} \equiv r^{\varphi(n)} \pmod{n}$. Пошто је и $1 \leq r < n$, важи да је $r \in \Phi_n$, па на основу претпоставке да теорема важи у основном случају, тј. да важи за све елементе скупа Φ_n, она важи и за r. Дакле, важи $r^{\varphi(n)} \equiv 1 \pmod{n}$, па важи и $a^{\varphi(n)} \equiv 1 \pmod{n}$.

Дакле, довољно је да се Ојлерова теорема докаже за сваки број $a \in \Phi_n$. Из тога следи и општи случај Ојлерове теореме. Докажимо да за овакве бројеве важи $a^{\varphi(n)} \equiv 1 \pmod{n}$. То је непосредна последица Лагранжеве теореме из теорије група која тврди да ред (број елемената) подгрупе увек дели ред групе из чега резултат директно следи посматрајући групу Φ_n и њену цикличку подргрупу генерисану са a (њу чине елементи $1, a, a^2, \ldots, a^{k-1}$, за неко $k|\varphi(n)$ такво да је $a^k = 1$), међутим, даћемо директан доказ, без позивања на резултате теорије групе.

Доказ. [Ојлерове теореме за $1 \leq a < n$]

Основу овог доказа чини чињеница да се приликом множења по модулу n елемената уређеног скупа $\Phi_n = \{r_1, \ldots, r_{\varphi(n)}\}$ неким бројем a који је узајамно прост са n добија нека пермутација тог скупа (што смо и доказали приликом доказивања критеријума за постојање инверза у теореми 3.4.1). Заиста, на основу првог закона скраћивања (лема 3.4.1), $ar_i \equiv ar_j \pmod{n}$ важи ако и само ако је $r_i = r_j$ тј. $i = j$. Зато се множењем различитих остатака из Φ_n бројем a добијају различити остаци, па слика скупа Φ_n при пресликавању $r \mapsto ar$ мора бити скуп Φ_n. Зато је

$$\prod_{i=1}^{\varphi(n)} r_i \equiv \prod_{i=1}^{\varphi(n)} ar_i = a^{\varphi(n)} \cdot \prod_{i=1}^{\varphi(n)} r_i \pmod{n}.$$

Тражено тврђење се добија на основу првог закона скраћивања (лема 3.4.1), скраћивањем фактора $\prod_{i=1}^{\varphi(n)} r_i$, који је узајамно прост са n. □

Специјални случај Ојлерове теореме, када је n прост број, је *Мала Фермаова теорема* (док Ојлерова теорема говори о групама $\Phi_n^\times$, Мала Фермаова говори само о групама $\mathbb{Z}_p^\times$).

Теорема 3.4.3 **[Мала Фермаова теорема]**

Ако је p прост број и a позитиван природан број који није дељив бројем p, тада је $a^{p-1} \equiv 1 \pmod{p}$.

Ако a и n нису узајамно прости, тада једнакост $a^{\varphi(n)} \equiv 1 \pmod{n}$ не мора да важи. На пример, ако је a дељиво са p тада је $a^0 = 1$, међутим важи $a^1 \equiv 0 \pmod{p}$, $a^2 \equiv 0 \pmod{p}$, па и $a^{p-1} \equiv 0 \pmod{p}$, јер су сви степени броја a дељиви са p, па $a^{p-1} \not\equiv 1 \pmod{n}$ и једнакост $a^{p-1} \equiv 1 \pmod{p}$ не мора да важи. Ипак, наредна, модификована, формулација Мале Фермаове теореме исправно обухвата и овај случај.

Теорема 3.4.4 **[Мала Фермаова теорема – општи облик]**

Ако је p прост број и a природан број, тада је $a^p \equiv a \pmod{p}$.

Доказ. Ако a није дељиво са p, тврђење је еквивалентно полазној формулацији (када се обе стране помноже са a), а ако a јесте дељиво са p, онда је $a \equiv 0 \pmod{p}$ и $a^p \equiv 0 \pmod{p}$, па је $a^p \equiv a \pmod{p}$. □

Пошто за прост број p важи $\varphi(p) = p - 1$, мала Фермаова теорема је директна последица Ојлерове теореме. Дакле, њена коректност се обезбеђује било којим исправним доказом основног случаја Ојлерове теореме, међутим, алтернативни доказ који приказујемо има за циљ да поред оправдања да и дубље објашњење теореме.

Доказ. [Мале Фермаове теореме (елементаран)]

Један веома леп, а крајње елементаран начин да се докаже Мала Фермаова теорема је да се размотре све ниске дужине p у којима се јављају слова из азбуке која има a симбола. Таквих ниски има a^p. На пример, ако је $p = 3$ и $a = 2$, и ако азбука садржи слова `X` и `Y`, то су ниске `XXX`, `XXY`, `XYX`, `XYY`, `YXX`, `YXY`, `YYX` и `YYY`. Замислимо сада да су слова сваке ниске написана на кругу. Тиме све дате ниске делимо у одређене групе тако да две ниске припадају истој групи ако и само ако се добијају читањем слова са истог круга. У једној групи је само `XXX`, у другој су `XXY`, `XYX` и `YXX`, у трећој групи су `XYY`, `YYX` и `YXY`, док је у последњој, четвртој групи само `YYY`.

```
XXX   XXY   XYY   YYY
      XYX   YXY
      YXX   YYX
```

У општем случају, постојаће a група које садрже само један елемент и у њима ће бити ниске које садрже p понављања једног истог симбола из азбуке. Све остале групе имаће по p елемената. Наиме, ако би се запис дат на неком кругу могао прочитати на два иста начина кренувши од два различита слова, тада би тај запис морао бити периодичан и период би морао да дели дужину ниске. Међутим, пошто је p прост број, његови једини делиоци су 1 и p, тако да свака група има или 1 или p различитих елемената. Дакле, важи да је $a^p = x \cdot p + a$, где је x број група са по p елемената, да је $a^p - a = x \cdot p$, тј. да је

$a(a^{p-1}-1) = x \cdot p$. Дакле, $a^p - a$ је дељиво са p, што је тврђење најопштијег облика Фермаове теореме. Додатно, ако a није дељиво са p, тада a не може да има заједничких простих фактора са p (јер је p прост), па следи да $a^{p-1} - 1$ мора да буде дељиво са p, што је тврђење недегенерисаног случаја Мале Фермаове теореме. □

Малу Фермаову теорему је једноставно доказати и индукцијом (Ојлер је први објавио доказ ове теореме и тај је доказ био индукцијом).

Примене Ојлерове и Мале Фермаове теореме су разне: испитивање да ли је број прост, израчунавање модуларног мултипликативног инверза, RSA криптографија и слично. Прикажимо сада примену Мале Фермаове теореме на испитивање да ли је број прост, а остале примене ових теорема ће бити приказане у наредним поглављима (на пример, у поглављу 3.5 биће приказана примена Ојлерове теореме у области криптографије).

3.4.2.1 Фермаов тест да ли је број прост

Једна примена Мале Фермаове теореме је за тестирање да ли је дати број n прост или не (енгл. Fermat primality test). Мала Фермаова теорема даје потребан услов да би број био прост. Наиме, за дати број n можемо изабрати произвољан број a који није дељив са n и израчунати вредност $a^{n-1} \bmod n$. Уколико је резултат различит од 1, број n је сигурно сложен. Ипак, теорема не даје и довољан услов да би број био прост, јер постоје и сложени бројеви n такви да за неке вредности a које су узајамно просте са n важи да је $a^{n-1} \bmod n$ једнако 1. Ипак, ако је $a^{n-1} \bmod n = 1$, показује се да је мало вероватно да број n није прост. Посебно, ако се за велики број различитих вредности за a које су узајамно просте са n покаже да је вредност израза $a^{n-1} \bmod n$ једнака 1, прилично је вероватно да је број n прост. Нажалост, ни испитивање свих вредности a које су узајамно просте са n нам не даје гаранцију, јер постоје сложени бројеви n такви да за све бројеве a који су узајамно прости са n важи $a^{n-1} \equiv 1 (\bmod n)$. Такви бројеви се зову *Кармајклови бројеви* (енгл. Carmichael numbers) и релативно су ретки. Најмањи такав број је 561.

3.4.3 Цикличност модуларних мултипликативних група

На основу мале Фермаове теореме, за свако $1 \leq a < p$ важи да је $a^{p-1} \equiv 1 \pmod p$. Природно се поставља питање шта се дешава са степенима броја a када је изложилац мањи од $p-1$ тј. какав је скуп $\{a^0 \bmod p, a^1 \bmod p, \ldots, a^{p-2} \bmod p\}$. Може се доказати следећа лема.

Лема 3.4.5 **[Модуларне мултипликативне групе су цикличке]**

Групе $(\mathbb{Z}_p, \times_p)$ су цикличке, тј. за сваки прост број p постоји број $a \in Z_p \setminus \{0\}$ такав да је $\{a^0 \bmod p, a^1 \bmod p, \ldots, a^{p-2} \bmod p\} = \{1, 2, \ldots, p-1\} = Z_p \setminus \{0\}$.

Дакле у групи $\mathbb{Z}_p^\times$ увек постоји елемент a такав да се сви елементи групе могу изразити као његови степени (по модулу p). Такав елемент се назива *генератор групе* или *примитивни корен по модулу* p. Показује са да у групи $\mathbb{Z}_p^\times$ постоји $\varphi(p-1)$ генератора. Проналажење генератора је нетривијалан проблем и размотрићемо га у поглављу 3.6.4.

Пример 3.4.8

Размотримо степене елемената у групи $\mathbb{Z}_7^\times$.

a	a^0	a^1	a^2	a^3	a^4	a^5	a^6
1	1	1	1	1	1	1	1
2	1	2	4	1	2	4	1
3	1	3	2	6	4	5	1
4	1	4	2	1	4	2	1
5	1	5	4	6	2	3	1
6	1	6	1	6	1	6	1

Видимо да су генератори групе $a = 3$ *и* $a = 5$ *и њих је* $\varphi(7-1) = \varphi(6) = 2$.

Групе $\Phi_n^\times$, када n није прост, не морају бити цикличке.

3.4.4 Израчунавање модуларног мултипликативног инверза

У поглављу 3.4.1 о модуларним групама смо видели да је *мултипликативни инверз броја* a *по модулу* n (енгл. modular multiplicative inverse) број x за који важи $a \cdot x \equiv 1 \pmod{n}$.

Пример 3.4.9

Мултипликативни инверз броја 5 *по модулу* 8 *је* 5 *јер важи* $5 \cdot 5 \equiv 1 \pmod{8}$.

Најчешће се као мултипликативни инверз броја a по модулу n разматра вредност из скупа $Z_n = \{1, 2, \ldots, n-1\}$.

У теореми 3.4.1 је доказано да инверз по модулу n броја a постоји ако и само ако су a и n узајамно прости и да је у том случају јединствен по модулу n. Размотримо сада како га је могуће израчунати.

Проблем

Ако су дата два узајамно проста позитивна цела броја a *и* n, *одредити мултипликативни инверз броја* a *по модулу* n, *тј. број* x *такав да је* $ax \equiv 1 \pmod{n}$.

3.4.4.1 Алгоритам грубе силе

При наивном приступу решавању овог проблема се за x разматрају редом сви бројеви x од 1 до $n-1$ (елементи скупа Z_n) и проверава се да ли је остатак при дељењу броја $a \cdot x$ са n једнак 1. Сложеност овог алгоритма је $O(n)$.

3.4.4.2 Алгоритам заснован на проширеном Еуклидовом алгоритму

На основу теореме 3.4.1 модуларни инверз броја a по модулу n постоји ако и само ако су a и n узајамно прости. Доказ наредне теореме нам говори да се, када постоји, он може одредити проширеним Еуклидовим алгоритмом.

Лема 3.4.6

Ако су a и n узајамно прости, тада постоји јединствен број x из скупа $Z_n = \{1, \ldots, n-1\}$ такав да је $a \cdot x \equiv 1 \pmod{n}$.

Доказ. Да би број x био инверз броја a, треба да важи $a \cdot x \equiv 1 \pmod{n}$ тј. $n|(a \cdot x - 1)$. Зато мора да постоји број y такав да је $a \cdot x - 1 = y \cdot n$ тј.

$$x \cdot a - y \cdot n = 1.$$

Ми решавамо ову Диофантову једначину по непознатим целобројним коефицијентима x и y и тражимо оно решење за које је $1 \leq x < n$. Пошто су a и n узајамно прости, на основу Безуове теореме 3.1.3, постоји број x који ово задовољава. На основу леме 3.1.1, постоје тачно два решења за које важи $|x| < |n/d|$ и $|y| < |a/d|$ (једно у ком је x позитивно и једно у ком је x негативно), па, пошто је $d = 1$, постоји тачно једно решење $x \in Z_n$. □

Дакле, проналажење модуларног инверза се може свести на решавање једначине $ax = yn + 1$ тј. $ax - ny = 1$, а на основу теореме 3.1.4, знамо да једначина $ax + by = c$ има целобројна решења ако и само ако $d = \text{nzd}(a, b)$ дели број c. Дакле, за $c = 1$ једначина $ax - ny = 1$ има целобројна решења (x, y) ако и само ако су a и n узајмно прости.

За ефикасно израчунавање модуларног инверза можемо искористити проширени Еуклидов алгоритам. Потребно је да пронађемо коефицијенте x и y такве да је $x \cdot a - y \cdot n = 1$, што можемо урадити применом проширеног Еуклидовог алгоритма на два дата, узајамно проста броја a и n (он ће нам, додуше, вратити и вредност $-y$, али тај коефицијент нас ионако не занима). С обзиром на то да добијена вредност коефицијента x може бити и негативна и већа од n, ако желимо да x припада Z_n, тј. да задовољава услов $0 \leq x < n$, онда уместо x као модуларни мултипликативни инверз у програму враћамо вредност $(x \bmod n + n) \bmod n$.

Дакле, одређивање модуларног мултипликативног инверза броја можемо свести на проширени Еуклидов алгоритам.

```
// racunanje multiplikativnog inverza broja a po modulu m
// koriscenjem prosirenog Euklidovog algoritma
int modInverz(int a, int n) {
  int x, y;
  // određujemo x i y tako da vazi a*x + n*y = d
  int d = nzdProsireni(a, n, x, y);

  // ako a i n nisu uzajamno prosti, onda ne postoji inverz
  if (d != 1)
    cout << "Modularni multiplikativni inverz ne postoji" << endl;
  else{
    // obezbedjujemo da je inverz iz skupa [0, n)
    int inverz = (x % n + n) % n;
    return inverz;
   }
}
```

Приметимо да за рачунање модуларног мултипликативног инверза броја вредност коефицијента y није од интереса, те се може конструисати алгоритам који не позива експлицитно проширени Еуклидов алгоритам, већ на исти начин као код проширеног Еуклидовог алгоритма итеративно рачуна само вредност коефицијента x.

```
// funkcija racuna x -- multiplikativni inverz broja a po modulu n,
// koriscenjem prosirenog Euklidovog algoritma
// funkcija vraca da li je uspela da izracuna broj x
bool modInverz(int a, int n, int& x) {
  // pocetne vrednosti za r su n i a
  int r_preth = n;
  int r_tek = a;
  // pocetne vrednosti za x su 0 i 1
  int x_preth = 0;
  int x_tek = 1;

  while (r_tek > 0) {
    int q = r_preth / r_tek;
    int pom;
```

```
    // azuriramo tekucu i prethodnu vrednost niza r
    pom = r_preth;
    r_preth = r_tek;
    r_tek = pom - q * r_tek;

    // azuriramo tekucu i prethodnu vrednost niza x
    pom = x_preth;
    x_preth = x_tek;
    x_tek = pom - q * x_tek;
  }

  // obezbedjujemo da je inverz iz skupa [0,m)
  x = (x_preth % n + n) % n;

  // vracamo true ako je nzd(a, n) = 1, inace false
  return r == 1;
}
```

Сложеност претходног алгоритма за израчунавање мултипликативног инверза броја a по модулу n једнака је сложености проширеног Еуклидовог алгоритма за бројеве a и n и износи $O(\log(a+n))$, односно када је a реда $O(n)$ једнака је $O(\log n)$ (што је веома ефикасно).

3.4.4.3 Алгоритам заснован на Ојлеровој и Малој Фермаовој теореми

Још један ефикасан начин за рачунање модуларног мултипликативног инверза броја је коришћењем Ојлерове теореме. Подсетимо се њеног тврђења: ако су a и n узајамно прости бројеви онда важи: $a^{\varphi(n)} \equiv 1 \pmod{n}$. Приметимо да је услов да су бројеви a и n узајамно прости такође услов и да би постојао мултипликативни инверз броја a по модулу n. Из претходне једначине следи:

$$a \cdot a^{\varphi(n)-1} \equiv 1 \pmod{n}. \tag{3.5}$$

Приметимо да за произвољну вредност n важи $\varphi(n) \geq 1$, односно $\varphi(n) - 1 \geq 0$, те је $a^{\varphi(n)-1}$ позитивна целобројна вредност. Дакле, за $x = a^{\varphi(n)-1} \bmod n$ важи $a \cdot x \equiv 1 \pmod{n}$, па је $a^{\varphi(n)-1}$ мултипликативни инверз броја a по модулу n. Додатно, $a^{\varphi(n)-1}$ може бити веће од n те је као резултат потребно вратити вредност $x \bmod n$ (што је унапред осигурано ако се током степеновања свођење врши након сваког множења).

На овај начин смо проблем рачунања модуларног мултипликативног инверза броја a свели на рачунање Ојлерове функције броја n што се даље своди на факторизацију

броја n, која се основним алгоритмом описаним у поглављу 3.2 врши у сложености $O(\sqrt{n})$, што је неприхватљиво за бројеве од рецимо 100 цифара.

Пример 3.4.10

Вратимо се на пример са почетка овог поглавља и пронађимо инверз броја $a = 5$ *по модулу* $n = 8$. *Важи* $\varphi(n) = \varphi(8) = 4$ *и стога важи* $5^4 \equiv 1 \pmod 8$. *Одатле добијамо* $5 \cdot 5^3 \equiv 1 \pmod 8$, *тј. број* $5^3 \bmod 8 = 125 \bmod 8 = 5$ *је мултипликативни инверз броја* 5 *по модулу* 8. *Заиста,* $5 \cdot 5 = 25 \equiv 1 \pmod 8$.

Уколико је пак број n прост, тврђење Ојлерове теореме се своди на малу Фермаову теорему: $a^{n-1} \equiv 1 \pmod n$, одакле следи:

$$a \cdot a^{n-2} \equiv 1 \pmod n. \tag{3.6}$$

Пошто је број n прост, важи $n \geq 2$, односно $n-2 \geq 0$, те је вредност a^{n-2} целобројна и позитивна. Стога важи да је $x = a^{n-2} \bmod n$ мултипликативни инверз броја a по модулу n. Приметимо да је овде ствар једноставнија, јер је вредност Ојлерове функције броја n унапред позната (и једнака $n-1$).

Пример 3.4.11

Ако је $n = 7$ *и* $a = 3$, *пошто је број* m *прост, мултипликативни инверз броја* 3 *по модулу* 7 *можемо наћи по формули* $a^{n-2} \bmod n = 3^5 \bmod 7 = 243 \bmod 7 = 5$ *и заиста важи* $3 \cdot 5 \equiv 1 \pmod 7$.

Из конгруенција (3.5) и (3.6) можемо једноставно одредити мултипликативни инверз броја a по модулу n. Једино што преостаје јесте што ефикасније израчунати одговарајући степен броја a, што се може извести коришћењем ефикасног алгоритма за модуларно степеновање броја, који је сложености $O(\log k)$, где је k вредност експонента.

Напоменимо да је у ситуацији када n није прост број, потребно израчунати вредност Ојлерове функције броја n што укључује факторизацију броја n и може бити тежак проблем. Међутим, у ситуацији када је број n прост, али такође и када број n није прост али је позната његова факторизација, сложеност овог алгоритма је $O(\log n)$.

У наставку је дат алгоритам за рачунање мултипликативног инверза броја a по модулу n за прост број n.

```
// funkcija za mnozenje po modulu n
int puta_mod(int a, int b, int n) {
  return ((a % n) * (b % n)) % n;
}

// funkcija za brzo stepenovanje po modulu n
int stepen_mod(int a, int b, int n) {
    // a^0 = 1
    if (b == 0)
        return 1;
    // racunamo a^(b/2) mod n
    int rez = stepen_mod(a, b / 2, n);
    // racunamo rez*rez mod n
    rez = puta_mod(rez, rez, n);
    if (b % 2)
      // racunamo rez*a mod n
      return puta_mod(rez, a, n);
    else
        return rez;
}

// racunanje multiplikativnog inverza broja a po modulu n
// koriscenjem Male Fermaove teoreme
int modInverz(int a, int n) {
  return stepen_mod(a, n - 2, n);
}
```

3.4.5 Кинеска теорема о остацима

Према једној кинеској легенди, генерал Хан Хин је, да би избегао да шпијуни сазнају са коликом је војском кренуо у бој, свом куриру давао само информацију о остатку броја војника у правоугаоној формацији. Његове поруке су биле у наредној форми: „Када се војници организују у редове од по 9, преостаје њих 4; ако су у редовима од по 11, нико не преостаје; ако су у редовима од по 13, само један преостаје, а ако су у редовима од по 19, преостаје њих тројица". Генерал је такође свом куриру пренео да је број војника друго најмање решење овог проблема. Питање које се поставља је коликим бројем војника је генерал заповедао. Овај проблем одговара тзв. *Кинеској теореми о остацима*, једном од стандардних проблема елементарне теорије бројева. Први познат исказ теореме се приписује кинеском математичару Сунзију Суанцингу између 3. и 5. века нове ере. Формулишимо овај проблем у стандардним терминима.

Проблем

Дата су два низа бројева r: $r_0, r_1, \ldots, r_{n-1}$ и m: $m_0, m_1, \ldots, m_{n-1}$ при чему за сваки пар бројева низа m важи да су узајамно прости. Одредити најмањи позитиван број x за који важи:

$$\begin{aligned} x \bmod m_0 &= r_0 \\ x \bmod m_1 &= r_1 \\ &\ldots \\ x \bmod m_{n-1} &= r_{n-1} \end{aligned}$$

Тражи се, дакле, да се реши следећи систем конгруенција по модулу.

$$\begin{aligned} &x \equiv r_0 \pmod{m_0} \\ &x \equiv r_1 \pmod{m_1} \\ &\ldots \\ &x \equiv r_{n-1} \pmod{m_{n-1}} \end{aligned} \tag{3.7}$$

Теорема 3.4.5 **[Кинеска теорема о остацима]**

Постоји број x који задовољава систем конгруенција (3.7) и он је јединствен по модулу $M = m_0 \cdot m_1 \cdot \ldots \cdot m_{n-1}$.

3.4.5.1 Груба сила

У наивном приступу решавању овог проблема разматрају се редом бројеви од 1 навише по скупу природних бројева и проверава се да ли текући број задовољава дате услове. Пошто увек постоји број који задовољава овај скуп једначина, на овај начин бисмо дошли до решења, али у броју корака пропорционалном траженој вредности x, што је у најгорем случају $O(M)$.

3.4.5.2 Алгоритам заснован на просејавању

Постоји један поступак који није превише математички захтеван, али који није оптималан у смислу ефикасности (но, много је бољи од грубе силе и често успева да реши проблем унутар задатих временских ограничења). Приступ је заснован на паметној претрази. Ако број x са бројем m_0 даје остатак r_0 онда је он сигурно један од чланова аритметичког низа $r_0, r_0 + m_0, r_0 + 2m_0, \ldots$ У петљи редом проверавамо ове бројеве све док не наиђемо на први елемент који при дељењу са m_1 даје остатак r_1. Када такав број r_{01} нађемо (он ће бити најмањи позитиван број са особином да при дељењу са m_0 и m_1 даје редом остатке r_0 и r_1), знамо да ће сваки наредни број који задовољава то својство бити члан аритметичког низа $r_{01}, r_{01} + m_0 \cdot m_1, r_{01} + 2 \cdot m_0 \cdot m_1, \ldots$ Редом

претражујемо елементе овог низа док не наиђемо на елемент r_{012} који при дељењу са m_2 даје остатак r_2. Поступак се наставља тако што се након проналажења r_{012} посматрају бројеви r_{012}, $r_{012} + m_0 \cdot m_1 \cdot m_2$, $r_{012} + 2 \cdot m_0 \cdot m_1 \cdot m_2$ и тако даље, све док се не нађе број који задовољава сва дата ограничења.

Пример 3.4.12

Прикажимо претходни алгоритам на једном примеру. Нека је $(r_0, m_0) = (2, 3)$, $(r_1, m_1) = (3, 5)$ *и* $(r_2, m_2) = (2, 7)$. *Важи да је* $M = m_0 \cdot m_1 \cdot m_2 = 105$.

Посматрамо прво аритметички низ $r_0 + k \cdot m_0$ *чији су чланови* 2, 5, 8, 11, *итд. и у њему тражимо први број који при дељењу са* $m_1 = 5$ *даје тражени остатак* $r_1 = 3$. *Први такав број је* $r_{01} = 8$.

Сада посматрамо низ $r_{01} + k \cdot (m_0 \cdot m_1)$, *тј. низ* 8, 23, 38 *итд. и у њему тражимо први елемент који при дељењу са* $m_2 = 7$ *даје тражени остатак* 2. *То је број* $r_{012} = 23$, *који је коначан резултат.*

3.4.5.3 Алгоритам заснован на Лагранжевом приступу

Докажимо тврђење кинеске теореме о остацима. Доказ је конструктиван и даје општи алгоритам за конструисање траженог броја x.

Доказ. Основна идеја је иста као и у конструкцији Лагранжовог интерполационог полинома. Претпоставимо да умемо да одредимо целе бројеве $w_0, w_1, \ldots, w_{n-1}$ такве да за свако i важи да w_i при дељењу са m_i (датим у поставци проблема) даје остатак 1, док при дељењу са свим другим бројевима m_j, $j \neq i$ даје остатак 0, тј. да имамо низ бројева w_i који задовољава услове дате у наредној табели.

	$\bmod m_0$	$\bmod m_1$	…	$\bmod m_{n-1}$
w_0	1	0	…	0
w_1	0	1	…	0
…	…	…	…	…
w_{n-1}	0	0	…	1

Множењем бројева w_i са r_i (датим у поставци проблема) добијају се бројеви при дељењу са m_j дају остатак 0 за $j \neq i$ и r_i за $j = i$. Зато се једно x које задовољава дати систем конгруенција може конструисати као $x = r_0 \cdot w_0 + \ldots + r_{n-1} \cdot w_{n-1}$. Нека је $M = m_0 \cdot m_1 \cdot \ldots \cdot m_{n-1}$. Најмање решење x које задовољава систем конгруенција се добија као $x = (r_0 \cdot w_0 + \ldots + r_{n-1} \cdot w_{n-1}) \bmod M$ и доказаћемо да је то једино решење мање од броја M.

Показаћемо сада на који начин се могу конструисати бројеви w_i. Пошто број w_i мора да буде дељив свим бројевима m_j за $0 \leq j < n$ и $j \neq i$, а они су узајамно прости, он мора бити неки умножак њиховог производа. Уведимо ознаке за те производе. Нека је:

$$z_0 = \frac{M}{m_0},\ z_1 = \frac{M}{m_1},\ \ldots,\ z_{n-1} = \frac{M}{m_{n-1}}$$

Дакле, z_i је производ свих бројева m_j за $0 \leq j < n$ и $j \neq i$. За свако $0 \leq i < n$ важи:

1. $z_i \equiv 0 \pmod{m_j}$ за $j \neq i$;
2. z_i и m_i су узајамно прости бројеви.
3. z_i при дељењу са m_i не даје остатак 1, него $z_i \bmod m_i$ (који може, али не мора бити једнак 1).

Прва два услова нам одговарају, али последњи не. Да бисмо од остатка $z_i \bmod m_i$ добили жељени остатак 1, добијени производ z_i треба некако поделити бројем $z_i \bmod m_i$. Да радимо са реалним бројевима, вредност 1 бисмо добили реалним дељењем са $z_i \bmod m_i$, тј. множењем са коефицијентом $1/(z_i \bmod m_i)$. У модуларној аритметици дељење се изводи множењем инверзним елементом. Ствар зато можемо поправити тако што z_i помножимо модуларним инверзом броја $z_i \bmod m_i$ по модулу m_i (а који је исти као модуларни инверз броја z_i по модулу m_i). Наиме, множење било којим бројем не може нарушити дељивост и сви остаци који су били нула остаће нула. Број z_i тј. $z_i \bmod m_i$ има модуларни мултипликативни инверз по модулу m_i (јер су z_i и m_i узајамно прости), па његово множење тим инверзом даје вредност 1 по модулу m_i. Дакле, коефицијент којим множимо треба да буде једнак модуларном мултипликативном инверзу броја z_i тј. $z_i \bmod m_i$ по модулу m_i. Обележимо тај инверз са y_i. Важи:

$$\begin{aligned}
&y_0 \equiv z_0^{-1} \pmod{m_0}\\
&y_1 \equiv z_1^{-1} \pmod{m_1}\\
&\ldots\\
&y_{n-1} \equiv z_{n-1}^{-1} \pmod{m_{n-1}}
\end{aligned}$$

Ове инверзе можемо одредити, на пример, проширеним Еуклидовим алгоритмом. Дакле, ако уместо z_i посматрамо бројеве $y_i \cdot z_i$ добијамо бројеве за које важе оба жељена својства:

1. $y_i \cdot z_i \equiv 0 \pmod{m_j}$ за $j \neq i$;
2. $y_i \cdot z_i \equiv 1 \pmod{m_i}$.

Ова својства нам омогућавају да на следећи начин дефинишемо бројеве $w_0, w_1, \ldots, w_{n-1}$ који задовољавају захтеве из претходне табеле (рачунањем остатка са M својства остају да важе, а вредности бројева се потенцијално смањују што олакшава рачун):

$$\begin{aligned} w_0 &= (y_0 \cdot z_0) \bmod M \\ w_1 &= (y_1 \cdot z_1) \bmod M \\ &\ldots \\ w_{n-1} &= (y_{n-1} \cdot z_{n-1}) \bmod M \end{aligned}$$

Непосредно се проверава да број

$$x = (r_0 \cdot w_0 + \ldots + r_{n-1} \cdot w_{n-1}) \bmod M$$

представља решење система (3.7) тј. даје остатак r_i при дељењу са m_i. Наиме,

$$\begin{aligned} x \bmod m_i &= ((r_0 y_0 z_0 + r_1 y_1 z_1 + \ldots + r_{n-1} y_{n-1} z_{n-1}) \bmod M) \bmod m_i \\ &= (r_0 y_0 z_0 + r_1 y_1 z_1 + \ldots + r_{n-1} y_{n-1} z_{n-1}) \bmod m_i \\ &= r_i y_i z_i \bmod m_i \\ &= r_i \end{aligned}$$

Друга једнакост у овом низу важи на основу тога што је $M \bmod m_i = 0$, трећа на основу тога што је $y_j \cdot z_j \equiv 0 \pmod{m_i}$ за $j \neq i$, а четврта на основу својства $y_i \cdot z_i \equiv 1 \pmod{m_i}$ и чињенице да је $0 \leq r_i < m_i$.

Покажимо да сва решења система (3.7) дају исти остатак при дељењу са M. Размотримо два решења x_1 и x_2. Важи:

$$m_0 \mid (x_1 - x_2),\ m_1 \mid (x_1 - x_2), \ldots,\ m_{n-1} \mid (x_1 - x_2).$$

С обзиром на то да су сви $m_0, m_1, \ldots, m_{n-1}$ узајамно прости, важи и да

$$m_0 \cdot m_1 \cdot \ldots \cdot m_{n-1} \mid (x_1 - x_2).$$

Дакле, важи да је $x_1 \equiv x_2 \pmod{M}$, тј. решење је јединствено по модулу $M = m_0 \cdot m_1 \cdot \ldots \cdot m_{n-1}$.

Пошто у интервалу $[0, M)$ не постоји број различит од $x = (r_0 \cdot w_0 + ... + r_{n-1} \cdot w_{n-1}) \bmod M$ који је конгруентан са x по модулу M, x је једино позитивно решење система (3.7) мање од M. Тиме је доказана Кинеска теорема о остацима. □

На овој идеји заснива се и ефикаснији алгоритам за решавање полазног проблема. Нека су бројеви m_i и r_i представљени машинским типовима и нека је број услова (број n) мали, тако да су израчунавање производа M и производа z_i операције сложености $O(1)$. Најзахтевнија операција је проналажење модуларних инверза бројева z_i, и она се извршава у времену $O(\log M)$ (јер је $z_i \leq M$), што је и сложеност целог алгоритма.

Пример 3.4.13

Вратимо се на полазни пример и израчунајмо коликом је војском генерал Хан Хин руководио. Ако је x број војника, задате услове можемо да запишемо на следећи начин:

$$\begin{aligned} x &\equiv 4 \pmod 9 \\ x &\equiv 0 \pmod{11} \\ x &\equiv 1 \pmod{13} \\ x &\equiv 3 \pmod{19} \end{aligned}$$

Означимо са $M = 9 \cdot 11 \cdot 13 \cdot 19 = 24453$ и са: $z_0 = \frac{M}{9} = 2717$, $z_1 = \frac{M}{11} = 2223$, $z_2 = \frac{M}{13} = 1881$, $z_3 = \frac{M}{19} = 1287$.

Израчунајмо модуларне мултипликативне инверзе ових бројева:

$$\begin{aligned} y_0 &= z_0^{-1} = 2717^{-1} \bmod 9 = 8^{-1} \bmod 9 = 8 \\ y_1 &= z_1^{-1} = 2223^{-1} \bmod 11 = 1^{-1} \bmod 11 = 1 \\ y_2 &= z_2^{-1} = 1881^{-1} \bmod 13 = 9^{-1} \bmod 13 = 3 \\ y_3 &= z_3^{-1} = 1287^{-1} \bmod 19 = 14^{-1} \bmod 19 = 15 \end{aligned}$$

На основу ових вредности, можемо израчунати потребне бројеве w_i:

$$\begin{aligned} w_0 &= y_0 \cdot z_0 \bmod M = 8 \cdot 2717 \bmod 24453 = 21736 \\ w_1 &= y_1 \cdot z_1 \bmod M = 1 \cdot 2223 \bmod 24453 = 2223 \\ w_2 &= y_2 \cdot z_2 \bmod M = 3 \cdot 1881 \bmod 24453 = 5643 \\ w_3 &= y_3 \cdot z_3 \bmod M = 15 \cdot 1287 \bmod 24453 = 19305 \end{aligned}$$

Најзад добијамо:

$$\begin{aligned} x &= (r_0 \cdot w_0 + ... + r_{n-1} \cdot w_{n-1}) \bmod M \\ &= (4 \cdot 21736 + 0 \cdot 2223 + 1 \cdot 5643 + 3 \cdot 19305) \bmod 24453 \\ &= 3784 \end{aligned}$$

С обзиром на то да је речено да је број војника друго најмање решење овог проблема, укупан број војника једнак је $x = 3784 + 24453 = 28237$

Размотримо сада имплементацију решења полазног проблема заснованог на тврђењу Кинеске теореме о остацима. Засебно су имплементиране сабирања и множења по модулу (овим се остатак при дељењу рачуна и чешће него што је то заиста неопходно, али је добијени програм прегледнији).

```
// na osnovu Kineske teoreme o ostacima se izracunava rezultat tako da
// za sve elemente nizova m i a vazi da je rezultat mod m[i] = r[i]
// funkcija vraca da li je rezultat bilo moguce naci
bool kto(const vector<int>& m, const vector<int>& r, long long& rezultat) {
  // broj zadatih uslova (m.size() == r.size())
  int n = m.size();
  // racunamo proizvod svih modula
  long long M = 1;
  for (int i = 0; i < n; i++)
    M *= m[i];

  rezultat = 0;
  for (int i = 0; i < n; i++) {
    // racunamo zi
    long long zi = M / m[i];
    long long yi;
    // racunamo yi kao inverz broja zi po modulu m[i]
    if (!modInverz(zi, m[i], yi))
      return false;
    // na rezultat dodajemo sabirak ri*yi*zi mod M
    // funkcija pm racuna proizvod, a zm zbir po modulu M
    rezultat = zm(rezultat, pm(r[i], yi, zi, M), M);
  }
```

```
    return true;
}
```

3.4.5.4 Алгоритам заснован на Безуовој теореми

Поред општег поступка за решавање проблема за k остатака, у случају само два остатка постоји сличан поступак, који се заснива на томе да умемо да решимо систем од две једначине (две конгруенције), применом Безуове теореме.

За узајамно просте бројеве m_1 и m_2, на основу Безуове теореме постоје бројеви y_1 и y_2 такви да је $y_1 \cdot m_1 + y_2 \cdot m_2 = 1$. Може се лако показати да коефицијент y_1 представља модуларни инверз броја m_1 по модулу m_2, док коефицијент y_2 представља модуларни инверз броја m_2 по модулу m_1. Зато број $x_{12} = r_1 \cdot y_2 \cdot m_2 + r_2 \cdot y_1 \cdot m_1$ тј. његов остатак по модулу $m_1 \cdot m_2$, евентуално увећан за $m_1 \cdot m_2$ ако је негативан, при дељењу са m_1 даје остатак r_1, а при дељењу са m_2 даје остатак r_2.

Важи да је

$$\begin{aligned} x_{12} &= r_1 \cdot y_2 \cdot m_2 + r_2 \cdot y_1 \cdot m_1 \\ &= r_1 \cdot y_2 \cdot m_2 + r_2 \cdot y_1 \cdot m_1 + r_1 \cdot y_1 \cdot m_1 - r_1 \cdot y_1 \cdot m_1 \\ &= r_1 \cdot (y_2 \cdot m_2 + y_1 \cdot m_1) + y_1 \cdot m_1 \cdot (r_2 - r_1) \\ &= r_1 + y_1 \cdot m_1 \cdot (r_2 - r_1) \end{aligned}$$

Зато је остатак при дељењу x_{12} бројем m_1 једнак r_1.

Слично се доказује и да је $x_{12} = r_2 + y_2 \cdot m_2 \cdot (r_1 - r_2)$ па је остатак при дељењу x_{12} бројем m_2 једнак r_2.

Дакле, ако Кинеску теорему о остацима примењујемо на два броја, не морамо два пута независно рачунати модуларни инверз, него оба потребна коефицијента можемо добити једном применом проширеног Еуклидовог алгоритма.

Ако има више бројева на које је потребно применити Кинеску теорему о остацима, након одређивања решења за прва два, исти поступак се примењује да се одреди број који при дељењу са $m_1 \cdot m_2$ даје остатак x_{12}, а при дељењу са m_3 даје остатак r_3. Ако је $k > 3$, поступак се наставља на исти начин, док се не добије коначан резултат.

Привремени резултати могу да прекораче опсег 64-битног типа, чак иако су сви r_i и m_i 32-битни целих бројеви и ако се модули рачунају пре и након сваке операције множења. Наиме, у формули $(r_1 \cdot y_2 \cdot m_2 + r_2 \cdot y_1 \cdot m_1) \bmod (m_1 \cdot m_2)$ се множе два пута по три 32-битна броја, а модул $m_1 \cdot m_2$ може бити прилично велики број (јер је добијен множењем два 32-битна броја). То се може предупредити ако се примети да је $(r_1 \cdot y_2 \cdot$

$m_2) \bmod (m_1 \cdot m_2) = m_2 \cdot ((r_1 \cdot y_2) \bmod m_1)$ и да је $(r_2 \cdot y_1 \cdot m_1) \bmod (m_1 \cdot m_2) = m_1 \cdot ((r_2 \cdot y_1) \bmod m_2)$.

Пример 3.4.14

Прикажимо претходни алгоритам на једном примеру. Нека је $(r_1, m_1) = (2, 3)$, $(r_2, m_2) = (3, 5)$ *и* $(r_3, m_3) = (2, 7)$. *Важи да је* $M = m_0 \cdot m_1 \cdot m_2 = 105$.

Пронађимо прво бројеве y_1 *и* y_2 *такве да је* $y_1 \cdot m_1 + y_2 \cdot m_2 = 3y_1 + 5y_2 = 1$. *Важи, на пример, да је* $y_1 = 2$ *и* $y_2 = -1$, *јер је* $3 \cdot 2 + 5 \cdot (-1) = 1$. *Тражени број је* $x_{12} = r_1 \cdot y_2 \cdot m_2 + r_2 \cdot y_1 \cdot m_1 = 2 \cdot (-1) \cdot 5 + 3 \cdot 2 \cdot 3 = 8$. *Он при дељењу са* 3 *даје остатак* 2, *а при дељењу са са* 5 *даје остатак* 3.

У наредном кораку тражимо број такав да при дељењу са $m_1 \cdot m_2 = 15$ *даје остатак* $r_{12} = x_{12} = 8$, *а при дељењу са* $m_3 = 7$ *даје остатак* $r_3 = 2$. *Зато одређујемо бројеве* y_1' *и* y_2' *такве да је* $15y_1' + 7y_2' = 1$. *То могу бити бројеви* $y_1' = 1$ *и* $y_2' = -2$. *Тражени број је тада* $8 \cdot 7 \cdot (-2) + 2 \cdot 15 \cdot 1 = -82$ *тј.* $-82 + 105 = 23$ *(пошто је* -82 *негативан увећавамо га за* $(m_1 m_2) m_3 = 105$*).*

Задатак: Билијар

Билијарски сто је правоугаоног облика димензије $m \times n$ и има четири рупе у ћошковима. Лоптица се удара из поља са целобројним координатама (x, y) (при чему то не може бити нека рупа), дуж линије која је паралелна или је под углом од 45 степени у односу на неку од ивица стола. Ако претпоставимо да лоптица не успорава своје кретање, да се од сваке ивице одбија под углом од 45°, да је веома мала и да у рупу упада само ако су јој координате центра једнаке координати рупе, напиши програм који одређује да ли ће лоптица некада упасти у рупу и ако хоће у коју рупу ће упасти.

Опис улаза

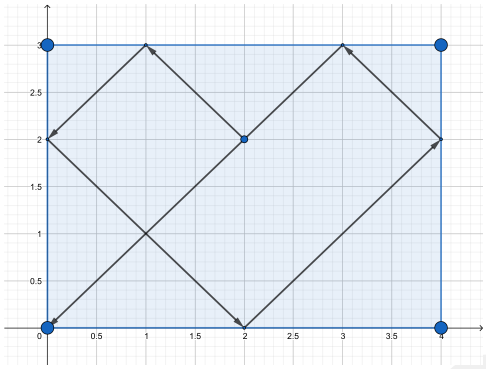
Са стандардног улаза се учитава 6 целих бројева. Димензије стола m и n ($1 \leq m, n \leq 10^9$), координате почетне позиције лоптице x и y ($0 \leq x \leq m$ и $0 \leq y \leq n$), тако да те координате не одређују рупу и хоризонтална и вертикална компонента брзине лоптице v_x и v_y ($-1 \leq v_x, v_y \leq 1$).

Опис излаза

На стандардни излаз исписати координате рупе у коју ће упасти лоптица или -1 ако ће се лоптица бесконачно дуго одбијати.

Пример 1

Улаз	*Излаз*	*Објашњење*
`4 3 2 2 -1 1`	`0 0`	Путања лоптице је приказана на слици.



Пример 2

Улаз	*Излаз*
`4 4 2 0 1 1`	`-1`

Пример 3

Улаз	*Излаз*
`10 10 10 1 -1 0`	`-1`

Решење

Случајеви када се лоптица испаљује паралелно са ивицом стола су тривијални и могу се једноставно директно разрешити. Претпоставимо зато да је лоптица испаљена дуж неке косе праве.

Обележимо са r_x почетно хоризонтално растојање које лоптица пређе од њеног почетног положаја до леве тј. десне ивице стола ка којој се креће чим се испали. Растојање r_x биће једнако нули ако се лоптица већ налази на левој или десној ивици, биће једнако x ако се лоптица испаљује налево или $m - x$ ако се испаљује надесно. Током свог кретања лоптица се налази на левој или на десној ивици стола када хоризонтално пређе растојање r_x, након тога када додатно пређе једну ширину стола (када пређе $r_x + m$), након тога када додатно пређе још једну ширину стола (када пређе $r_x + 2m$) и тако даље. Дакле, лоптица ће бити на левој или десној ивици када је дужина њеног пређеног пута у хоризноталном правцу једнако $r_x + k_x m$, за неко целобројно k_x. Пошто је почетно растојање r_x до најближе ивице увек мање од ширине стола m, лоптица ће бити на левој или десној ивици ако и само ако је остатак при дељењу хоризонталног растојања са m једнак r_x. Слично, лоптица ће бити на горњој или доњој ивици стола ако и само ако је остатак при дељењу вертикалног растојања са n једнак r_y, где је r_y почетно растојање до горње или доње ивице ка којој је лоптица испаљена. Пошто је интензитет хоризонталне и вертикалне компоненте брзине увек једнак, укупно хоризонтално и вертикално пређено растојање биће увек једнако. Дакле, лоптица ће бити у рупи када хоризонтално и вертикално пређе најмање растојање r такво да је остатак

при дељењу r са m једнако r_x и када је остатак при дељењу r са n једнако r_y.

Када су r_x и r_y једнаки нули (када се лоптица на почетку налази у рупи), r ће бити НЗС бројева m и n. Када нису, онда се r може одредити применом Кинеске теореме о остацима. Бројеви m и n не морају бити узајамно прости, па се не можемо директно позвати на основни облик теореме. Нека је d највећи заједнички делилац бројева m и n. Докажимо да лопта упада у рупу ако и само ако бројеви r_x и r_y имају исти остатак при дељењу са d.

Претпоставимо да је $r_x = q_x \cdot d + r'_x$ и да је $r_y = q_y \cdot d + r'_y$, за $0 \leq r'_x, r'_y < d$. Ако би постојао број r који би давао остатак r_x при дељењу са m и остатак r_y при дељењу са n, важило би да је $r = q_m \cdot m + r_x = q_n \cdot n + r_y$. Пошто је d делилац бројева m и n важило би да је $q_m \cdot (m' \cdot d) + q_x \cdot d + r'_x = q_n(n' \cdot d) + q_y \cdot d + r'_y$. Зато разлика $r'_x - r'_y$ мора бити дељива са d, а пошто су оба та броја мања од d, они морају бити једнаки. Дакле, ако решење постоји, остатак при дељењу бројева r_x и r_y са d мора бити исти. Доказ супротног смера је конструктиван и дат је у наставку.

Ако бројеви r_x и r_y дају исти остатак при дељењу са d, тада задатак можемо решити мало модификованом применом Кинеске теореме о остацима. Тврдимо да ће постојати јединствен број мањи од највећег заједничког садржаоца бројева m и n (који је једнак $(m \cdot n)/d$) који при дељењу са m даје остатак r_x, а при дељењу са n даје остатак r_y.

Коришћењем Безуове теореме изразимо $d = c_m \cdot m + c_n \cdot n$, за неке целобројне коефицијенте c_m и c_n. Скраћивањем са d добијамо да је $c_m \cdot (m/d) + c_n \cdot (n/d) = 1$. Тражено решење је број $(r_x \cdot c_n \cdot (n/d) + r_y \cdot c_m \cdot (m/d)) \bmod ((m \cdot n)/d)$.

```
int m, n, x, y, vx, vy;
cin >> m >> n >> x >> y >> vx >> vy;
if (vx == 0 && vy == 0)
  // loptica stoji
  cout << "-1" << endl;
else if (vx == 0) {    // loptica se krece samo vertikalno
  if (x != 0 && x != m)
    cout << "-1" << endl;
  else if (vy > 0)
    cout << x << " " << m << endl;
  else
    cout << x << " " << 0 << endl;
} else if (vy == 0) {  // loptica se krece samo horizontalno
  if (y != 0 && y != n)
    cout << "-1" << endl;
```

```
  else if (vx > 0)
    cout << m << " " << y << endl;
  else
    cout << 0 << " " << y << endl;
} else {               // loptica se krece dijagonalno
  // horizontalno rastojanje potrebno da loptica dodje do leve ili
  // desne ivice
  int rx = (vx > 0 ? (m - x) : x) % m;
  // vertikalno rastojanje potrebno da loptica dodje do gornje ili
  // donje ivice
  int ry = (vy > 0 ? (n - y) : y) % n;
  // horizontalno i vertikalno rastojanje koje loptica predje dok ne
  // upadne u rupu
  long long r;
  // rastojanje odredjujemo primenom KTO
  if (!kto(rx, m, ry, n, r))
    // loptica nikada ne upada u rupu
    cout << "-1" << endl;
  else {
    // prva ivica koju loptica dodirne
    int prva_ivica_x;        // 0 je leva, a 1 je desna
    if (x == 0)              // vec je na levoj ivici
      prva_ivica_x = 0;
    else if (x == m)         // vec je na desnoj ivici
      prva_ivica_x = 1;
    else                     // zavisi od brzine
      prva_ivica_x = vx < 0 ? 0 : 1;
    int prva_ivica_y;        // 0 je donja, a 1 je gornja
    if (y == 0)              // vec je na donjoj ivici
      prva_ivica_y = 0;
    else if (y == n)         // vec je na gornjoj ivici
      prva_ivica_y = 1;
    else                     // zavisi od brzine
      prva_ivica_y = vy < 0 ? 0 : 1;
    // broj prelaza stola koja loptica napravi
    long long broj_prelaza_x = (r - rx) / m;
    long long broj_prelaza_y = (r - ry) / n;
    // koordinate na kojima se loptica nalazi
    int rupax = (prva_ivica_x + broj_prelaza_x) % 2 == 0 ? 0 : m;
```

```
    int rupay = (prva_ivica_y + broj_prelaza_y) % 2 == 0 ? 0 : n;
    cout << rupax << " " << rupay << endl;
  }
}
```

3.5 RSA криптографија

RSA (Ривест[12], Шамир[13], Адлеман[14]) је алгоритам који је уведен 1977. године и који се и данас интензивно користи у многим областима примене рачунара. У наставку ћемо описати овај алгоритам и тиме приказати како се алгоритми и концепти које смо до сада у овом поглављу упознали лепо комбинују и дају веома озбиљну, реалну примену.

Основна подела криптографских алгоритама је на:

- *симетричне*, који подразумевају да се за шифровање и дешифровање порука користи исти кључ и
- *асиметричне*, који подразумевају да се за шифровање и дешифровање порука користе различити кључеви.

Обе врсте алгоритама се користе у савременој електронској комуникацији. Симетрична криптографија је бржа, али је проблем размена кључева између странака које комуницирају. Асиметрична криптографија не захтева размену кључева, али је спорија и мање сигурна. Обично се асиметрична криптографија употребљава у првој фази комуникације током које се установљава идентитет странака које комуницирају и размењује се симетрични кључ који се даље користи у другој фази комуникације (на пример, тако ради протокол TLS, који је основа шифроване комуникације на вебу).

Асиметрична криптографија подразумева да особа која прима поруке има свој:

- *јавни кључ* који је познат свима и који онај ко шаље поруку користи за шифровање, и
- *тајни кључ* који је само њој познат који она користи за дешифровање.

Алгоритам RSA се сматра зачетником асиметричне криптографије. И шифровање и дешифровање се врше помоћу степеновања по модулу n. У алгоритму RSA јавни кључ је пар бројева (e, n), а тајни кључ (d, n) (користимо слово e за *encryption*, тј. шифро-

[12] Роналд Ривест (енгл. Ronald Rivest), рођен 1947., амерички криптограф и информатичар.
[13] Ади Шамир (енгл. Adi Shamir), рођен 1952., израелски криптограф.
[14] Леонард Адлеман (енгл. Leondard Adleman), рођен 1945., амерички информатичар.

вање, а d за *decryption*, тј. дешифровање)[15]. Шифрује се порука (енгл. *message*) m и добија се шифрат (енгл. *cypher*) c тако да важи $c = m^e \bmod n$. Већи документи се деле на мање поруке и свака порука m се представља бројем мањим од n и засебно се шифрује. Величина поруке зависи од величине кључева. Поруке не могу да буду веће од величине кључа. Ако се бројеви d и e представљају са по 4096 битова, поруке m су обично величине око 500 бајтова и често се допуњавају додатним насумично генерисаним бајтовима, да би се спречило да се исте поруке увек шифрују на исти начин. Кључно својство које мора бити испуњено је да се операције шифровања и дешифровања поништавају, тј. да се дешифровањем шифрата c израчунавањем $c^d \bmod n$ добија оригинална порука m. Да би ово својство важило, довољно је да бројеви e и d буду међусобно инверзни по модулу $\varphi(n)$, што је једноставна последица Ојлерове теореме.

Теорема 3.5.1 **[RSA: дешифровање шифроване поруке]**

Ако су бројеви e и d међусобно инверзни по модулу $\varphi(n)$, тј. ако је $e \cdot d \equiv 1 \pmod{\varphi(n)}$ и ако је порука $m < n$ узајамно проста са бројем n, тада важи $c^d = (m^e)^d \equiv m \pmod n$.

Доказ. Пошто је $e \cdot d \equiv 1 \pmod{\varphi(n)}$ број $e \cdot d - 1$ је дељив са $\varphi(n)$, тј. постоји k такво да је $e \cdot d = k\varphi(n) + 1$. Пошто су m и n узајамно прости, на основу Ојлерове теореме 3.4.2 важи да је $m^{\varphi(n)} \equiv 1 \pmod n$. Зато је

$$c^d \equiv (m^e)^d = m^{e\cdot d} = m^{k\varphi(n)+1} = (m^{\varphi(n)})^k \cdot m \equiv 1^k \cdot m = m \pmod n \qquad \square$$

Дакле, ако јавни кључ садржи број e, тада број d треба изаберати као његов инверз по модулу $\varphi(n)$. Бројеви e и n су познати нападачу, али желимо да он не може у разумном времену да израчуна d на основу њихове вредности. Шта је то што ће нама омогућити да приликом генерисања кључева израчунамо d, а што недостаје нападачу да би и он то могао ефикасно да уради? То што ми у тренутку генерисања кључева можемо да знамо вредност $\varphi(n)$, а нападачу је потребно огромно време да од n израчуна $\varphi(n)$. Шта би нама омогућило да брзо израчунамо $\varphi(n)$? Ако бисмо знали факторизацију броја n, онда би израчунавање Ојлерове функције било једноставно. Претпоставићемо зато да смо број n одабрали као производ нека два проста броја p и q. Њих ћемо искористити приликом генерисања кључева e и d и израчунавања вредности $\varphi(n)$ и заборавити одмах након тога (нигде их нећемо записати). Познавање факторизације $n = p \cdot q$ нам омогућава да ефикасно израчунамо вредност $\varphi(n) = (p-1)\cdot(q-1)$ (коју ћемо такође заборавити одмах након генерисања кључева), а затим да на основу

[15] Може се сматрати и да су кључеви бројеви e и d, међутим, пошто је n потребно за израчунавање степена по модулу, и њега ћемо сматрати делом оба кључа.

вредности дела јавног кључа e израчунамо његов модуларни инверз d (по модулу $\varphi(n)$), што се лако ради коришћењем проширеног Еуклидовог алгоритма.

Дакле, алгоритам RSA ради на следећи начин.

1. Генерисање кључева започиње генерисањем два проста броја p и q и одређивањем вредности модула $n = p \cdot q$. Бројеви p и q су тајни, једнократно се употребљавају и одмах заборављају (нигде се не складиште). Бројеви p и q се генеришу коришћењем генератора насумичних бројева, све док се не установи да су прости. Бројеви p и q морају бити велики и њихова разлика мора бити велика.

2. Израчунава се вредност Ојлерове функције $\varphi(n)$ (и она је тајна, употребљава се једнократно и нигде не складишти). У погављу 3.3 о мултипликативним функцијама је доказано да је у случају $n = p \cdot q$ вредност $\varphi(n)$ једнака $(p-1) \cdot (q-1)$, тако да се она лако израчунава када се знају бројеви p и q.

3. Бира се вредност јавног кључа e, таква да је $2 \leq e < \varphi(n)$ и да је e узајамно просто са $\varphi(n)$. Вредност e се често бира као мали број (када је e мали број, степеновање се брже врши). Може се одабрати да је e прост број (чиме се обезбеђује да се лакше испита да ли је узајамно просто са $\varphi(n)$). Још чешће, узима се нека вредност облика $2^k + 1$ (нпр. $2^{16} + 1 = 65537$), у чијем је бинарном запису мало јединица, што обезбеђује ефикасно модуларно степеновање.

4. Вредност d се одређује као модуларни мултипликативни инверз вредности e по модулу $\varphi(n)$, тј. тако да важи да је $e \cdot d \equiv 1 (\mathrm{mod}\ \varphi(n))$. Она постоји (јер је e одабрано тако да је узајамно просто са $\varphi(n)$) и може се ефикасно израчунати неким од раније описаних алгоритама (пре свега проширеним Еуклидовим алгоритмом). Ако се e одабере као мали број, вредност d може бити прилично велики број, што значи да ће се шифровање вршити брже него дешифровање.

5. Порука m се шифрује израчунавањем вредности $c = (m^e) \bmod n$.

6. Порука се дешифрује израчунавањем вредности $m = (c^d) \bmod n$.

Дешифровање је коректно на основу теореме 3.5.1. Услов да је број m узајамно прост са n се лако обезбеђује. Ако је m мање од простих бројева p и q, оно је узајамно просто са њима, па и са њиховим производом. Чак и ако се за m узме нека већа вредност, вероватноћа да она буде умножак неког од бројева p или q је мала.

Пример 3.5.1

Нека је $p = 61$ *и* $q = 53$. *Онда је* $n = p \cdot q = 3233$ *и важи* $\varphi(n) = (p-1) \cdot (q-1) = 60 \cdot 52 = 3120$. *За број* e *бирамо број мањи од* 3120 *који је узајамно прост са* 780, *на пример* $e = 17$. *Модуларни мултипликативни инверз броја* 17 *по модулу* 3120

једнак је $d = 413$ *(њега можемо ефикасно израчунати проширеним Еуклидовим алгоритмом). Шифровање поруке* m *своди се на рачунање* $c = m^{17} \bmod 3233$*, а дешифровање поруке* c *на рачунање* $m = c^{413} \bmod 3233$*. На пример, шифровањем поруке* $m = 65$ *добија се* $c = 65^{17} \bmod 3233 = 2790$*, а дешифровањем ове поруке добијамо* $m = 2790^{413} \bmod 3233 = 65$*, што јесте вредност полазне поруке.*

Алгоритам RSA се може употребити и за потписивање порука. Да би поступак био бржи, уместо потписивање целе поруке обично се потписује само њена хеш-вредност[16]. На основу поруке m израчунава се њена хеш-вредност $h(m)$ и уз m се шаље потпис $h(m)^d$ (приметимо да се хеш-вредност степенује коришћењем тајног кључа пошаљиоца d, који се иначе користи за дешифровање). Прималац поруке m дешифрује потпис $h(m)$ коришћењем јавног кључа e пошиљаоца (који му је познат), степенујући добијени потпис бројем e. Тиме ће реконструисати вредност $h(m)$. Такође, на основу поруке m, прималац може поново да израчуна вредност $h(m)$ и ако се та вредност поклопи са оном добијеном дешифровањем потписа, може да тврди да порука m није у међувремену мењана, као и да ју је послала особа која је била у поседу тајног кључа d.

Алгоритам RSA, дакле, почива на следећим чињеницама:

- Ако су e и d инверзни елементи по модулу $\varphi(n)$, тј. ако је $e \cdot d \equiv 1 \pmod{\varphi(n)}$, тада (на основу Ојлерове теореме) важи $m^{ed} \equiv m \pmod{n}$ за све бројеве m узајамно просте са n.
- Ако се зна број $\varphi(n)$, тада се мултипликативни инверз броја e по модулу $\varphi(n)$ може брзо израчунати.
- Ако се знају прости бројеви p и q такви да је $n = p \cdot q$, тада се вредност $\varphi(n)$ може израчунати јако брзо.
- Ако се зна вредност n, али не и бројеви p и q, тада се вредности p и q, нити вредност $\varphi(n)$, практично не могу израчунати.
- Ако се неком грешком сазна пар вредности порука m и шифрат $c = m^e \bmod n$, то не омогућава да се у разумном времену одреди кључ d. Наиме, иако важи $c^d \bmod n = m$, проналажење d захтева израчунавање дискретног логаритма, што практично није могуће урадити.

Сигурност алгоритма RSA се, дакле, заснива на тежини факторизације великих бројева, тежини израчунавања Ојлерове функције и тежини израчунавања дискретног логаритма. Наиме, иако је информација о броју n јавна, она нам не омогућава да ефикасно израчунамо вредност $\varphi(n)$ која је потребна да би се од јавног кључа e израчунао тајни

[16] У криптографским применама се уместо класичних, користе тзв. криптографске хеш-функције, за које је веома тешко израчунати улазну вредност чијим се хеширањем добија задата хеш-вредност, тј. за које је јако тешко пронаћи две улазне вредности чијим се хеширањем добија иста хеш-вредност (које доводе до колизије).

кључ d (као његов инверз по модулу $\varphi(n)$). Сигурност алгоритма RSA би се нарушила ако би се број n могао ефикасно факторисати или ако би се $\varphi(n)$ могло израчунати без факторизације броја n, на неки ефикаснији начин. Претпоставимо да је n једнако производу два велика проста броја p и q, који немају сличан ред величине. Тада је проблем факторизације броја $n = p \cdot q$ јако тежак. У поглављима 3.2 и 3.3 посвећеним факторизацији бројева и њеним применама приказани су алгоритми факторизације броја n и израчунавања $\varphi(n)$ чија је сложеност једнака $O(\sqrt{n})$, што је за велике вредности броја p и q (вредности са неколико стотина, па и хиљада бинарних цифара) неизводиво у разумном времену. Тада ни најнапреднији познати алгоритми немају никакву шансу да изврше факторизацију, нити да израчунају вредност $\varphi(n)$ у неком разумном времену (данима, месецима, годинама), чак ни уз масовну паралелизацију. Нагласимо да бројеви p и q не смеју да буду сличног реда величине, тј. да имају сличан број цифара, иначе би били блиски вредности $\sqrt{n}$ и могли би се брзо одредити Фермаовим алгоритмом факторизације (који је описан у поглављу 3.2.2). Препоручује се, на пример, да се бројеви битова бројева p и q разликују бар за 2.

Факторизација омогућава једноставно израчунавање Ојлерове функције броја $n = p \cdot q$, али важи и обратно. Ако је поред броја n позната и вредност Ојлерове функције $\varphi(n)$, онда се бројеви p и q могу једноставно одредити. Наиме, пошто важи $\varphi(n) = (p-1) \cdot (q-1)$, множењем израза са десне стране знака једнакости добијамо да је $n - \varphi(n) + 1 = p + q$. На основу вредности $p + q$ и $p \cdot q$ се могу одредити бројеви p и q као решења квадратне једначине $x^2 - (n - \varphi(n) + 1)x + n = 0$.

За сада није познат ефикасан алгоритам за факторизацију великих бројева n нити за израчунавање вредности $\varphi(n)$, али није ни доказано да такав алгоритам не постоји (ово је слично као код NP комплетних проблема, међутим, није доказано да су ови проблеми NP комплетни – ови проблеми су у класи NP, али је могуће да су лакши од NP комплетних проблема). Када би се нашао ефикасан алгоритам којим се решава неки од ових проблема, то би могло имати озбиљан утицај на сигурност и безбедност података.

3.6 Брза Фуријеова трансформација

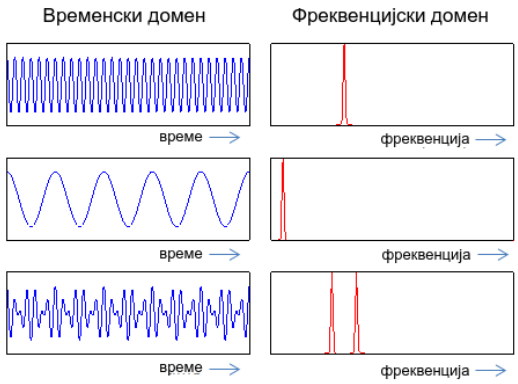
Брза Фуријеова трансформација (енгл. *Fast Fourier transform*, *FFT*) од свог открића средином шездесетих година двадесетог века спада у најзначајније алгоритме 20. века.[17] Иако су значајне примене брзе Фуријеове трансформације настале након рада Кулија[18] и Тукија[19] 1965. године, на пољу обраде сигнала (у циљу детектовања нуклеарних проба

[17] На пример, удружење IEEE објавило је ову листу: https://www.andrew.cmu.edu/course/15-355/misc/Top%20Ten%20Algorithms.html

[18] Џејмс Кули (енгл. James Cooley), (1926-2016), амерички математичар.

[19] Џон Туки (енгл. John Tukey), (1915-200), амерички математичар.

обрадом сеизмичких мерења), испоставило се да су ова два аутора независно изнова „измислили" алгоритам који је осмислио Гаус[20] још давне 1805. године. Брза Фуријеова трансформација има бројне примене у инжењерству, у обради дигиталних сигнала попут обраде звука и обраде дигиталних слика и у математици. *Фуријеовом трансформацијом* непрекидног сигнала врши се његово превођење из временског у фреквенцијски домен, тј. сигнал се разлаже на збир синусоидалних елементарних таласа (на пример, музички акорд се може разложити на појединачне тонове који га сачињавају). Ако је сигнал представљен вектором добијеним мерењем његовог интензитета у дискретним временским интервалима, примењује се *дискретна Фуријеова трансформација*. На слици 3.1 приказана су три сигнала, посебно у временском, а посебно у фреквенцијском домену. Види се да су прва два правилне синусоиде (имају само једну доминантну фреквенцију), док је трећи добијен сабирањем две правилне синусоиде (има две доминантне фреквенције).



Слика 3.1: Сигнали у временском домену (лево) у ком су на x-оси приказани временски тренуци, а на y-оси амплитуда сигнала и фреквенцијском домену (десно) у ком су на x-оси приказане фреквенције, а на y-оси мера присуства те фреквенције у сигналу.

Брза Фуријеова трансформација је заправо само ефикасан алгоритам за израчунавање дискретне Фуријеове трансформације. Ми ћемо се у овом уџбенику ограничити на једну њену примену, а то је множење полинома.

Проблем

Израчунати производ два задата полинома $P(x)$ *и* $Q(x)$.

Формулација проблема који треба решити је прецизна само на први поглед, јер није прецизиран начин на који су полиноми представљени. Обично се полином $P(x) =$

[20] Карл Фридрих Гаус (нем. Carl Friedrich Gauss), (1777-1885), немачки математичар.

$a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_1x + a_0$, степена $n-1$, представља низом својих n коефицијената $a_0, a_1, a_2, ... a_{n-1}$ уз $1, x, x^2, ..., x^{n-1}$, међутим, то није једина могућност. Алтернативно, полином степена $n-1$ могуће је представити његовим вредностима у n различитих тачака: те вредности једнозначно одређују полином. Полином степена 1, односно линеарна функција, јединствено је одређена вредностима у две различите тачке, полином степена 2, односно квадратна функција, вредностима у три различите тачке итд.

Пример 3.6.1

Вредности $P(0) = 3$, $P(1) = 6$, $P(-2) = 3$ *једнозначно одређују полином* $P(x) = x^2 + 2x + 3$ *степена* 2, *па се он може једнозначно представити било низом коефицијената* $[1, 2, 3]$, *било низом парова* $[(0, 3), (1, 6), (-2, 3)]$, *који садржи вредности* $(x, P(x))$.

Множење два полинома која су задата низом својих коефицијената може се извршити основним алгоритмом сложености $O(n^2)$ који се састоји у множењу сваког монома првог полинома сваким мономом другог. Могуће је и ефикасније множење, коришћењем Карацубиног алгоритма заснованог на паметној примени декомпозиције, који је сложености $O(n^{\log_2 3})$. Циљ овог поглавља је да применом алгоритма FFT изведемо алгоритам сложености $O(n \log n)$, али за то је потребно да детаљније размотримо другу репрезентацију.

Представљање полинома вредностима на скупу тачака је интересантно због једноставности множења. Наиме, производ два полинома степена $n-1$ је полином степена $2n-2$, па је одређен својим вредностима у $2n-1$ тачака. Ако претпоставимо да су вредности полинома-чинилаца степена $n-1$ дате на истом скупу од $2n-1$ различитих тачака (што је, приметимо, више тачака него што је неопходно за једнозначност), онда се производ ова два полинома може израчунати помоћу $2n-1$, односно $O(n)$ обичних множења. Наиме, вредност полинома PQ у било којој тачки x једнака је производу вредности полинома P у тачки x и вредности полинома Q у тачки x, тј. важи $(P \cdot Q)(x) = P(x) \cdot Q(x)$.

Пример 3.6.2

Производ полинома $[(0, 3), (1, 6), (-2, 3)]$ *и* $[(0, 1), (1, -1), (-2, 4)]$ *је полином* $[(0, 3), (1, -6), (-2, 12)]$.

Нажалост, представљање полинома његовим вредностима на скупу тачака није погодно за неке примене. Пример је израчунавање вредности полинома у произвољној тачки:

при репрезентацији полинома вредностима на скупу тачака, ово је много теже у односу на то када је полином задат низом својих коефицијената (потребно је решити проблем интерполације). Са друге стране, вредност полинома $P(x)$ степена $n-1$ задатог низом својих коефицијената у произвољној тачки x може се помоћу Хорнерове шеме израчунати коришћењем n множења, тј. алгоритмом сложености $O(n)$:

$$P(x) = a_0 + x(a_1 + x(a_2 + ... + x \cdot a_{n-1} ...))$$

Дакле, свака од репрезентација је погодна за једну, а није погодна за другу операцију, како је илустровано у табели 3.1.

Табела 3.1: Сложеност основних операција за рад са полиномима у зависности од репрезентације полинома

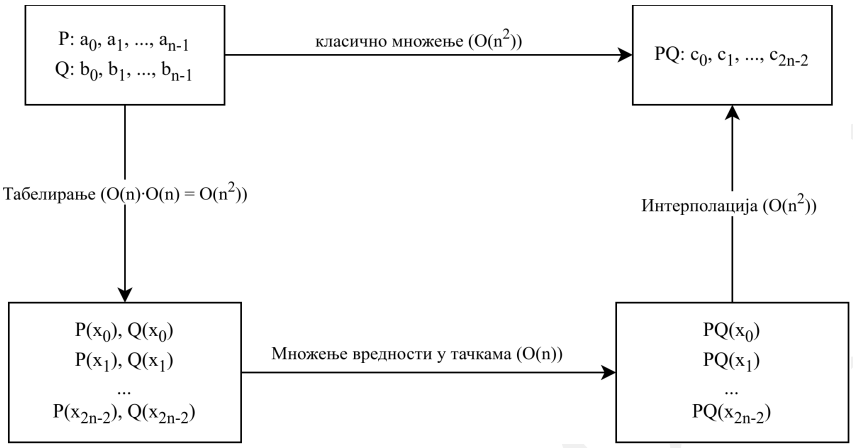
репрезентација	рачунање вредности полинома у тачки	множење полинома
коефицијенти	$\mathbf{O(n)}$	$O(n^2)$ или $O(n^{\log_2 3})$
вредности	$O(n^2)$	$\mathbf{O(n)}$

Ако бисмо могли да ефикасно преводимо полиноме из једне у другу репрезентацију, добили бисмо могућност да обе најзначајније операције (множење и израчунавање вредности) извршавамо ефикасно. Применом брзе Фуријеове трансформације, тј. алгоритма FFT, сложеност превођења из једне у другу репрезентацију полинома постаје $O(n \log n)$.

Без примене алгоритма FFT прелаз од представљања полинома коефицијентима на представљање полинома вредностима у тачкама (који ћемо звати *табелирање*), решава се узастопним израчунавањем вредности полинома у свакој од тих тачака. Израчунавање вредности полинома $P(x)$ степена $n-1$ у n произвољних тачака (што је потребно за једнозначно представљање полинома) изводљиво је применом Хорнерове схеме помоћу $O(n^2)$ множења. Прелаз од представљања полинома вредностима на скупу тачака на представљање коефицијентима се зове *интерполација*. Вредности коефицијента полинома на основу вредности $(x_j, y_j), 0 \leq j \leq n$ полинома у $n+1$ различитих тачака могу се одредити коришћењем Лагранжове интерполационе формуле (која се обично изучава у склопу нумеричке математике):

$$P(x) = \sum_{j=0}^{n-1} a_j x^j = \sum_{j=0}^{n-1} y_j \cdot \frac{\prod_{k \neq j}(x - x_k)}{\prod_{k \neq j}(x_j - x_k)}$$

На основу овог израза коефицијенти полинома могу се одредити алгоритмом сложености $O(n^2)$.



Слика 3.2: Две могућности за рачунање производа два полинома $P(x) = \sum_{i=0}^{n-1} a_i x^i$ и $Q(x) = \sum_{i=0}^{n-1} b_i x^i$: директним множењем полинома што је сложености $O(n^2)$ (горња стрелица удесно) или преко репрезентације полинома вредностима на скупу тачака (стрелица надоле, доња стрелица удесно, стрелица нагоре). Убрзавањем корака табелирања и интерполације можемо добити ефикаснији алгоритам.

Поставља се питање како је могуће убрзати кораке табелирања и интерполације? Кључна идеја да се не користи произвољних n тачака: ми имамо слободу да по жељи изаберемо погодан скуп од n различитих тачака. Брза Фуријеова трансформација користи специјалан скуп тачака, тако да се и табелирање и интерполација могу ефикасније извршавати.

3.6.1 Директна брза Фуријеова трансформација

Основна идеја на којој почива брза Фуријеова трансформација је изложена у наредном примеру.

Пример 3.6.3

Претпоставимо да треба израчунати вредност полинома $P(x) = x^7 + 2x^6 + 3x^5 + 4x^4 + 5x^3 + 6x^2 + 7x + 8$ у две различите тачке. Ако су оне међусобно супротне, можемо смањити количину потребних израчунавања на основу тога што су полиноми x^k непарног степена k непарне, а парног степена k парне функције. Претпоставимо да, на пример, треба да израчунамо $P(2)$ и $P(-2)$. Важи

$$\begin{aligned} P(2) &= 2^7 + 2\cdot 2^6 + 3\cdot 2^5 + 4\cdot 2^4 + 5\cdot 2^3 + 6\cdot 2^2 + 7\cdot 2 + 8 \\ P(-2) &= (-2)^7 + 2\cdot(-2)^6 + 3\cdot(-2)^5 + 4\cdot(-2)^4 + 5\cdot(-2)^3 + 6\cdot(-2)^2 + 7\cdot(-2) + 8 \\ &= -2^7 + 2\cdot 2^6 - 3\cdot 2^5 + 4\cdot 2^4 - 5\cdot 2^3 + 6\cdot 2^2 - 7\cdot 2 + 8. \end{aligned}$$

Ако знамо вредности $P_n(2) = 2^7+3\cdot 2^5+5\cdot 2^3+7\cdot 2$ *и* $P_p(2) = 2\cdot 2^6+4\cdot 2^4+6\cdot 2^2+8$, *тада се* $P(2)$ *може израчунати као* $P(2) = P_p(2) + P_n(2)$, *а* $P(-2) = P_p(2) - P_n(2)$. *Дакле, вредност полинома* P *у тачкама* -2 *и* 2 *је израчуната помоћу вредности полинома* P_p *и* P_n *у тачки* 2. *Полиноми* P_p *и* P_n *имају по* 4 *коефицијента, али им је степен двоструко већи од броја коефицијената, но то можемо лако поправити.*

Важи да је $P_p(2) = 2\cdot(2^2)^3 + 4\cdot(2^2)^2 + 6\cdot 2^2 + 8$, *па је* $P_p(2)$ *заправо вредност полинома* $P_0(x) = 2x^3+4x^2+6x+8$ *у тачки* 2^2, *а* $P_n(2) = 2\cdot((2^2)^3+3\cdot(2^2)^2+5\cdot 2^2+7)$ *је заправо вредност полинома* $P_1(x) = x^3 + 3x^2 + 5x + 7$ *у тачки* 2^2 *помножена са* 2. *Важи* $P(2) = P_0(2^2) + 2\cdot P_1(2^2)$ *и* $P(-2) = P_0(2^2) - 2\cdot P_1(2^2)$. *Дакле, израчунавање вредности полинома степена* 7 *у две супротне тачке се своди на израчунавање вредности два полинома степена* 3 *у једној тачки (квадрату полазних супротних тачака). Та два полинома настала су од полазног издвајањем коефицијената на парним и коефицијената на непарним позицијама.*

Брза Фуријеова трансформација је алгоритам типа подели-па-владај. Претпоставља се да је димензија улаза степен двојке. Ако низ коефицијената није дужине $n = 2^k$, он се увек може допунити нулама. Размотримо за почетак случај $n = 8 = 2^3$ тј. претпоставимо да је потребно израчунати вредност полинома

$$P(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$$

у неких 8 различитих тачака $x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7$.

Прегрупишимо сабирке на следећи начин:

$$\begin{aligned} P(x) &= a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7 \\ &= (a_0 + a_2x^2 + a_4x^4 + a_6x^6) + x(a_1 + a_3x^2 + a_5x^4 + a_7x^6). \end{aligned}$$

На овај начин добијамо два мања полинома $P_0(x) = a_0+a_2x+a_4x^2+a_6x^3$ и $P_1(x) = a_1 + a_3x + a_5x^2 + a_7x^3$. Важи да је $P(x_i) = P_0(x_i^2) + x_iP_1(x_i^2)$. Потребно је, дакле, да израчунамо следећих 8 вредности полинома P:

$$
\begin{aligned}
P(x_0) &= P_0(x_0^2) + x_0 P_1(x_0^2) &&= (a_0 + a_2x_0^2 + a_4(x_0^2)^2 + a_6(x_0^2)^3) + x_0(a_1 + a_3x_0^2 + a_5(x_0^2)^2 + a_7(x_0^2)^3)\\
P(x_1) &= P_0(x_1^2) + x_1 P_1(x_1^2) &&= (a_0 + a_2x_1^2 + a_4(x_1^2)^2 + a_6(x_1^2)^3) + x_1(a_1 + a_3x_1^2 + a_5(x_1^2)^2 + a_7(x_1^2)^3)\\
P(x_2) &= P_0(x_2^2) + x_2 P_1(x_2^2) &&= (a_0 + a_2x_2^2 + a_4(x_2^2)^2 + a_6(x_2^2)^3) + x_2(a_1 + a_3x_2^2 + a_5(x_2^2)^2 + a_7(x_2^2)^3)\\
P(x_3) &= P_0(x_3^2) + x_3 P_1(x_3^2) &&= (a_0 + a_2x_3^2 + a_4(x_3^2)^2 + a_6(x_3^2)^3) + x_3(a_1 + a_3x_3^2 + a_5(x_3^2)^2 + a_7(x_3^2)^3)\\
P(x_4) &= P_0(x_4^2) + x_4 P_1(x_4^2) &&= (a_0 + a_2x_4^2 + a_4(x_4^2)^2 + a_6(x_4^2)^3) + x_4(a_1 + a_3x_4^2 + a_5(x_4^2)^2 + a_7(x_4^2)^3)\\
P(x_5) &= P_0(x_5^2) + x_5 P_1(x_5^2) &&= (a_0 + a_2x_5^2 + a_4(x_5^2)^2 + a_6(x_5^2)^3) + x_5(a_1 + a_3x_5^2 + a_5(x_5^2)^2 + a_7(x_5^2)^3)\\
P(x_6) &= P_0(x_6^2) + x_6 P_1(x_6^2) &&= (a_0 + a_2x_6^2 + a_4(x_6^2)^2 + a_6(x_6^2)^3) + x_6(a_1 + a_3x_6^2 + a_5(x_6^2)^2 + a_7(x_6^2)^3)\\
P(x_7) &= P_0(x_7^2) + x_7 P_1(x_7^2) &&= (a_0 + a_2x_7^2 + a_4(x_7^2)^2 + a_6(x_7^2)^3) + x_7(a_1 + a_3x_7^2 + a_5(x_7^2)^2 + a_7(x_7^2)^3)
\end{aligned}
$$

Да би се ово директно израчунало, потребно је израчунати укупно 16 вредности полинома P_0 и P_1, међутим, паметним избором тачака могуће је тај број смањити на 8. Наиме ми имамо слободу избора тачака x_k (док год их бирамо тако да су различите) и можемо одабрати тачке за које, на пример, важи $x_0^2 = x_4^2$, $x_1^2 = x_5^2$, $x_2^2 = x_6^2$, $x_3^3 = x_7^2$. Не смемо да узмемо исте вредности тачака, али можемо супротне. Наиме, начин да се постигне једнакост квадрата је да се вредности изаберу тако да важи $x_4 = -x_0$, $x_5 = -x_1$, $x_6 = -x_2$ и $x_7 = -x_3$. Тада се израчунавање своди на:

$$
\begin{aligned}
P(x_0) &= P_0(x_0^2) + x_0 P_1(x_0^2) &&= (a_0 + a_2x_0^2 + a_4(x_0^2)^2 + a_6(x_0^2)^3) + x_0(a_1 + a_3x_0^2 + a_5(x_0^2)^2 + a_7(x_0^2)^3)\\
P(x_1) &= P_0(x_1^2) + x_1 P_1(x_1^2) &&= (a_0 + a_2x_1^2 + a_4(x_1^2)^2 + a_6(x_1^2)^3) + x_1(a_1 + a_3x_1^2 + a_5(x_1^2)^2 + a_7(x_1^2)^3)\\
P(x_2) &= P_0(x_2^2) + x_2 P_1(x_2^2) &&= (a_0 + a_2x_2^2 + a_4(x_2^2)^2 + a_6(x_2^2)^3) + x_2(a_1 + a_3x_2^2 + a_5(x_2^2)^2 + a_7(x_2^2)^3)\\
P(x_3) &= P_0(x_3^2) + x_3 P_1(x_3^2) &&= (a_0 + a_2x_3^2 + a_4(x_3^2)^2 + a_6(x_3^2)^3) + x_3(a_1 + a_3x_3^2 + a_5(x_3^2)^2 + a_7(x_3^2)^3)\\
P(x_4) &= P_0(x_0^2) - x_0 P_1(x_0^2) &&= (a_0 + a_2x_0^2 + a_4(x_0^2)^2 + a_6(x_0^2)^3) - x_0(a_1 + a_3x_0^2 + a_5(x_0^2)^2 + a_7(x_0^2)^3)\\
P(x_5) &= P_0(x_1^2) - x_1 P_1(x_1^2) &&= (a_0 + a_2x_1^2 + a_4(x_1^2)^2 + a_6(x_1^2)^3) - x_1(a_1 + a_3x_1^2 + a_5(x_1^2)^2 + a_7(x_1^2)^3)\\
P(x_6) &= P_0(x_2^2) - x_2 P_1(x_2^2) &&= (a_0 + a_2x_2^2 + a_4(x_2^2)^2 + a_6(x_2^2)^3) - x_2(a_1 + a_3x_2^2 + a_5(x_2^2)^2 + a_7(x_2^2)^3)\\
P(x_7) &= P_0(x_3^2) - x_3 P_1(x_3^2) &&= (a_0 + a_2x_3^2 + a_4(x_3^2)^2 + a_6(x_3^2)^3) - x_3(a_1 + a_3x_3^2 + a_5(x_3^2)^2 + a_7(x_3^2)^3)
\end{aligned}
$$

Потребно је, дакле, рекурзивно израчунати само следећих 8 вредности полинома P_0 и P_1 (уместо полазних 16), а онда их искомбиновати помоћу претходних формула:

$$
\begin{aligned}
P_0(x_0^2) &= a_0 + a_2x_0^2 + a_4(x_0^2)^2 + a_6(x_0^2)^3 & \qquad P_1(x_0^2) &= a_1 + a_3x_0^2 + a_5(x_0^2)^2 + a_7(x_0^2)^3\\
P_0(x_1^2) &= a_0 + a_2x_1^2 + a_4(x_1^2)^2 + a_6(x_1^2)^3 & \qquad P_1(x_1^2) &= a_1 + a_3x_1^2 + a_5(x_1^2)^2 + a_7(x_1^2)^3\\
P_0(x_2^2) &= a_0 + a_2x_2^2 + a_4(x_2^2)^2 + a_6(x_2^2)^3 & \qquad P_1(x_2^2) &= a_1 + a_3x_2^2 + a_5(x_2^2)^2 + a_7(x_2^2)^3\\
P_0(x_3^2) &= a_0 + a_2x_3^2 + a_4(x_3^2)^2 + a_6(x_3^2)^3 & \qquad P_1(x_3^2) &= a_1 + a_3x_3^2 + a_5(x_3^2)^2 + a_7(x_3^2)^3
\end{aligned}
$$

Да ли можемо још уштедети на броју операција паметним избором тачака x_0, x_1, x_2 и x_3? Размотримо израчунавање вредности полинома P_0 (вредности полинома P_1 се рачунају аналогно). Групишимо поново коефицијенте на парним и на непарним позицијама. Тиме добијамо још мање полиноме $P_{00}(x) = a_0 + a_4x$ и $P_{01}(x) = a_2 + a_6x$.

$$
\begin{aligned}
P_0(x_0^2) &= P_{00}(x_0^4) + x_0^2 P_{01}(x_0^4) &&= (a_0 + a_4x_0^4) + x_0^2(a_2 + a_6x_0^4)\\
P_0(x_1^2) &= P_{00}(x_1^4) + x_1^2 P_{01}(x_1^4) &&= (a_0 + a_4x_1^4) + x_1^2(a_2 + a_6x_1^4)\\
P_0(x_2^2) &= P_{00}(x_2^4) + x_2^2 P_{01}(x_2^4) &&= (a_0 + a_4x_2^4) + x_2^2(a_2 + a_6x_2^4)\\
P_0(x_3^2) &= P_{00}(x_3^4) + x_3^2 P_{01}(x_3^4) &&= (a_0 + a_4x_3^4) + x_3^2(a_2 + a_6x_3^4)
\end{aligned}
$$

За израчунавање 4 вредности полинома P_0 потребно је израчунати укупно 8 вредности полинома P_{00} и P_{01} и желимо то да смањимо. Међутим, сада имамо проблем. Да бисмо

могли да смањимо број израчунавања на исти начин као у првом кораку, потребно је да одаберемо x_0 и x_2 тако да важи $x_0^4 = x_2^4$. Не можемо да одаберемо да је $x_0^2 = x_2^2$, јер је тада или $x_0 = x_2$ или $x_0 = -x_2 = x_6$, што није допуштено, јер све тачке морају бити различите. Дакле, мора да важи да је $x_2^2 = -x_0^2$. Међутим то није могуће ако су x_0 и x_2 реални различити бројеви, јер су тада и x_0^2 и x_2^2 ненегативни и бар један од њих је позитиван. Да ли је онда уопште могуће изабрати различите вредности x_0 и x_2 тако да је $x_2^2 = -x_0^2$? Јесте, ако допустимо да вредности x_i буду *комплексни бројеви*. Наиме, тражимо комплексан коефицијент k такав да је $x_2 = kx_0$ и $x_2^4 = x_0^4$ тј. $x_2^2 = (kx_0)^2 = k^2x_0^2 = -x_0^2$. Пошто је $k^2 = -1$, за k се може узети вредност имагинарне јединице i, за коју важи $i^2 = -1$. Заиста, ако је $x_2 = ix_0$, тада је $x_2^2 = (ix_0)^2 = i^2x_0^2 = -x_0^2$ и $x_2^4 = (x_2^2)^2 = (-x_0^2)^2 = x_0^4$. Бројеве x_3 и x_1 можемо изабрати тако да је $x_3 = ix_1$ из чега следи да је $x_3^2 = -x_1^2$ и $x_3^4 = x_1^4$. Након тога, израчунавање потребних вредности поинома P_0 се своди на израчунавање само 4 вредности полинома P_{00} и P_{01}:

$$\begin{array}{lll}
P_0(x_0^2) & = P_{00}(x_0^4) + x_0^2 P_{01}(x_0^4) & = (a_0 + a_4 x_0^4) + x_0^2(a_2 + a_6 x_0^4) \\
P_0(x_1^2) & = P_{00}(x_1^4) + x_1^2 P_{01}(x_1^4) & = (a_0 + a_4 x_1^4) + x_1^2(a_2 + a_6 x_1^4) \\
P_0(x_2^2) & = P_{00}(x_0^4) - x_0^2 P_{01}(x_0^4) & = (a_0 + a_4 x_0^4) + x_2^2(a_2 + a_6 x_0^4) \\
P_0(x_3^2) & = P_{00}(x_1^4) - x_1^2 P_{01}(x_1^4) & = (a_0 + a_4 x_1^4) + x_3^2(a_2 + a_6 x_1^4)
\end{array}$$

Претходни начин избора тачака x_2 и x_3 доноси сличну уштеду и приликом рачунања вредности полинома P_1. Овим се цео проблем своди на израчунавање следећих 8 вредности полинома P_{00}, P_{01}, P_{10} и P_{11}:

$$\begin{array}{ll|ll}
P_{00}(x_0^4) & = a_0 + a_4 x_0^4 & P_{01}(x_0^4) & = a_2 + a_6 x_0^4 \\
P_{00}(x_1^4) & = a_0 + a_4 x_1^4 & P_{01}(x_1^4) & = a_2 + a_6 x_1^4 \\
P_{10}(x_0^4) & = a_1 + a_5 x_0^4 & P_{11}(x_0^4) & = a_3 + a_7 x_0^4 \\
P_{10}(x_1^4) & = a_1 + a_5 x_1^4 & P_{11}(x_1^4) & = a_3 + a_7 x_1^4
\end{array}$$

Последња уштеда се добија тиме што се вредности x_0 и x_1 одаберу тако да је $x_1^8 = x_0^8$ тј. $x_1^4 = -x_0^4$. То се може постићи ако је $x_1 = kx_0$, где се коефицијент k одређује тако да је $k^4 = -1$. Пошто је $e^{i\pi} = -1$, један такав број је број $k = e^{i\frac{\pi}{4}} = \cos(\frac{\pi}{4}) + i\sin(\frac{\pi}{4}) = \frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}$. Подсетимо се Ојлерове формуле:

$$e^{i\phi} = \cos\phi + i\sin\phi$$

Важи $(e^{i\phi})^n = e^{in\phi}$, што одговара Моавровој[21] формули:

[21] Абрахам де Моавр (фр. Abraham de Moivre), (1667-1754), француски математичар.

$$(\cos\phi + i\sin\phi)^n = \cos(n\phi) + i\sin(n\phi)$$

Тада се претходно израчунавање своди на:

$$P_{00}(x_0^4) = a_0 + x_0^4 \cdot a_4 \qquad P_{00}(x_1^4) = a_0 - x_0^4 \cdot a_4$$
$$P_{01}(x_0^4) = a_2 + x_0^4 \cdot a_6 \qquad P_{01}(x_1^4) = a_2 - x_0^4 \cdot a_6$$
$$P_{10}(x_0^4) = a_1 + x_0^4 \cdot a_5 \qquad P_{10}(x_1^4) = a_1 - x_0^4 \cdot a_5$$
$$P_{11}(x_0^4) = a_3 + x_0^4 \cdot a_7 \qquad P_{11}(x_1^4) = a_3 - x_0^4 \cdot a_7$$

Могли бисмо рећи да се вредности ових линеарних полинома израчунавају комбиновањем полинома степена 0 (у питању су 8 полинома који одговарају коефицијентима a_k: $P_{000} = a_0$, $P_{001} = a_4$, $P_{010} = a_2$, $P_{011} = a_6$, $P_{100} = a_1$, $P_{101} = a_5$, $P_{110} = a_3$ и $P_{111} = a_7$) и чија се вредност тривијално израчунава (без даљих рекурзивних позива). Да би се све уклопило у општу схему, можемо рећи да се вредност тих константних 8 полинома израчунава у тачки x_0^8.

Ако резимирамо све односе између промењљивих које смо до сада успоставили, видимо да још једино можемо слободно да бирамо вредност x_0, док се све остале вредности тачака израчунавају на основу вредности x_0. Имамо низ тачака

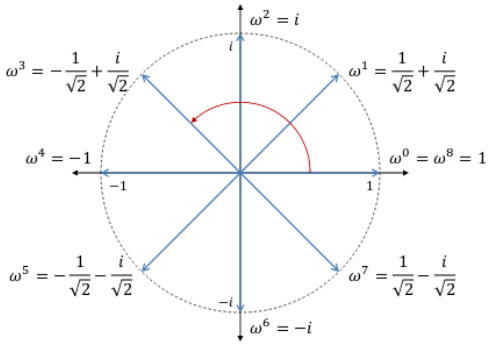
$$x_0 \qquad x_4 = -x_0$$
$$x_1 = (\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2})x_0 \qquad x_5 = -x_1 = (-\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2})x_0$$
$$x_2 = ix_0 \qquad x_6 = -x_2 = -ix_0$$
$$x_3 = ix_1 = (-\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2})x_0 \qquad x_7 = -x_3 = (\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2})x_0$$

Најједноставније формуле се добијају, наравно, ако се одабере $x_0 = 1$. Тада се може приметити да су тачке x_i осми корени из јединице. Надаље ћемо ове тачке обележавати са w_k уместо x_k (да бисмо нагласили да су у питању комплексни бројеви). Дакле, израчунаваћемо вредности полинома редом у тачкама $w_0 = 1 = e^0$, $w_1 = \frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2} = e^{i\frac{\pi}{4}}$, $w_2 = i = e^{i\frac{\pi}{2}}$, $w_3 = -\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2} = e^{i\frac{3\pi}{4}}$, $w_4 = -1 = e^{i\pi}$, $w_5 = -\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2} = e^{i\frac{5\pi}{4}}$, $w_6 = -i = e^{i\frac{3\pi}{2}}$, $w_7 = \frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2} = e^{i\frac{7\pi}{4}}$, тј. у тачкама

$$w_k = e^{\frac{k\pi i}{4}} = e^{\frac{2k\pi i}{8}}, \quad 0 \leq k < 8.$$

У општем случају ћемо израчунавати вредности полинома степена n (где се n бира тако да буде степен броја 2) и израчунаваћемо вредности полинома у тачкама w_k дефинисаним на следећи начин:

$$w_k = e^{\frac{2k\pi i}{n}}, \quad 0 \leq k < n$$



Слика 3.3: Визуелизација једног осмог примитивног корена из јединице ($e^{i\frac{\pi}{4}}$) и његових степена у комплексној равни.

На слици 3.3 је приказано да се сви бројеви w_k могу изразити као неки степен броја $w_1 = e^{\frac{2\pi i}{n}}$, који ћемо обележити са w. Број w је *примитивни n-ти корен из јединице* (у нашем примеру, осми), што значи да је $w^n = 1$, док је $w^k \neq 1$ за све $0 < k < n$. На пример, имагинарна јединица i је примитивни четврти корен из јединице јер је $i^1 = i \neq 1$, $i^2 = -1 \neq 1$, $i^3 = -i \neq 1$ и $i^4 = 1$, док број -1 јесте четврти корен из јединице (јер је $(-1)^4 = 1$), али није примитиван четврти корен, јер је $(-1)^2 = 1$ (-1 јесте примитивни други корен из јединице). Може се показати и да је реципрочна вредност броја w тј. број $w^{-1} = e^{\frac{-2\pi i}{n}}$ такође примитивни n-ти корен из јединице (што ће нам бити важно за инверзну Фуријеову трансформацији о којој ће бити речи касније). Може се показати да је сваки број облика $e^{ik\frac{2\pi}{n}}$, за узајамно просте бројеве k и n примитиван n-ти корен из јединице. Када је n степен броја 2, тада се примитивни n-ти корени добијају за све непарне вредности k. Сви бројеви w_k се могу изразити као степени броја w (као и било ког другог примитивног n-тог корена из јединице). Наиме, очигледно важи да је

$$w_k = e^{\frac{2k\pi i}{n}} = \left(e^{\frac{2\pi i}{n}}\right)^k = w^k.$$

Још једна важна чињеница у вези са примитивним n-тим коренима из јединице је следећа. Ако је w примитивни n-ти корен из јединице, за неки паран број $n > 0$, тада је w^2 примитивни $\frac{n}{2}$-ти корен из јединице (видећемо да се ово користи током рекурзивних позива у брзој Фуријеовој трансформацији). Заиста, важи:

$$w^2 = \left(e^{\frac{2\pi i}{n}}\right)^2 = e^{\frac{2\cdot 2\pi i}{n}} = e^{\frac{2\pi i}{\frac{n}{2}}}$$

Нагласимо да у извођењима попут претходних треба бити јако обазрив. Наиме, користили смо неколико пута закон $(a^b)^c = a^{bc}$ који не мора бити тачан за произвољне комплексне (па чак ни реалне бројеве). На пример, $1 = \sqrt{1} = \sqrt{(-1)^2} = ((-1)^2)^{\frac{1}{2}} \neq (-1)^{2\cdot\frac{1}{2}} = (-1)^1 = -1$. Ипак, може се показати да су претходна извођења коректна (пре свега јер је спољни изложилац степеновања увек био целобројан).

Пример 3.6.4

Илуструјмо сада поступак израчунавања, тако што ћемо га применити на одређивање вредности полинома $7x^7 + 6x^6 + 5x^5 + 4x^4 + 3x^3 + 2x^2 + 1x^1$, *тј. на вектор коефицијената* $[7, 6, 5, 4, 3, 2, 1, 0]$.

Полиноми степена 0 *су константни и њихова вредност не зависи од* x, *али да бисмо наГласили опште правило, претпоставићемо да се израчунавају само у тачки* $(w_0)^8 = (w_8)^0 = w^0 = 1$:

$$\begin{array}{llll}
P_{000}(w_8^0) & = P_{000}(1) & = a_0 & = 0,\\
P_{001}(w_8^0) & = P_{001}(1) & = a_4 & = 4,\\
P_{010}(w_8^0) & = P_{010}(1) & = a_2 & = 2,\\
P_{011}(w_8^0) & = P_{011}(1) & = a_6 & = 6,\\
P_{100}(w_8^0) & = P_{100}(1) & = a_1 & = 1,\\
P_{101}(w_8^0) & = P_{101}(1) & = a_5 & = 5,\\
P_{110}(w_8^0) & = P_{110}(1) & = a_3 & = 3,\\
P_{111}(w_8^0) & = P_{111}(1) & = a_7 & = 7
\end{array}$$

Полиноми степена 1 *(то су* P_{00}, P_{01}, P_{10} *и* P_{11}*) се израчунавају само у тачкама* $(w_0)^4 = (w_4)^0 = w^0 = 1$ *и* $(w_1)^4 = (w_4)^1 = w^4 = -1$:

$$
\begin{array}{llll}
P_{00}(w_4^0) &= P_{000}(w_8^0) + w_4^0 \cdot P_{001}(w_8^0) & \text{тј.} & P_{00}(1) = P_{000}(1) + 1 \cdot P_{001}(1) = 4, \\
P_{00}(w_4^1) &= P_{000}(w_8^0) - w_4^0 \cdot P_{001}(w_8^0) & \text{тј.} & P_{00}(-1) = P_{000}(1) - 1 \cdot P_{001}(1) = -4, \\
P_{01}(w_4^0) &= P_{010}(w_8^0) + w_4^0 \cdot P_{011}(w_8^0) & \text{тј.} & P_{01}(1) = P_{010}(1) + 1 \cdot P_{011}(1) = 8, \\
P_{01}(w_4^1) &= P_{010}(w_8^0) - w_4^0 \cdot P_{011}(w_8^0) & \text{тј.} & P_{01}(-1) = P_{010}(1) - 1 \cdot P_{011}(1) = -4, \\
P_{10}(w_4^0) &= P_{010}(w_8^0) + w_4^0 \cdot P_{011}(w_8^0) & \text{тј.} & P_{10}(1) = P_{100}(1) + 1 \cdot P_{101}(1) = 6, \\
P_{10}(w_4^1) &= P_{010}(w_8^0) - w_4^0 \cdot P_{011}(w_8^0) & \text{тј.} & P_{10}(-1) = P_{100}(1) - 1 \cdot P_{101}(1) = -4, \\
P_{11}(w_4^0) &= P_{010}(w_8^0) + w_4^0 \cdot P_{011}(w_8^0) & \text{тј.} & P_{11}(1) = P_{110}(1) + 1 \cdot P_{111}(1) = 10, \\
P_{11}(w_4^1) &= P_{010}(w_8^0) - w_4^0 \cdot P_{011}(w_8^0) & \text{тј.} & P_{11}(-1) = P_{110}(1) - 1 \cdot P_{111}(1) = -4,
\end{array}
$$

Полиноми степена 3 *(то су* P_0 *и* P_1*) се израчунавају само у тачкама* $(w_0)^2 = (w_2)^0 = w^0 = 1$, $(w_1)^2 = (w_2)^1 = w^2 = i$, $(w_2)^2 = i^2 = -1$ *и* $(w_3)^2 = (w_2)^3 = w^6 = -i$*:*

$$
\begin{array}{llll}
P_0(w_2^0) &= P_{00}(w_4^0) + w_2^0 \cdot P_{01}(w_4^0) & \text{тј.} & P_0(1) = P_{00}(1) + 1 \cdot P_{01}(1) = 12, \\
P_0(w_2^1) &= P_{00}(w_4^1) + w_2^1 \cdot P_{01}(w_4^1) & \text{тј.} & P_0(i) = P_{00}(-1) + i \cdot P_{01}(-1) = -4 - 4i, \\
P_0(w_2^2) &= P_{00}(w_4^0) - w_2^0 \cdot P_{01}(w_4^0) & \text{тј.} & P_0(-1) = P_{00}(1) - 1 \cdot P_{01}(1) = -4, \\
P_0(w_2^3) &= P_{00}(w_4^1) - w_2^1 \cdot P_{01}(w_4^1) & \text{тј.} & P_0(-i) = P_{00}(-1) - i \cdot P_{01}(-1) = -4 + 4i, \\
P_1(w_2^0) &= P_{10}(w_4^0) + w_2^0 \cdot P_{11}(w_4^0) & \text{тј.} & P_1(1) = P_{10}(1) + 1 \cdot P_{11}(1) = 16, \\
P_1(w_2^1) &= P_{10}(w_4^1) + w_2^1 \cdot P_{11}(w_4^1) & \text{тј.} & P_1(i) = P_{10}(-1) + i \cdot P_{11}(-1) = -4 - 4i, \\
P_1(w_2^2) &= P_{10}(w_4^0) - w_2^0 \cdot P_{11}(w_4^0) & \text{тј.} & P_1(-1) = P_{10}(1) - 1 \cdot P_{11}(1) = -4, \\
P_1(w_2^3) &= P_{10}(w_4^1) - w_2^1 \cdot P_{11}(w_4^1) & \text{тј.} & P_1(-i) = P_{10}(-1) - i \cdot P_{11}(-1) = -4 + 4i,
\end{array}
$$

На крају, полином степена 7 *(то је* P*) се израчунава у свим тачкама* $w_0 = (w_1)^0 = w^0 = 1$, $w_1 = (w_1)^1 = w^1 = w = \frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}$, $w_2 = (w_1)^2 = w^2 = i$, $w_3 = (w_1)^3 = w^3 = iw$, $w_4 = (w_1)^4 = w^4 = -1$, $w_5 = (w_1)^5 = w^5 = -w$, $w_6 = (w_1)^6 = w^6 = -i$ *и* $w_7 = (w_1)^7 = w^7 = -iw$*:*

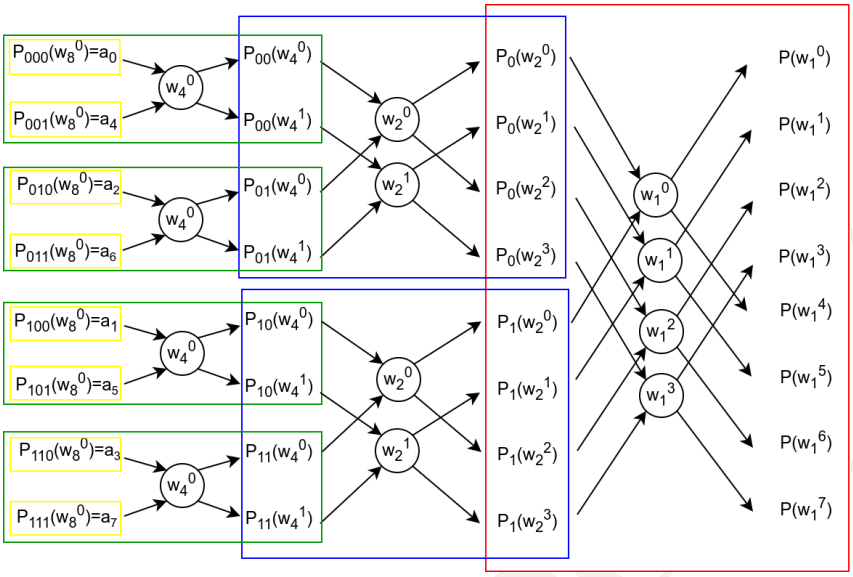
$$
\begin{array}{llll}
P(w_1^0) &= P_0(w_2^0) + w_1^0 \cdot P_1(w_2^0) & \text{тј.} & P(1) = P_0(1) + 1 \cdot P_1(1) = 28, \\
P(w_1^1) &= P_0(w_2^1) + w_1^1 \cdot P_1(w_2^1) & \text{тј.} & P(\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}) = P_0(i) + w \cdot P_1(i) = -4 - (4 + 4\sqrt{2})i, \\
P(w_1^2) &= P_0(w_2^2) + w_1^2 \cdot P_1(w_2^2) & \text{тј.} & P(i) = P_0(-1) + i \cdot P_1(-1) = -4 - 4i, \\
P(w_1^3) &= P_0(w_2^3) + w_1^3 \cdot P_1(w_2^3) & \text{тј.} & P(-\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}) = P_0(-i) + w^3 \cdot P_1(-i) = -4 + (4 - 4\sqrt{2})i, \\
P(w_1^4) &= P_0(w_2^0) - w_1^0 \cdot P_1(w_2^0) & \text{тј.} & P(-1) = P_0(1) - 1 \cdot P_1(1) = -4, \\
P(w_1^5) &= P_0(w_2^1) - w_1^1 \cdot P_1(w_2^1) & \text{тј.} & P(-\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2}) = P_0(i) - w \cdot P_1(i) = -4 - (4 - 4\sqrt{2})i, \\
P(w_1^6) &= P_0(w_2^2) - w_1^2 \cdot P_1(w_2^2) & \text{тј.} & P(-i) = P_0(-1) - i \cdot P_1(-1) = -4 + 4i, \\
P(w_1^7) &= P_0(w_2^3) - w_1^3 \cdot P_1(w_2^3) & \text{тј.} & P(\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}) = P_0(-i) - w^3 \cdot P_1(-i) = -4 + (4 + 4\sqrt{2})i,
\end{array}
$$

Ово израчунавање се схематски приказује коришћењем тзв. *„лептир” схеме* (слика 3.4). Један „лептир” описује израчунавање приказано на слици 3.5.

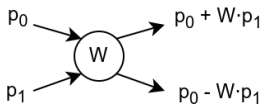
Израчунавање из претходног примера је приказано „лептир” схемом на слици 3.6.

На слици 3.4 се јасно види рекурзивна структура алгоритма (сваки рекурзивни позив је представљен једним обојеним правоугаоником). Структура рекурзивних позива приказана је и на слици 3.7.

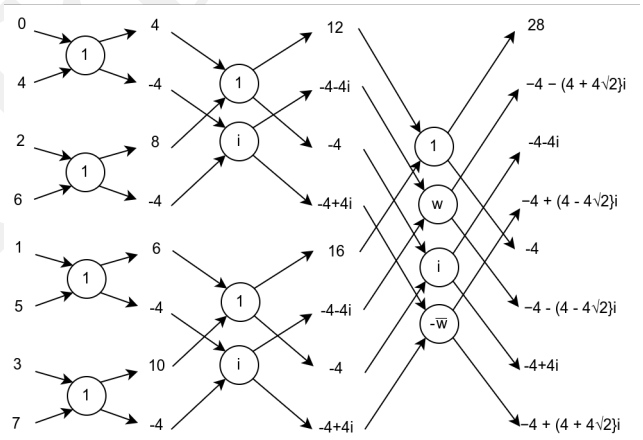
Сваки рекурзивни позив прима полином P_n тј. низ његових коефицијената a_i дужине n и број w који је неки степен полазног примитивног корена из јединице (а заправо



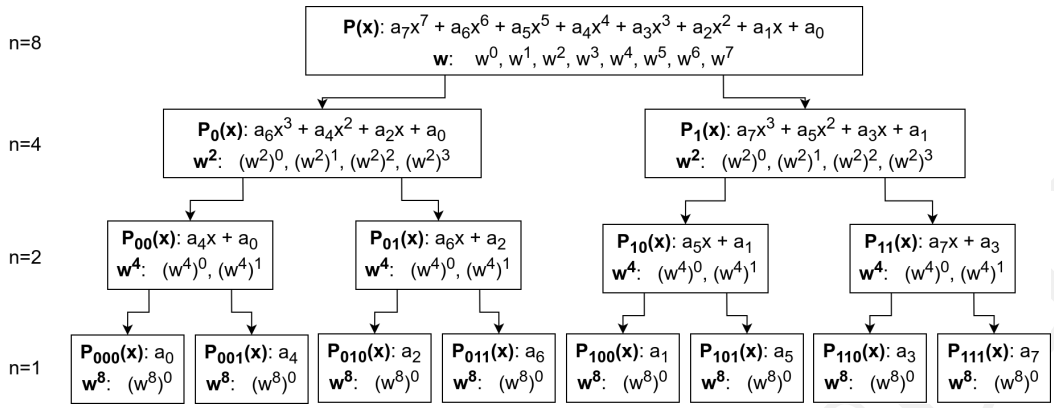
Слика 3.4: Лептир схема брзе Фуријеове трансформације.



Слика 3.5: Дефиниција „лептир” израчунавања.



Слика 3.6: Лептир схема брзе Фуријеове трансформације за пример полинома $P(x) = 7x^7 + 6x^6 + 5x^5 + 4x^4 + 3x^3 + 2x^2 + x$.



Слика 3.7: Структура рекурзивних позива.

је примитивни n-ти корен из јединице). Рекурзивни позив израчунава вредности полинома P_n у тачкама $w^0, w^1, \ldots w^{n-1}$. Ако је $n = 1$, излази се из рекурзије и резултат је једини коефицијент a_i. У супротном се врше два рекурзивна позива (за полиноме добијене од коефицијената на парним и непарним позицијама) и резултати се обједињавају на основу правила, чија коректност следи из наредне леме.

Лема 3.6.1

Нека је

$$P_n = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_0 = \sum_{j=0}^{n-1} a_j x^j,$$

полином степена n (за неки степен двојке $n = 2^k$) и нека је $w = e^{\frac{2\pi i}{n}}$ примитивни n-ти корен из јединице (важи $w^n = 1$). Нека је P_{n0} полином који се добија издвајањем коефицијената уз парне степене тј.

$$P_{n0} = a_{n-2}x^{\frac{n}{2}-1} + \ldots + a_2 x + a_0 = \sum_{j=0}^{\frac{n}{2}-1} a_{2j} x^j,$$

а P_{n1} полином који се добија издвајањем коефицијената уз непарне степене тј.

$$P_{n1} = a_{n-1}x^{\frac{n}{2}-1} + \ldots + a_3 x + a_1 = \sum_{j=0}^{\frac{n}{2}-1} a_{2j+1} x^j.$$

Оба та полинома су степена $\frac{n}{2}$.

За $0 \leq k < \frac{n}{2}$ *важи:*

$$\begin{aligned} P_n(w^k) &= P_{n0}((w^2)^k) + w^k P_{n1}((w^2)^k) \\ P_n(w^{k+\frac{n}{2}}) &= P_{n0}((w^2)^k) - w^k P_{n1}((w^2)^k). \end{aligned}$$

Доказ. Запишимо вредности сва три полинома из прве једнакости у одговарајућим тачкама:

$$\begin{aligned} P_n(w^k) &= \sum_{j=0}^{n-1} a_j \cdot (w^k)^j \\ P_{n0}((w^2)^k) &= \sum_{j=0}^{\frac{n}{2}-1} a_{2j} \cdot ((w^2)^k)^j = \sum_{j=0}^{\frac{n}{2}-1} a_{2j} \cdot w^{2kj} \\ P_{n1}((w^2)^k) &= \sum_{j=0}^{\frac{n}{2}-1} a_{2j+1} \cdot ((w^2)^k)^j = \sum_{j=0}^{\frac{n}{2}-1} a_{2j+1} \cdot w^{2kj} \end{aligned}$$

Зато је

$P_{n0}((w^2)^k) + w^k P_{n1}((w^2)^k) = \sum_{j=0}^{\frac{n}{2}-1} a_{2j} \cdot w^{k\cdot 2j} + \sum_{j=0}^{\frac{n}{2}-1} a_{2j+1} \cdot w^{k\cdot(2j+1)} = \sum_{j=0}^{n-1} a_w^{kj} = P_n(w^k)$

Нагласимо да смо у претходном извођењу користили искључиво целобројне изложиоце (бројеве 2, i и k) и може се лако показати да у том случају важе сви уобичајени закони степеновања.

Докажимо сада да је $w^{k+\frac{n}{2}} = -w^k$.

$$w^{k+\frac{n}{2}} = w^k \cdot w^{\frac{n}{2}} = w^k \cdot \left(e^{\frac{2\pi i}{n}}\right)^{\frac{n}{2}} = w^k \cdot e^{\frac{2\pi i}{n}\cdot\frac{n}{2}} = w^k \cdot e^{\pi i} = w^k \cdot (-1) = -w^k$$

Нагласимо да је n паран број, па се у претходном извођењу све време ради са целобројним изложиоцима.

Зато је

$$\begin{aligned} P(w^{k+\frac{n}{2}}) = P(-w^k) &= \sum_{j=0}^{n-1} a_j \cdot (-w^k)^j \\ &= \sum_{j=0}^{n-1} a_j \cdot (-1)^j \cdot w^{kj} \\ &= \sum_{j=0}^{\frac{n}{2}-1} a_{2j} \cdot w^{k\cdot 2j} + \sum_{j=0}^{\frac{n}{2}-1} (-1) \cdot a_{2j+1} \cdot w^{k\cdot(2j+1)} \\ &= \sum_{j=0}^{\frac{n}{2}-1} a_{2j} \cdot w^{2kj} - w^k \sum_{j=0}^{\frac{n}{2}-1} a_{2j+1} \cdot w^{2kj} \\ &= P_{n0}((w^2)^k) - w^k P_{n1}((w^2)^k) \qquad \square \end{aligned}$$

Опис и имплементација алгоритма брзе Фуријеове трансформације дати су у поглављу 3.6.3.

3.6.2 Инверзна Фуријеова трансформација

Брза Фуријеова трансформација решава само пола проблема: на основу коефицијената полинома степена n одређују се његове вредности у n специјално одабраних тачака тј. врши се табелирање полинома. Остаје питање како да на основу вредности полинома у тим тачкама одредимо његове коефицијенте, тј. како да ефикасно извршимо интерполацију. Испоставља се да је проблем интерполације врло сличан проблему табелирања и да га решава практично исти алгоритам.

Приметимо прво да се табелирање, тј. израчунавање вредности полинома у тачкама $1, w, \ldots, w^{n-1}$ може представити и матрично.

$$\begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & w & w^2 & \ldots & w^{n-1} \\ 1 & w^2 & w^{2\cdot 2} & \ldots & w^{2\cdot(n-1)} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & w^{n-1} & w^{(n-1)\cdot 2} & \ldots & w^{(n-1)\cdot(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \ldots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} P(1) \\ P(w) \\ \ldots \\ P(w^{n-1}) \end{pmatrix}$$

Уведимо следеће ознаке:

- вектор коефицијената полинома P обележимо са $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})^T$,

- вектор вредности полинома са $\mathbf{v} = (P(1), P(w), \ldots, P(w^{n-1}))^T$,
- Вандермондову матрицу која садржи степене броја w обележимо са $V(w)$.

Ако су задати коефицијенти полинома $\mathbf{a}$, његове вредности $\mathbf{v}$ у n тачака $1, w, \ldots, w^{n-1}$ добијају се израчунавањем производа:

$$\mathbf{v} = V(w)\mathbf{a}.$$

Овај производ се наивним алгоритмом множења матрице и вектора извршава у времену $O(n^2)$, а директном брзом Фуријеовом трансформацијом у времену $O(n \log n)$. Приметимо да је, пошто се може представити помоћу множења матрицом, дискретна Фуријеова трансформација линеарна трансформација.

С друге стране, ако су задате вредности полинома $\mathbf{v} = (P(1), P(w), \ldots, P(w^{n-1}))^T = (v_0, v_1, \ldots, v_{n-1})^T$, а потребно је израчунати његове коефицијенте $\mathbf{a}$, матрична једнакост постаје систем линеарних једначина по непознатим коефицијентима $\mathbf{a}$. Решавање система једначина своди се на рачунање инверзне матрице $V(w)^{-1}$, што се увек може урадити Гаусовом елиминацијом, што је алгоритам сложености $O(n^3)$. С друге стране, ако би се матрица $V(w)^{-1}$ унапред знала, систем би се могао решити множењем једнакости $\mathbf{v} = V(w)\mathbf{a}$ с леве стране матрицом $V(w)^{-1}$:

$$\mathbf{a} = V(w)^{-1}\mathbf{v}.$$

У општем случају, ово је алгоритам сложености $O(n^2)$. Приметимо, међутим, да Вандермондова матрица $V(\omega)$ има правилну структуру која се може искористити и за ефикасније израчунавање њој инверзне матрице и за ефикасније множење вектора њоме.

Лема 3.6.2

Инверзна матрица матрице $V(w)$ Фуријеове трансформације једнака је

$$V(w)^{-1} = \frac{1}{n}V(w^{-1}).$$

Доказ. Доказаћемо да важи да је $V(w)V(w^{-1}) = nI$, где је са w^{-1} означена реципрочна вредност броја w тј. број $e^{\frac{-2\pi i}{n}}$, а са I је означена јединична матрица димензије n. Заиста, ако је $r \neq s$, онда је за $r, s = 0, \ldots, n-1$ производ $(r+1)$-е врсте матрице $V(w)$ и $(s+1)$-е колоне матрице $V(w^{-1})$ једнак

$$\sum_{k=0}^{n-1} w^{rk} w^{-sk} = \sum_{k=0}^{n-1} w^{(r-s)k} = \frac{1 - w^{n(r-s)}}{1 - w^{r-s}} = 0,$$

јер је $w^n = 1$.

Ако је пак $r = s$, онда је тај производ једнак

$$\sum_{k=0}^{n-1} w^{rk} w^{-rk} = \sum_{k=0}^{n-1} 1 = n.$$

Из $V(w)V(w^{-1}) = nI$, дељењем са n следи $V(w)^{-1} = \frac{1}{n} V(w^{-1})$, чиме је теорема доказана. □

Решавање система једначина $\mathbf{v} = V(w)\mathbf{a}$ своди се, дакле, на израчунавање производа $\mathbf{a} = \frac{1}{n} V(w^{-1})\mathbf{v}$. Посао се даље поједностављује захваљујући следећој теореми.

Лема 3.6.3

Ако је w примитивни n-ти корен из јединице, онда је w^{-1} такође примитивни n-ти корен из јединице.

Пошто брза Фуријеова трансформација ефикасно рачуна производ неке Вандермондове матрице и вектора, производ $\frac{1}{n} V(w^{-1})\mathbf{v}$ може се израчунати њеном применом, замењујући полазну вредност w са w^{-1}, израчунавањем производа $V(w^{-1})\mathbf{v}$ и затим дељењем компоненти добијеног вектора са n. Ова трансформација зове се *инверзна Фуријеова трансформација* (енгл. inverse Fourier transform).

Напоменимо да се у применама обраде сигнала терминологија разликује тако што се мења улога директне и инверзне трансформације.

Узимајући све до сада речено у обзир, производ полинома P и Q степена $n-1$ (за $n = 2^k$) може се израчунати коришћењем $O(n \log n)$ операција (са комплексним бројевима) на следећи начин:

- рачунамо вредност полинома P и Q у $2n-1$ тачака (степени $2n$-тог примитивног корена из јединице w) брзом Фуријеовом трансформацијом,
- у свакој од тачака множимо вредност полинома P и Q,
- рачунамо брзу Фуријеову трансформацију добијеног вектора, при чему уместо вредности w користимо вредност w^{-1} и резултат множимо са $\frac{1}{2n}$.

3.6.3 Алгоритам брзе Фуријеове трансформације

Прикажимо сада алгоритам брзе Фуријеове транформације, најпре у псеудокоду (алгоритам 10), а затим и у језику C++.

Алгоритам 10 Брза Фуријеова трансформација

Input: n – broj koeficijenata polinoma P
Input: $a_0, \dots, a_{n-1}$ – koeficijenti polinoma P
Input: w - primitivni n-ti koren iz jedinice
Output: vrednosti $P_0, \dots, P_{n-1}$ polinoma P na skupu tačaka $1, w, \dots, w^{n-1}$
procedure FFT(n, $[a_0, a_1, \dots a_{n-1}]$, w)
    if $n = 1$ **then**
        $P_0 = a_0$
    else
        P^{parno} = FFT($n/2$, $[a_0, a_2, \dots, a_{n-2}]$, w^2)
        $P^{neparno}$ = FFT($n/2$, $[a_1, a_3, \dots, a_{n-1}]$, w^2)
        for $k = 0$ to $n/2 - 1$ **do**
            $P_k = P_k^{parno} + w^k \cdot P_k^{neparno}$
            $P_{k+n/2} = P_k^{parno} - w^k \cdot P_k^{neparno}$

Јасно је да алгоритам реализован претходном процедуром задовољава једначину $T(n) = 2T(n/2) + O(n)$, па му је сложеност $O(n \log n)$. Након трансформације (промене репрезентације израчунавањем вредности) два полинома њихово множење је могуће у времену $O(n)$, па је укупна сложеност множења полинома помоћу алгоритма FFT једнака $O(n \log n)$.

Инверзна трансформација се може израчунати коришћењем претходног алгоритма тако што се у главном позиву функције уместо аргумента w зада аргумент w^{-1} и тако што се на крају сваки елемент резултата подели са n.

Размотримо сада C++ имплементацију. У језику C++ комплексне бројеве имамо на располагању у облику типова `complex<double>` и `complex<float>` (запис у двострукој и једнострукој тачности). Директан начин имплементације је следећи.

```
typedef complex<double> Complex;
typedef vector<Complex> ComplexVector;

// brza Furijeova transformacija vektora a duzine n=2^k
// bool parametar inverse odredjuje da li je direktna ili inverzna
ComplexVector fft(const ComplexVector& a, bool inverzna) {
  // broj koeficijenata polinoma
```

```
  int n = a.size();

  // ako je stepen polinoma 0, vrednost u svakoj tacki jednaka
  // je jedinom koeficijentu
  if (n == 1)
    return ComplexVector(1, a[0]);

  // izdvajamo koeficijente na parnim i na neparnim pozicijama
  ComplexVector Pparno(n / 2), Pneparno(n / 2);
  for (int i = 0; i < n / 2; i++) {
    Pparno[i] = a[2 * i];
    Pneparno[i] = a[2 * i + 1];
  }

  // rekurzivno izracunavamo Furijeove transformacije tih polinoma
  ComplexVector fftParno   = fft(Pparno, inverzna),
                fftNeparno = fft(Pneparno, inverzna);

  // objedinjujemo rezultat
  ComplexVector rezultat(n);
  for (int k = 0; k < n / 2; k++) {
    // odredjujemo primitivni n-ti koren iz jedinice
    double koeficijent = inverzna ? -1.0 : 1.0;
    Complex w = exp((koeficijent * 2 * k * M_PI / n) * 1i);
    // racunamo vrednost polinoma u toj tacki
    rezultat[k] = fftParno[k] + w * fftNeparno[k];
    rezultat[k + n/2] = fftParno[k] - w * fftNeparno[k];
  }
  // vracamo konacan rezultat
  return rezultat;
}

// funkcija vrsi direktnu Furijeovu transformaciju polinoma ciji su
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector fft(const ComplexVector& a) {
  return fft(a, false);
}

// funkcija vrsi inverznu Furijeovu transformaciju polinoma ciji su
```

```
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector ifft(const ComplexVector& a) {
  ComplexVector rezultat = fft(a, true);
  // nakon izracunavanja vrednosti, potrebno je jos podeliti
  // sve koeficijente duzinom vektora
  int n = a.size();
  for (int k = 0; k < n; k++)
    rezultat[k] /= n;
  return rezultat;
}
```

Претходну имплементацију је могуће побољшати. Највећи проблем је то што се n-ти корени из јединице рачунају засебно у сваком рекурзивном позиву. Ефикасност се значајно поправиља ако се низ корена израчунава само једном, пре функције. Проблем је у томе што се током рекурзије n смањује, па су нам у сваком позиву потребни различити корени. Међутим, ако је неки број k-ти корен из јединице, онда је он и $2k$-ти корен из јединице. Зато, ако знамо низ n-тих корена из јединице ($e^{\frac{2k\pi i}{n}}$, за k од 0 до $n-1$), тада су елементи на парним позицијама тог вектора $n/2$-ти корени из јединице. Заиста, ако је $k = 2k'$, тада је $e^{\frac{2k\pi i}{n}} = e^{\frac{2k'\pi i}{\frac{n}{2}}}$ и важи да ако је $0 \leq k < n$, тада је $0 \leq k' < n/2$. Дакле, у почетку можемо израчунати низ свих n-тих корена из јединице и њих користити на почетном нивоу рекурзије, на наредном нивоу рекурзије ћемо користити сваки други елемент тог вектора, на наредном сваки четврти и тако даље. На пример, ако је $n = 8$, почетни низ корена је 1, $\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}$, i, $-\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}$, -1, $-\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2}$, $-i$, $\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2}$, на наредном нивоу рекурзије је $n = 4$ и користе се корени 1, i, -1, $-i$, на наредном нивоу рекурзије је $n = 2$ и користе се корени 1 и -1, а на последњем нивоу рекурзије је $n = 1$ и излази се из рекурзије без коришћења ових корена.

Још један проблем претходне имплементације је то што се током рекурзије алоцирају и попуњавају помоћни вектори, што доводи до губитка и времена и меморије. Фуријеову трансформацију је могуће реализовати и без коришћења помоћне меморије. На почетном нивоу рекурзије коефицијенти улазног полинома су сви дати почетним векторима коефицијената. На наредном се посматрају елементи на позицијама $0, 2, 4, \ldots, n-2$ и на позицијама $1, 3, \ldots, n-1$. На наредном се посматрају елементи на позицијама $0, 4, 8, n-4$, затим елементи на позицијама $1, 5, \ldots, n-3$, затим елемени на позицијама $2, 6, \ldots, n-2$, и на крају елементи на позицијама $3, 7, \ldots, n-1$. Слично се наставља и на даљим нивоима рекурзије. Дакле, уместо формирања помоћног улазног вектора са погодно одабраним улазним коефицијентима, прослеђиваћемо оригинални вектор, позицију почетка s и померај d и посматраћемо његове елементе на позицијама $s + dk$, за $0 \leq k < n$, где је $n = n_0/d$, а n_0 је дужина почетног вектора. Резултате рекурзив-

них позива можемо сместити у две половине резултујућег низа. Након тога резултате обједињујемо. Вредности на позицијама k и $k + n/2$ резултујућег вектора одређене су вредностима на позицији k у резултату првог и другог рекурзивног, међутим, оне се налазе управо на позицијама k и $k+n/2$ резултујућег вектора (јер смо резултате рекурзивних позива сместили у прву и другу половину резултата). Морамо водити рачуна да те две вредности морамо истовремено израчунати и ажурирати.

```
typedef complex<double> Complex;
typedef vector<Complex> ComplexVector;

// funkcija vrši Furijeovu transformaciju (direkntu ili inverznu) elemenata
//   a[start_a], a[start_a + korak], a[start_a + 2korak], ...
// i rezultat smešta u niz
//   rezultat[start_rezultat], rezultat[start_rezultat + 1], ...
// koristeci primitivne korene iz jedinice smestene u niz
//   w[0], w[korak], w[2*korak], ...
void fft(const ComplexVector& a, int start_a,
         const ComplexVector& w,
         ComplexVector& rezultat, int start_rezultat,
         int korak) {
  // broj elemenata niza koji se transformise
  int n = a.size() / korak;

  // stepen polinoma je nula, pa mu je vrednost u svakoj tacki jednaka
  // konstantnom koeficijentu
  if (n == 1) {
    rezultat[start_rezultat] = a[start_a];
    return;
  }

  // rekurzivno transformisemo niz koeficijenata na parnim pozicijama
  // smestajuci rezultat u prvu polovinu niza rez
  fft(a, start_a, w, rezultat, start_rezultat, korak*2);
  // rekurzivno transformisemo niz koeficijenata na neparnim pozicijama
  // smestajuci rezultat u drugu polovinu niza rez
  fft(a, start_a + korak, w, rezultat, start_rezultat + n/2, korak*2);

  // objedinjujemo dve polovine u rezultujuci niz
  for (int k = 0; k < n/2; k++) {
```

```
    Complex r1 = rezultat[start_rezultat + k];
    Complex r2 = rezultat[start_rezultat + (k + n/2)];
    rezultat[start_rezultat + k] = r1 + w[k*korak] * r2;
    rezultat[start_rezultat + (k + n/2)] = r1 - w[k*korak] * r2;
  }
}

// funkcija vrsi direktnu Furijeovu transformaciju polinoma ciji su
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector fft(const ComplexVector& a) {
  // duzina niza koeficijenata polinoma
  int n = a.size();
  // izracunavamo primitivne n-te korene iz jedinice
  ComplexVector w(n/2);
  for (int k = 0; k < n/2; k++)
    w[k] = exp((2 * k * M_PI / n) * 1i);
  // vektor u koji se smesta rezultat
  ComplexVector rezultat(n);
  // vrsimo transformaciju
  fft(a, 0, w, rezultat, 0, 1);
  // vracamo dobijeni rezultat
  return rezultat;
}

// funkcija vrsi inverznu Furijeovu transformaciju polinoma ciji su
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector ifft(const ComplexVector& a) {
  // duzina niza koeficijenata polinoma
  int n = a.size();
  // izracunavamo primitivne n-te korene iz jedinice
  ComplexVector w(n);
  for (int k = 0; k < n; k++)
    w[k] = exp((- 2 * k * M_PI / n) * 1i);
  // vektor u koji se smesta rezultat
  ComplexVector rezultat(n);
  // vrsimo transformaciju
  fft(a, 0, w, rezultat, 0, 1);
  // popravljamo rezultat
  for (int i = 0; i < n; i++)
```

```
    rezultat[i] /= n;
  // vracamo dobijeni rezultat
  return rezultat;
}

vector<double> proizvod(const vector<double>& p1,
                        const vector<double>& p2) {
  // duzina niza koeficijenata
  int n = p1.size();
  // kreiramo nizove kompleksnih koeficijenata dopunjavajuci ih nulama
  // do dvostruke duzine
  int N = 2*n;
  ComplexVector a(N, 0.0);
  copy(begin(p1), end(p1), begin(a));
  ComplexVector b(N, 0.0);
  copy(begin(p2), end(p2), begin(b));

  // vrsimo Furijeove transformacije oba vektora koeficijenata
  // (slozenost je O(n log(n)))
  ComplexVector va = fft(a), vb = fft(b);
  // mnozimo vrednosti u pojedinacnim tackama (slozenost je O(n))
  ComplexVector vc(N);
  for (int i = 0; i < N; i++)
    vc[i] = va[i] * vb[i];
  // inverznom Furijeovom transformacijom rekonstruisemo koeficijente
  // proizvoda (slozenost je O(n log(n))
  ComplexVector c = ifft(vc);

  // realne delove kompleksnih brojeva smestamo u niz rezultat i vracamo ga
  vector<double> rezultat(N);
  transform(begin(c), end(c), begin(rezultat),
            [](Complex x) { return real(x); });
  return rezultat;
}
```

Пажљивом анализом је могуће уклонити рекурзију из претходне имплементације (тако се добија Кули-Тукијев нерекурзивни алгоритам за FFT, заснован на раније приказаној лептир-схеми). Та имплементација неће бити приказана у овом уџбенику.

3.6.4 NTT: Фуријеова трансформација у модуларној аритметици

У алгоритму FFT прелаз са реалних бројева на комплексне је био неопходан зато што комплексни бројеви гарантују то да за свако n постоји примитиван n-ти корен из јединице, тј. то да свака једначина облика $w^n = 1$ има n различитих решења, што код реалних бројева није случај већ за $n > 2$. Као ни реални, ни комплексни бројеви не могу бити потпуно прецизно представљени у рачунару, већ се користи приближан запис (најчешће запис у покретном зарезу), што може довести до рачунских грешака. Уместо комплексних бројева могуће је користити и неке друге бројевне структуре. На пример, могуће је користити модуларну аритметику и коначна поља.

У модуларној аритметици n-ти корен из јединице по модулу p је број w такав да важи $w^n \equiv 1 \pmod{p}$. Ако за свако $1 \leq m < n$ важи $w^m \not\equiv 1 \pmod{p}$, тада је w примитиван n-ти корен из јединице. Природно се поставља питање да ли за дати прост број p и дати степен n постоји примитивни n-ти корен из јединице.

Лема 3.6.4 **[Постојање примитивног n-тог корена из јединице]**

За прост број p у мултипликативној модуларној групи $\mathbb{Z}_p^\times$, за $2 \leq n < p$, примитивни n-ти корен из јединице постоји ако и само ако је $p-1$ дељив бројем n.

Доказ. Претпоставимо да је w примитивни n-ти корен из јединице. Нека је $p - 1 = nq + r$ и $r < n$. Тада је $w^{p-1} = w^{nq+r} = (w^n)^q \cdot w^r$. Међутим, пошто је w n-ти корен из јединице важи $w^n \equiv 1 \pmod{p}$, па је $(w^n)^q \cdot w^r \equiv w^r \pmod{p}$. На основу мале Фермаове теореме важи $w^{p-1} \equiv 1 \pmod{p}$, па мора да важи $w^r \equiv 1 \pmod{p}$. Међутим, пошто је n примитивни n-ти корен, ово не може да важи ни за један број $r < n$, осим за $r = 0$. Зато је $p - 1 = nq$, па је $p - 1$ дељив бројем n.

Претпоставимо сада да је $p - 1$ дељив бројем n, тј. да је $p - 1 = qn$. За сваки број $a \in Z_p \setminus \{0\}$, на основу мале Фермаове теореме важи $a^{p-1} \equiv 1 \pmod{p}$, тј. $(a^q)^n \equiv 1 \pmod{p}$. Број a^q је, дакле, увек n-ти корен из јединице, али не мора бити примитиван.

Да би a^q био примитиван n-ти корен, потребно је пронаћи генератор групе, односно број a чији је ред $p - 1$, тј. елемент $a \in Z_P \setminus \{0\}$ такав да је $a^{p-1} \equiv 1 \pmod{p}$ (што увек важи), али ни за једно $1 \leq m < p - 1$ не важи $a^m \equiv 1 \pmod{p}$. На основу леме 3.4.5, група $\mathbb{Z}_p^\times$ је цикличка и генератор увек постоји.

Примитивни n-ти корен из јединице по модулу p можемо добити као $a^q \bmod p = a^{\frac{p-1}{n}} \bmod p$. Заиста, важи да је $(a^q)^n = a^{p-1} \equiv 1 \pmod{p}$. Ако би за неко $m < n$ важило $(a^q)^m \equiv 1 \pmod{p}$, тада би важило $a^{qm} \equiv 1 \pmod{p}$ и ред елемента a био би мањи или једнак $qm < qn = p - 1$, што је у супротности са избором елемента a (који је изабран као елемент реда $p - 1$, тј. елемент такав да за свако $1 \leq k < p - 1$ важи $a^k \not\equiv 1 \pmod{p}$). □

Пример 3.6.5

У групи $\mathbb{Z}_7^\times$ постоји примитивни n-ти корен из јединице за $n = 2$, $n = 3$ и $n = 6$, јер су то једини делиоци броја $p - 1 = 6$. У примеру 3.4.8 смо видели да су генератори групе $a = 3$ и $a = 5$.

Примитиван други корен из јединице је $a^{\frac{p-1}{n}} \bmod p$, што за $a = 3$ даје елемент $3^3 \bmod 7 = 6$. Заиста, $6^2 \bmod 7 = 1$, док је $6^1 \bmod 7 = 6 \neq 1$. За $a = 5$, поново се добија корен $5^3 \bmod 7 = 6$.

Примитиван трећи корен из јединице за $a = 3$ је $3^2 \bmod 7 = 2$. Заиста, $2^3 \bmod 7 = 1$, док је $2^1 \bmod 7 = 2 \neq 1$ и $2^2 \bmod 7 = 4 \neq 1$. За $a = 5$ добија се корен $5^2 \bmod 7 = 4$.

Примитиван шести корен из јединице за $a = 3$ је $3^1 \bmod 7 = 3$, док се за $a = 5$ добија $5^1 \bmod 7 = 5$.

Из претходне леме следи да у $\mathbb{Z}_p^\times$ примитивни n-ти корени не постоје за свако n. Међутим, имамо слободу избора модула p на основу познатог степена полинома n. Пошто нас занимају вредности n које су степени двојке потребно је да се пронађе неки прост број p такав да је $p - 1$ дељиво са што више степена двојке (онда тај исти број можемо користити за различите вредности n које су степени двојке). Један такав прост број је $p = 2^{23} \cdot 7 \cdot 17 + 1 = 998\,244\,353$ и он има примитивни n-ти корен за све степене двојке n до 2^{23} (што је у многим применама и више него довољно, јер нам омогућава множење полинома са по око 8 милиона коефицијената који су у опсегу до скоро милијарду).

Остаје питање како за дату вредност n пронаћи примитивни n-ти корен из јединице по модулу p. Покажимо један ефективни поступак за то. Претпоставимо да је $p - 1 = kn$. Из доказа леме 3.6.4 јасно је да је кључно пронаћи генератор a групе $\mathbb{Z}_p^\times$. тј. елемент $a \in Z_p \setminus \{0\}$ такав да је $a^{p-1} \equiv 1 \pmod p$ (што увек важи), али ни за једно $1 \leq m < p - 1$ не важи $a^m \equiv 1 \pmod p$. Да би овај услов важио за неко m потребно је да m буде делилац броја $p - 1$. Довољно је, дакле, да проверимо да је $a^m \not\equiv 1 \pmod p$ за све делиоце m броја $p - 1$. Ово се може даље редуковати тиме што се испитају само они делиоци који су облика $\frac{p-1}{f_i}$, где је f_i неки прост чинилац броја $p - 1$. Наиме, ови делиоци су максимални у мрежи дељивости (за сваки од њих, наредни делилац је управо $p - 1$) и ако за неки делилац m испред њих важи $a^m \equiv 1 \pmod p$, тада ће бар неки од њих, тј. бар неки делилац облика $\frac{p-1}{f_i}$ бити дељив са m и важиће $a^{\frac{p-1}{f_i}} \equiv 1 \pmod p$.

Пример 3.6.6

Да бисмо за број $p = 2^{23} \cdot 7 \cdot 17 + 1 = 998\,244\,353$ одредили број a чији је ред $p - 1$, довољно је да испробавамо редом вредности a (на пример, можемо размотрати редом

мале просте бројеве) и да пронађемо први број a такав да је $a^{2^{22}\cdot 7\cdot 17} \not\equiv 1 \pmod p$, $a^{2^{23}\cdot 17} \not\equiv 1 \pmod p$ и $a^{2^{23}\cdot 7} \not\equiv 1 \pmod p$. Први такав број је $a = 3$.

Када је познат примитивни n-ти корен из јединице w, извођење Фуријеове трансформације тече на потпуно исти начин, једино што се уместо комплексних бројева користе природни бројеви и све операције се извршавају по модулу p. Овај облик Фуријеове трансформације се некада назива NTT (енгл. number theoretic transform). Она може да има предности у односу на класичан приступ са комплексним бројевима, јер не постоје проблеми са прецизношћу записа броја.

Пример 3.6.7

Прикажимо израчунавање производа полинома $P(x) = 1 + x + x^2$ и $Q(x) = 3 + 5x$ применом NTT за вредност $p = 998\,244\,353$ (наравно, пошто су полиноми малог степена, са малим коефицијентима, могли бисмо користити и неку мању вредност за p). Коефицијенте полинома допуњавамо тако да се добију вектори чија је дужина степен двојке и довољна је да се смести производ који је степена 3, па добијамо два вектора дужине $n = 4$: $[1, 1, 1, 0]$ и $[3, 5, 0, 0]$. Вредност k једнака је $(p-1)/n = 249\,561\,088$, па је примитивни n-ти корен из јединице једнак $w = a^k \bmod p = 911\,660\,635$.

Фуријеова трансформација може добити множењем матрицом (наравно, може и брже, коришћењем алгоритма FFT):

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & w & w^2 & w^3 \\ 1 & w^2 & w^4 & w^6 \\ 1 & w^3 & w^6 & w^9 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 911\,660\,635 & 998\,244\,352 & 86\,583\,718 \\ 1 & 998\,244\,352 & 1 & 998\,244\,352 \\ 1 & 86\,583\,718 & 998\,244\,352 & 911\,660\,635 \end{pmatrix}$$

Множењем ове матрице векторима полинома (по модулу p), добијају се њихове Фуријеове трансформације: $[3,\ 911\,660\,635,\ 1,\ 86\,583\,718]$ *и* $[8,\ 565\,325\,766,\ 998\,244\,351,\ 432\,918\,593]$. *Њих множимо елемент по елемент (по модулу p) и добијамо вектор:* $[24,\ 738\,493\,194,\ 998\,244\,351,\ 259\,751\,149]$.

Еуклидовим алгоритмом се може одредити да је модуларни инверз броја w једнак $86\,583\,718$. *Матрица инверзне Фуријеове трансформације је зато:*

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 86\,583\,718 & 998\,244\,352 & 911\,660\,635 \\ 1 & 998\,244\,352 & 1 & 998\,244\,352 \\ 1 & 911\,660\,635 & 998\,244\,352 & 86\,583\,718 \end{pmatrix}$$

Множењем ове матрице вектором $[24,\ 738\,493\,194,\ 998\,244\,351,\ 259\,751\,149]$ *(по модулу* p*) и скраћивањем добијеног резултата са* $n = 4$ *(што је дужина вектора), добија се вектор* $[3, 8, 8, 5]$ *који представља полином* $3 + 8x + 8x^2 + 5x^3$*, што је заиста производ наша два полазна полинома.*

Задатак: Поравнавање ниски

Ниске s и t се састоје од слова и тачака. Ниска t је краћа од ниске s и може се поравнати са ниском s тако што се њен први карактер постави испод било ког карактера ниске s тако да се цела ниска t пореди са подниском ниске s која почиње од тог карактера. На пример, ниске s и t које су једнаке `ab.c.b` и `a.b` се могу поравнатни на следеће начине:

```
ab.c.b.  ab.c.b.  ab.c.b.   ab.c.b.
a.b.       a.b.      a.b.       a.b.
```

Написати програм који за свако могуће поравнање ниске s и t одређује број тачака који им се поклапају.

Опис улаза

Са стандардног улаза се учитава ниска s (дужине највише 10^5 карактера) и ниска t дужине највише 10^4 карактера).

Опис излаза

На стандардни излаз исписати број поклопљених тачака за свако поравнавање ниске s и t.

Пример

Улаз	*Излаз*
`ab.c..ab..a.a.baa.a.c..a` `b.ac..b.`	`2 3 1 2 4 1 2 2 2 1 2 0 3 1 2 2 2`

Објашњење

У првом поравнавању поклапају се две тачке.

```
ab.c..ab..a.a.baa.a.c..a
b.ac..b.
```

У другом поравнавању поклапају се три тачке.

```
ab.c..ab..a.a.baa.a.c..a
 b.ac..b.
```

Слично се броје тачке и код осталих поравнавања.

Решење

Груба сила

Решење грубом силом подразумева да се за свако могуће поравнавање експлицитно преброје тачке које се поклапају. Сложеност овог решења је $O(|s| \cdot |t|)$.

Множење полинома

Креирајмо бинарне низове S и T истих дужина као ниске s и t, такве да је $S_k = 1$ акко је `s[k] == '.'` и да је $T_k = 1$ акко је `t[k] == '.'`. Размотримо поравнање које почиње на позицији i. Тада се број тачака које се поклапају може израчунати изразом:

$$\sum_{j=0}^{|t|-1} S_{i+j} \cdot T_j.$$

Овај збир је потребно израчунати за сваку позицију $0 \leq i \leq |s| - |t|$. Операција у којој се краћи низ помера дуж дужег и рачунају се збирови производа елемената на одговарајућим позицијама називају се *конволуција* и она се једноставно своди на проблем множења полинома.

Размотримо множење полинома $P(x) = \sum_{i=0}^{m} a_i x^i$ и $Q(x) = \sum_{j=0}^{n} b_j x^j$. Њихов производ је полином:

$$P(x)Q(x) = \sum_{i=0}^{m} \sum_{j=0}^{n} a_i b_j x^{i+j}.$$

Групишући коефицијенте уз исти степен броја x добијамо:

$$P(x)Q(x) = \sum_{k=0}^{m+n} \left(\sum_{j=0}^{k} a_{k-j} b_j \right) x^k,$$

при чему подразумевамо да је $a_i = 0$ за $i \geq m$ и $b_j = 0$ за $j \geq n$. Фокусирајмо се на коефицијенте уз степене x^k за k између $m-1$ и $n-1$. Они су једнаки:

$$\sum_{j=0}^{m-1} a_{k-j} b_j,$$

што личи на суме које желимо да израчунамо конволуцијом, једино што се елементи множе у обратном редоследу (први елемент низа b множи са последњим елементом

одговарајућег дела низа a, други са претпоследњим итд.). Једна операција се лако своди на другу обртањем редоследа коефицијената једног од полинома[22]. Узмимо да је $P(x) = \sum_{i=0}^{|s|-1} S_i x^i$, а да је $Q(x) = \sum_{j=0}^{|t|-1} T_{|t|-1-j} x^j$. Тада су коефицијенти уз x^k за $|t| - 1 \leq k \leq |s| - 1$ једнаки:

$$\sum_{j=0}^{|t|-1} S_{k-j} T_{|t|-1-j} = \sum_{j=0}^{|t|-1} S_{k-|t|+1+j} T_j$$

Приметимо да су ово управо збирови који су нама потребни: збир за дати померај i је коефицијент уз x^k, за $k = |t| - 1 + i$. Дакле, потребно је само формирати полиноме P и Q на описани начин, помножити их (што можемо ефикасно урадити коришћењем брзе Фуријеове трансформације) и прочитати коефицијенте уз одговарајуће степене. Сложеност овог приступа је $O(|s| \log |s|)$.

```
string s, t;
cin >> s >> t;

// najmanji stepen dvojke veći od dužine prve niske
int n = pow2(s.size());

// koeficijenti prvog polinoma
ComplexVector S(n, 0);
for (int i = 0; i < s.size(); i++)
  if (s[i] == '.')
    S[i] = 1;

// koeficijenti drugog polinoma (smešteni u obratnom redosledu)
ComplexVector T(n, 0);
for (int i = 0; i < t.size(); i++)
  if (t[t.size() - i - 1] == '.')
    T[i] = 1;

// proizvod polinoma (korišćenjem fft i ifft)
ComplexVector st = mult(ss, tt);

// traženi brojevi su koeficijenti proizvoda
```

[22] У литератури не постоји консензус који се тачно облик ових операција назива конволуција.

```
for (int i = 0; i <= s.length() - t.length(); i++)
  cout << round(st[i + t.length() - 1].real()) << " ";
```

Задатак: Дигитални бројач

$2n$-то цифрени дигитални бројач одбројава од $000\ldots000$ до $999\ldots999$ и емитује звук сваки пут када је збир првих n цифара једнак збиру последњих n цифара. На пример, шестоцифрени бројач пушта звук за $000000, 001001, 001010, \ldots, 999999$. Написати програм који одређује колико пута ће се чути звук.

Опис улаза

У првој линији стандардног улаза налази се природан број n ($1 \leq n \leq 9$).

Опис излаза

На стандарном излазу приказати колико постоји $2n$-цифрених бројева са траженим својством.

Пример

Улаз	*Излаз*
3	55252

Решење

Функција генератриса

Прва и друга половина броја се могу бирати независно. Ако постоји b_k начина да се одабере половина броја тако да је збир цифара у тој половини једнак k, онда постоји b_k^2 начина да се одабере цео број тако да је збир обе половине једнак k. Коначно решење се онда добија сабирањем ових вредности за све могуће збирове k (од 0 до $9n$).

Размотримо полином $P(x) = 1 + x + x^2 + \ldots + x^9$. Можемо сматрати да сваки његов коефицијент (а сваки је једнак 1) даје број начина за које једноцифрени број има збир цифара редом 0, 1, па до 9. Размотримо полином

$$\begin{aligned} P(x)^2 &= (1 + x + x^2 + \ldots + x^9)(1 + x + x^2 + \ldots + x^9) \\ &= 1 + 2x + 3x^2 + x^3 + \ldots + 9x^8 + 10x^9 + 9x^{10} + \ldots 2x^{17} + 1x^{18}. \end{aligned}$$

Коефицијент уз x^k сада даје број начина да се број k добије као збир цифара неког двоцифреног броја. На сличан начин коефицијент уз x^k, полинома $P(x)^m$ даје број начина да се број k добије као збир цифара неког m-тоцифреног броја.

Заиста, полином $P(x)^m$ добија се сабирањем производа свих могућих m-торки монома из $P(x)$. Постоји бијекција између m-тоцифрених бројева и тих m-торки монома из $P(x)$, таква да се свакој цифри c додељује моном x^c. Производ монома такве m-торке је моном x^k, где је k збир цифара m-тоцифреног броја који одговара тој m-торки. Зато је број m-тоцифрених бројева чији је збир цифара k (број b_{mk}) једнак броју m-торки монома чији је производ x^k. Збир свих тих монома је $b_{mk}x^k$, тако да се број b_{mk} заиста може одредити као коефицијент уз x^k полинома $P(x)^m$.

За паран број n можемо одредити коефицијенте полинома $P(x)^{\frac{n}{2}}$ и резултат можемо добити као збир квадрата свих добијених коефицијената. Степен се може одредити комбинацијом брзог степеновања (свако множење полинома се може вршити помоћу брзе Фуријеове трансформације). Такође, могуће је само пронаћи брзу Фуријеову трансформацију полинома P, затим вредност у свакој тачки засебно степеновати (брзим степеновањем), па резултат реконструисати инверзном Фуријеовом трансформацијом.

Пребројавање комбинаторних објеката смо свели на операције над полиномима (други уобичајени начин пребројавања је заснован на динамичком прогамирању). Полиноми или редови чији коефицијенти дају број комбинаторних објеката називају се *функције генератрисе* и представљају веома значајну технику у комбинаторици.

Свако множење полинома применом FFT је сложености $O(n \log n)$. Ако се изводи брзо степеновање на степен $n/2$, врши се $O(\log n)$ множења полинома, па је укупна сложеност $O(n \log^2 n)$. Ако би се извршила само брза Фуријеова трансформација полинома $P(x)$, затим степеновале његове вредности у тачкама, па резултат добио инверзном брзом Фуријеовом трансформацијом, сложеност би била $O(n \log n)$.

```
// brzim stepenovanjem odredjuje se stepen polinoma P^k
ComplexVector stepen(const ComplexVector& a, int k) {
  if (k == 0)
    // Polinom P(x) = 1 predstavljen vektorom date duzine
    return one(a.size());
  if (k % 2 == 0)
    return stepen(proizvod(a, a), k/2);
  else
    return proizvod(a, stepen(a, k-1));
}

// za svaki broj iz [0, maxZbir] odredjujemo broj nacina da se taj
// broj predstavi kao zbir brojCifara cifara (pri cemu se u obzir
// uzima i redosled cifara)
```

```
vector<long long> brojParticija(int brojCifara, int maksZbir) {
  // najmanji stepen dvojke veci ili jednak od maksZbir+1
  int n = pow2(maksZbir + 1);
  // polinom P(x) = 1 + x + ... + x^9 sa vektorom koeficijenata duzine n
  ComplexVector P(n, 0);
  for (int i = 0; i < 10; i++)
    P[i] = 1;
  // stepen P(x)^brojCifara
  ComplexVector Pbc = power(P, brojCifara);
  // konvertujemo kompleksne brojeve (koeficijente polinoma) u cele
  vector<long long> rezultat(n);
  for (int i = 0; i < n; i++)
    rezultat[i] = round(Pn2[i].real());
  return rezultat;
}

// izracunava broj 2n-tocifrenih brojeva kod kojih je zbir prvih n i
// zbir drugih n cifara jednak
long long brojBrojeva(int n) {
  // za svaki broj od 0 do 9*n nas zanima koliko postoji razlicitih
  // n-tocifrenih brojeva ciji je zbir cifara taj broj
  vector<long long> bp = brojParticija(n, 9*n);
  // ukupan broj brojeva cija leva i desna polovina imaju isti broj cifara
  long long ukupnoBrojeva = 0;
  // za svaki moguci zbir cifara polovine broja
  for (int zbirCifara = 0; zbirCifara <= 9*n; zbirCifara++) {
    // postoji bp[zbirCifara] nacina da napravimo levu polovinu i
    // bp[zbirCifara] nacina da napravimo desnu polovinu broja
    // tj. bp[zbirCifara]*bp[zbirCifara] nacina da odaberemo broj kome
    // i leva i desna polovina imaju zbir cifara zbirCifara
    ukupnoBrojeva += bp[zbirCifara] * bp[zbirCifara];
  }
  return ukupnoBrojeva;
}
```

4. Алгоритми за анализу и обраду текста

Рачунари се користе за обраду великих количина текстуалних података и да би то било могуће урадити на ефикасан начин развијен је великих број алгоритама који се баве анализом и обрадом текста. Најчешћи проблеми који се решавају су претрага текста (провера да ли једна ниска садржи другу), проналажење најдужих заједничких подниски две ниске, проналажење најдуже подниске која се понавља унутар ниске, проналажење најкраће подниске који се јавља само једном унутар ниске, проналажење најдужег палиндрома унутар ниске и слично. Сви ови проблеми имају велике практичне примене. На пример, испитивање да ли се ниска јавља у тексту се користи у претрази докумената (веб-страна, PDF докумената, датотека итд.), провери правописа, детекцији плагијата, филтрирању нежељене поште и слично.

За све ове проблеме није тешко развити алгоритме сложености $O(n^2)$, међутим, показује се да је могуће доћи и до ефикаснијих алгоритама, сложености $O(n)$. Ако је текст насумично генерисан или је у питању текст неког природног језика, скуп карактера је богат, па до најгорег случаја ретко долази и наивни алгоритми могу бити довољно ефикасни (на пример, наивни алгоритам претраге ниске у тексту који испитује сваку позицију може бити довољно ефикасан, јер ће се на свакој позицији детектовати непоклапање већ након анализе првог или првих неколико карактера). Напреднији алгоритми се стога чешће користе када се ниска тражи у дугачком тексту чија је азбука мале кардиналности, на пример када се претражују молекули ДНК који се описују низом слова из четворословне азбуке A, C, G, T.

У наставку ће бити изложени неки класични алгоритми и структуре података које се користе за анализу и обраду текста.

- У првом поглављу биће описани алгоритми засновани **на хеширању ниски**, са посебним нагласком на **Рабин-Карпов** алгоритам за тражење ниске у тексту.
- Након тога биће описани **z-низ** и **z-алгоритам** за његову конструкцију, као и примена тог алгоритма на проблем тражења ниске у тексту.
- Затим је објашњен **Кнут-Морис-Пратов** алгоритам (**КМР**) којим се у првој фази одређују дужине најдужих префикс-суфикса сваког префикса ниске, а затим се тако припремљен низ примењује на проблем претраге ниске у тексту. Овај низ дужина се може применити на решавање многих сродних проблема, а у књизи је приказан проблем испитивања периодичности дате ниске.
- На крају је приказан и **Маначеров алгоритам** за проналажење најдуже палиндромске подниске.

Приказан скуп алгоритама, наравно, није исцрпан. Многи проблеми са текстом се могу решавати техником динамичког програмирања (на пример, едит-растојање, проналажење најдуже заједничке подниске или проналажење најдужег заједничког подниза две ниске). Велики број операција над текстом се ефикасно спроводи коришћењем суфиксних дрвета и суфиксних низова, па се читаоцу заинтересованом за тему обраде текста саветује њихово самостално проучавање.

4.1 Хеширање ниски

Хеширање ниски (енгл. string hashing) подразумева да се свакој ниски додели цео број из одређеног опсега. Иако не може да се гарантује да ће свакој ниски бити додељен различити број, вероватноћа да су две различите ниске представљене истим бројем треба да буде јако мала. На тај начин се проблем поређења дугачких ниски може свести на поређење бројева који их репрезентују, што је много ефикаснија операција. Ако бројеви којим су представљене ниске нису једнаки, онда ниске сигурно нису једнаке. Ако су бројеви једнаки, полазне ниске су готово извесно једнаке (ако желимо да будемо баш апсолутно сигурни, можемо их експлицитно упоредити). На пример, ако желимо да упоредимо да ли су две велике датотеке (на пример, књиге у PDF формату или видеозаписи исти), можемо упоредити само њихове хеш-вредности.

Техника хеширања ниски је добила на популарности када су Рабин[1] и Карп[2] осмислили ефикасан алгоритам за тражење једне ниске у другој (проверу да ли се једна ниска јавља као подниска тј. сегмент узастопних карактера друге) заснован на хеширању. Алгоритми хеширања могу бити корисни и у решавању неких других проблема над текстом, као што су одређивање броја различитих подниски дате ниске, проналажење најдуже

[1] Михаел Рабин (енгл. Michael Rabin), рођен 1931., израелски математичар и информатичар.
[2] Ричард Карп (енгл. Richard Karp), рођен 1935., амерички информатичар.

ниске која се јавља бар два пута као подниска дате ниске, компресија ниске, одређивање броја палиндромских подниски у датој ниски и др.

Хеширање има примену у разним доменима.

У криптографији се дефинишу *криптографске хеш-функције* (на пример, SHA, MD5) које поред уобичајених својстава хеш-функција имају и додатна сигурносна својства (на пример, иако се зна алгоритам хеширања, сложеност проналажења ниске која би била представљена датом хеш-вредношћу је огромна и овај задатак је практично нерешив). Једна од уобичајених примена криптографских хеш-функција је у системима за проверу лозинки. Наиме, из безбедоносних разлога, лозинке се обично чувају у хешираном облику (јер у случају да неко успе да добије неовлашћени приступ систему, он неће моћи да сазна стварне лозинке, већ само њихове хеш-вредности на основу којих је готово немогуће реконструисати оригиналне лозинке). Када корисник унесе лозинку, рачуна се њена хеш-вредност и упоређује са оном сачуваном. Ако се те две хеш-вредности не поклопе, тада је сигурно унета погрешна лозинка. Ако се поклопе, постоји извесна вероватноћа да је унета лозинка погрешна (јер хеш-функције нису инјективне), међутим, хеш-функције се праве тако да та вероватноћа буде јако мала (сматра се да је већа вероватноћа да хардвер погрешно израчуна неку хеш-вредност него да нека насумично одабрана лозинка има исту хеш-вредност као права лозинка).

Хеширање се користи и за обезбеђивање интегритета поруке која се шаље путем канала за комуникацију. Провера да ли је послата порука неизмењена се врши поређењем хеш вредности поруке пре и после слања: ако су ове две вредности једнаке, сматрамо да је пренос успешно завршен. Често се приликом преузимања докумената са интернета (на пример, инсталационих датотека) поред докумената на сајту наводе и њихове хеш-вредности да би корисник био сигуран да је преузео документе који нису у међувремену измењени.

Хеш-вредности се користе и приликом дигиталног потписа. Уместо да се читав документ потписује тајним кључем (што може бити веома неефикасно), израчунава се његова хеш-вредност која се потписује. Хеш-вредност гарантује и да није дошло до измена документа након његовог потписивања.

Хеширање има примену и у преводиоцима програмских језика. Наиме, већина програмских језика има велики број резервисаних речи, попут кључних речи `if`, `for` и `while`, које је потребно обрадити на другачији начин од осталих идентификатора. Како би утврдио да ли је прочитана реч из изворног кода резервисана, преводилац чува хеширане вредности резервисаних речи тог програмског језика.

4.1.1 Хеш-функције и њихова својства

Приликом хеширања, ниски s се коришћењем неке *хеш-функције* (енгл. hash function) h придружује *хеш-вредност* (енгл. hash value) $h(s)$ која обично припада неком интервалу природних бројева $[0, m)$. Хеш-функција задовољава наредни услов: ако су две ниске s и t једнаке, и њихове хеш-вредности $h(s)$ и $h(t)$ су једнаке. Наиме, хеш-функција је увек *детерминистичка* – за један улаз увек даје један исти излаз. Са друге стране, хеш-функција не мора бити инјективна: ако је $h(s) = h(t)$, не мора да важи $s = t$. Ситуација када за две различите ниске s и t важи $h(s) = h(t)$ назива се *колизија* (енгл. collision).

Пошто је број различитих ниски које је могуће хеширати обично јако велики, хеш-функције готово никада нису инјективне и колизије је немогуће избећи. На пример, ако бисмо хтели да вредности хеш-функције буду јединствене за сваку ниску дужине до 14 карактера која се састоји само од малих слова енглеске абецеде, потребно би нам било $1 + 26^1 + ... + 26^{14} = \frac{26^{15}-1}{26-1}$ различитих хеш-вредности, што је око 26^{14}. Пошто је $26^{14} > 2^{64}$, те хеш-вредности не би стале у опсег 64-битних целих бројева, што је најшири примитивни целобројни тип података. У пракси обично разматрамо и много дуже ниске и шири скуп карактера и јасно је да чак ни са повећањем броја битова (данас се често користе хеш-вредности представљене помоћу 256 битова) инјективност није могуће постићи.

Пожељно је да хеш-функција буде сурјективна на свом кодомену $[0, m)$ као и да вероватноћа да хеш-вредности две различите ниске буду једнаке, односно да дође до колизије, буде што је могуће мања. У великом броју случајева та је вероватноћа толико мала, да се могућност доласка до колизије игнорише, чиме се теоријски нарушава коректност алгоритма. Могуће је, пак, да се у случајевима када се добију једнаке хеш-вредности две ниске, провери једнакост ове две ниске карактер по карактер. Тиме се гарантује тачност, али се потенцијално жртвује ефикасност алгоритма. Да ли ће се та провера вршити зависи од тога да ли је примена таква да је у реду толерисати јако малу вероватноћу нетачног одговора.

Приликом хеширања две фиксиране ниске обично не долази до колизије (тј. не дешава се да две различите ниске имају исту хеш-вредност). На пример, за одабир вредности параметра $m = 10^9$ (што одговара хеш-вредностима од тридесетак битова), под претпоставком да су све хеш-вредности једнако вероватне, вероватноћа да дође до колизије је само $1/m = 10^{-9}$. Међутим, уколико једну фиксирану ниску s поредимо са 10^6 других различитих ниски, вероватноћа да дође до бар једне колизије износи око $10^6 \cdot 10^{-9} = 10^{-3}$, док ако поредимо 10^6 ниски међусобно вероватноћа да дође до бар једне колизије је скоро једнака 1.[3]

[3] Последњи од поменутих сценарија познат је под називом *рођендански парадокс* (енгл. birthday paradox)

Постоји једноставан „трик” којим се може смањити вероватноћа да до колизије дође – рачунањем вредности две различите хеш-функције (штавише, може се искористити и иста хеш-функција за различите вредности својих параметара). Ако је вредност параметра m око 10^9 за обе хеш-функције, онда је ово упоредиво са коришћењем једне хеш-функције за вредност параметра m која је приближно једнака 10^{18}. Сада, ако поредимо 10^6 ниски међусобно, вероватноћа да дође до колизије смањиће се на $(10^6 \cdot 10^6) \cdot 10^{-18} = 10^{-6}$.

Хеширање се често користи да би се алгоритми грубе силе учинили ефикаснијим. Размотримо проблем упоређивања две ниске s и t. Алгоритам грубе силе пореди дате ниске s и t карактер по карактер и сложености је $O(\min\{|s|, |t|\})$, где су $|s|$ и $|t|$ дужине ниски s и t редом. Поставља се питање да ли постоји ефикаснији алгоритам за поређење ниски s и t. Сваку од ниски s и t можемо хеширати и пресликати у целобројну вредност и уместо ниски упоредити добијене целобројне вредности. Упоређивање ниски свођењем на упоређивање њихових хеш-вредности је сложености $O(1)$. Међутим, не треба занемарити чињеницу да је само хеширање ниски s и t, као што ћемо ускоро видети, сложености $O(|s| + |t|)$.

4.1.2 Дефинисање хеш-функције

У језику C++ на располагању нам је структура `hash`, која се може употребити за рачунање хеш-вредности ниски (али и хеш-вредности осталих основних типова података).

```
hash<string> h;
cout << h("abrakadabra") << endl;
```

Ипак, у већини наредних алгоритама користићемо хеш-функције које ћемо сами дефинисати.

Јасно је да хеш-функција треба да зависи од мултискупа карактера који се јављају у ниски и од редоследа карактера у ниски. Ниске `maja` и `mara` би требало да имају различите хеш-вредности, као и ниске `maja` и `jama`. Добар и широко распрострањен начин дефинисања хеш-функције ниске s дужине n је коришћењем *полиномске хеш-функције* (енгл. polynomial rolling hash function). Она долази у варијанти слева-надесно и здесна-налево.

4.1.2.1 Полиномска хеш-функција слева-надесно

Полиномска хеш-функција слева-надесно за ниску s дужине n се дефинише на следећи начин:

и односи се на следећи контекст: ако се у једној соби налази n особа, вероватноћа да неке две особе имају рођендан истог дана за $n = 23$ износи око 50%, док за $n = 70$ она износи чак 99.9%.

$$\begin{aligned} h(s) &= (s[0] \cdot p^{n-1} + s[1] \cdot p^{n-2} + ... + s[n-1]p^0) \bmod m \\ &= (\sum_{i=0}^{n-1} s[i] \cdot p^{n-i-1}) \bmod m, \end{aligned} \tag{4.1}$$

при чему је $s[i]$ целобројни кôд карактера на позицији i ниске s (при чему се позиције броје од нуле), а p и m су неки унапред одабрани, позитивни бројеви. Приметимо да претходна дефиниција полиномске хеш-функције одговара представљању „броја” s у бројевном систему са основом p (уз одређивање остатка при дељењу добијене вредности са са m на крају).

Потребно је сваки карактер ниске s кодирати целим бројем. На пример, када се кодирају само мала слова енглеске абецеде, могуће је користити следеће кодирање: $a \to 1, b \to 2, \ldots, z \to 26$. Оно се може имплементирати на следећи начин:

```
// funkcija koja izracunava kod datog karaktera: a->1, b->2, ..., z->26
int kod(char c) {
  return c - 'a' + 1;
}
```

Придруживање кода 0 неком карактеру (на пример, $a \to 0$), није погодно јер би онда вредности хеш-функције свих ниски које се састоје од тог карактера (на пример, `a`, `aa`, `aaa`, `aaaa`, ...) биле једнаке 0, док би ниске од којих се једна добија додавањем неколико тих карактера на почетак друге (на пример, ниске `na` и `ana`) имале исту вредност хеш-функције.

Пажљив одабир параметара p и m је важан да би се осигурала добра својства хеш-функције. Често се за p бира први прост број већи од броја карактера у улазној азбуци. На пример, ако се ниске састоје само од малих слова енглеске абецеде, онда је добар избор $p = 31$. Наиме, показује се да када је вредност параметра p мања од величине азбуке, онда је `p` кôд неког могућег карактера ниске и лако је пронаћи две ниске већ дужине 2 код којих се јавља колизија (на пример, то могу бити ниске чији су кодови карактера `[2, 0]` и `[1, p]`). Пожељно је, такође, да вредност параметра m буде неки велики број, јер је вероватноћа да две случајно одабране ниске имају исту хеш-вредност приближно једнака $1/m$. Ипак, избегаваћемо превелике вредности за параметар m, јер је погодно да вредности $m \cdot m$ и $p \cdot m$ не изазивају прекорачење (што ћемо образложити нешто касније).

Као и за параметар p, и за параметар m је погодно бирати неки прост број. Прости бројеви се користе да се не би дешавао велики број колизија када скуп ниски које се хеширају испољава одређена математичка својства. На пример, ако би m био паран број и ако би кодови свих карактера који се јављају у том скупу били парни бројеви,

тада би и све хеш-вредности биле парне (јер је остатак при дељењу парног броја парним бројем увек паран), што би повећало вероватноћу колизија, јер би практично пола скупа вредности било неискоришћено. У наредним примерима за вредност параметра m бираћемо прост број $10^9 + 9$. Он је мањи од 2^{30}, па вредност $m \cdot m$ стаје у 64-битни цео број. Приметимо и следеће: када израчунавања не бисмо вршили по модулу m, односно ако у дефиницији полиномске хеш-функције не би фигурисала вредност m, онда би се, услед прекорачења, операције вршиле по модулу броја подржаних вредности одговарајућег целобројног типа (дакле по модулу 2^{32} или 2^{64}).

Број различитих ниски расте јако брзо, па се и за скупове веома кратких ниски може гарантовати постојање колизије. На пример, ако за $m = 10^9 + 9$ размотримо скуп ниски састављених од малих слова енглеске абецеде, таквих да је дужина ниске мања или једнака 7, укупан број оваквих ниски износи: $26^0 + 26^1 + 26^2 + 26^3 + 26^4 + 26^5 + 26^6 + 26^7 = 8353082583 > 10^9 + 9 = m$. У овој ситуацији гарантовано је постојање колизије.

Израчунавање хеш-вредности се на основу дефинисане полиномске хеш-функције спроводи применом Хорнерове шеме и линеарне је сложености $O(|s|)$ у односу на дужину ниске $|s|$. Имплементација у језику C++ може бити следећа:

```
long long izracunajHesVrednost(const string &s) {
  int p = 31;
  int m = 1e9 + 9;

  // vrednost hes-funkcije niske s racunamo u duhu Hornerove sheme
  long long h = 0;
  for (int i = 0; i < s.size(); i++) {
    h = (h * p + kod(s[i])) % m;
  }
  return h;
}
```

Да би се рачун свео на мање бројеве, у функцији за рачунање хеш-вредности ниске s дужине n међурезултати су рачунати по модулу m, тј. разматран је низ међувредности h_i за $i = 0, 1, \ldots, n$, који одговара хеш-вредностима префикса ниске s дужине i.

$$\begin{aligned} h_0 &= 0 \\ h_{i+1} &= (h_i \cdot p + s[i]) \bmod m, \quad 0 \leq i < n \end{aligned}$$

Коначна хеш-вредност је вредност h_n.

Рачунање међурезултата по модулу m математички је коректно на основу правила модуларне аритметике.

Приметимо да пошто је $h_i < m$, $s[i] < p$ и пошто $p \cdot m$ не изазива прекорачење, приликом израчунавања вредности h_{i+1} не долази до прекорачења.

Ова полиномска хеш-функција има својство да јој се вредност може лако ажурирати када се симболи додају или уклањају са крајева ниске (одатле потиче термин `rolling` у енглеском називу). Наиме, ако познајемо хеш-вредност ниске $s_i s_{i+1} \dots s_{i+n-1}$, тада лако можемо израчунати хеш-вредност ниске $s_{i+1} s_{i+2} \dots s_{i+n}$. Прва вредност је једнака $h_i = (s[i] \cdot p^{n-1} + s[i+1] \cdot p^{n-2} + \dots + s[i+n-1]) \bmod m$, а друга $h_{i+1} = (s[i+1] \cdot p^{n-1} + s[i+2] \cdot p^{n-2} + \dots + s[i+n]) \bmod m$, па је

$$h_{i+1} = \left((h_i - s[i]p^{n-1}) \cdot p + s[i+n]\right) \bmod m. \tag{4.2}$$

4.1.2.2 Полиномска хеш-функција здесна-налево

Полиномска хеш-функција здесна-налево ниске s се дефинише на следећи начин:

$$\begin{aligned} h'(s) &= (s[0] + s[1] \cdot p + \dots + s[n-1] \cdot p^{n-1}) \bmod m \\ &= (\textstyle\sum_{i=0}^{n-1} s[i] \cdot p^i) \bmod m \end{aligned} \tag{4.3}$$

код које је, насупрот претходно дефинисаној хеш-функцији, најмањи степен броја p уз први карактер ниске s, а највећи уз последњи карактер ниске.

За израчунавање ове варијанте хеш-функције, пожељно је разматрати низ h'_i за $i = n, n-1, \dots, 0$, који одговара хеш-вредностима суфикса ниске s дужине $n-i$, при чему је циљ израчунати вредност h'_0:

$$\begin{aligned} h'_n &= 0 \\ h'_i &= (h'_{i+1} \cdot p + s[i]) \bmod m, \quad 0 \le i < n \end{aligned}$$

У језику C++ имплементација може бити следећа:

```
long long izracunajHesVrednost(const string &s) {
  int p = 31;
  int m = 1e9 + 9;

  // vrednost hes-funkcije niske s racunamo u duhu Hornerove sheme
  // razmatranjem karaktera niske u obratnom poretku
```

```
  long long h = 0;
  for (int i = s.size() - 1; i >= 0; i--) {
    h = (h * p + kod(s[i])) % m;
  }
  return h;
}
```

И ова полиномска хеш-функција има својство да јој се вредност може лако ажурирати када се симболи додају или уклањају са крајева ниске. Наиме, ако познајемо хеш-вредност ниске $s_i s_{i+1} \dots s_{i+n-1}$, тада лако можемо израчунати хеш-вредност ниске $s_{i+1} s_{i+2} \dots s_{i+n}$. Прва вредност је једнака $h_i = (s[i] + s[i+1] \cdot p + \dots + s[i+n-1] \cdot p^{n-1}) \bmod m$, а друга $h_{i+1} = (s[i+1] + s[i+2] \cdot p + \dots + s[i+n] \cdot p^{n-1}) \bmod m$, па је

$$h_{i+1} = \left(\frac{h_i - s[i]}{p} + s[i+n] \cdot p^{n-1} \right) \bmod m.$$

Уместо дељења са p, могуће је вршити множење модуларним инверзом броја p по модулу m.

4.1.3 Тражење ниске у тексту (Рабин-Карпов алгоритам)

Рабин и Карп су 1987. године предложили алгоритам који решава проблем тражења ниске (енгл. pattern) у тексту (енгл. text) коришћењем технике хеширања ниски.

Проблем

За дати текст $t = t_0 t_1 \dots t_{N-1}$ и ниску $p = p_0 p_1 \dots p_{M-1}$ одредити све подниске узастопних елемената (сегменте) ниске t које су једнаке p.

Идеја Рабин-Карповог алгоритма је следећа: коришћењем полиномске хеш-функције израчунати хеш-вредност ниске p и свих сегмената текста t, који имају дужину $|p| = M$ и упоредити њихове вредности. Ако се хеш-вредност ниске и сегмента не поклопи, текући сегмент се сигурно не поклапа са подниском коју тражимо и прелази се на наредни сегмент. Ако се хеш-вредности поклопе, тада треба извршити експлицитну проверу да ли се текући сегмент поклапа са подниском коју тражимо. На почетку израчунавамо хеш-вредност почетне подниске текста дужине M. У сваком кораку ову вредност инкрементално ажурирамо (у случају када је хеш-функција задата формулом (4.1) ажурирање вршимо на основу формуле (4.2)).

Размотримо имплементацију Рабин-Карповог алгоритма.

```
// Rabin-Karpov algoritam za trazenje niske u tekstu
vector<int> traziRabinKarp(const string &niska, const string &tekst) {
  int M = niska.size(), N = tekst.size();

  // vektor pocetnog indeksa pojavljivanja niske u tekstu
  vector<int> pozicije;

  // ako je tekst kraci od niske, on je ne sadrzi
  if (M > N) return pozicije;

  int p = 31, m = 1e9 + 9;
  // p^(M-1) mod m
  long long pS = stepen(p, M-1, m);

  // racunamo hes-vrednost date niske
  long long hesNiske = 0;
  for (int i = 0; i < M; i++)
    hesNiske = (hesNiske*p + kod(niska[i])) % m;

  // racunamo hes-vrednost pocetnih M karaktera teksta
  long long hesSegmenta = 0;
  for (int i = 0; i < M; i++)
    hesSegmenta = (hesSegmenta*p + kod(tekst[i])) % m;

  // proveravamo sve pocetne pozicije niske unutar teksta
  for (int i = 0; i <= N - M; i++) {
    // ako se poklapaju hesh vrednosti segmenta teksta i niske
    if (hesSegmenta == hesNiske)
      // eksplicitna provera poklapanja
      if (niska == tekst.substr(i, M))
        pozicije.push_back(i);
    // azuriramo hes-segmenta
    if (i < N - M)
      hesSegmenta = ((hesSegmenta - kod(tekst[i])*pS + m)*p
                     + kod(tekst[i + M])) % m;
  }
  return pozicije;
}
```

Израчунавање хеш-вредности ниске коју тражимо и почетног сегмента текста се може

урадити у времену $O(M)$. Након тога $O(N)$ пута ажурирамо хеш-вредност сегмента у времену $O(1)$ (претпостављајући да је степен $p^{M-1} \bmod m$ једном израчунат, што може бити урађено у времену $O(\log M)$ алгоритмом брзог степеновања). Ако се ниска не јавља у тексту, тада је сложеност претходног поступка $O(M+N)$. Међутим, сваки пут када се хеш-вредности поклопе, врши се експлицитна провера једнакости ниски, која захтева време $O(M)$. Пошто се у веома специјалним случајевима ниска може јављати унутар подниске $O(N)$ пута, те експлицитне провере захтевају време $O(MN)$, што је заправо сложеност најгорег случаја Рабин-Карповог алгоритма и он по томе није бољи од наивног алгоритма. Ипак, за очекивати је да ће се алгоритам примењивати у проблемима претраге обичних текстуалних докумената када се подниска у тексту јавља обично само $O(1)$ пута и тада је сложеност Рабин-Карповог алгоритма само $O(M+N)$, што је много ефикасније од наивног алгоритма.

4.1.4 Рачунање хеш-вредности сегмената ниске

У неким применама је потребно израчунавати хеш-вредности већег броја различитих сегмената дате ниске. То се може ефикасно урадити ако се израчунају хеш-вредности свих префикса ниске (идеја је слична израчунавању збирова сегмената на основу познатих збирова префикса низа).

Проблем

Дата је ниска s дужине n. Написати алгоритам којим се за дате парове индекса i и j (њих укупно q) за које важи $0 \leq i \leq j < n$, ефикасно израчунава хеш-вредности подниски $s[i..j]$ полазне ниске (подниске су сегменти суседних карактера полазне ниске s од позиције i закључно са позицијом j).

У наставку ћемо приказати како се овај проблем решава за обе варијанте полиномске хеш-функције. Видећемо да је коришћење полиномске функције слева-надесно мало једноставније, јер полиномска функција здесна-налево захтева израчунавање модуларног инверза.

4.1.4.1 Полиномска хеш-функција слева-надесно

Размотримо најпре прву варијанту полиномске хеш-функције која је задата формулом (4.1). Према дефиницији хеш-функције важи:

$$\begin{aligned} h(s[i..j]) &= (s[i] \cdot p^{j-i} + s[i+1] \cdot p^{j-i-1} + \ldots + s[j]) \bmod m \\ &= \left(\sum_{k=i}^{j} s[k] \cdot p^{j-k} \right) \bmod m \end{aligned}$$

Ова вредност може се директно израчунати алгоритмом временске сложености $O(j-i)$, односно у најгорем случају је сложености $O(|s|)$, где је $|s|$ дужина ниске s. Да би се наивним приступом израчунала вредност за q сегмената, потребно је време $O(q \cdot |s|)$. Да ли можемо ефикасније решити овај проблем?

Употребимо идеју префиксних сума, која омогућује да се збир сваког сегмента низа може изразити као разлика два збира префикса. Дефинишимо низ H_k тако да је $H_0 = 0$, а $H_k = h(s[0..k-1])$. Запишимо чему су једнаке хеш-вредности префикса ниске s дужина редом $j+1$ и i:

$$\begin{aligned} H_{j+1} = h(s[0..j]) &= (s[0] \cdot p^j + s[1] \cdot p^{j-1} + ... + s[j]) \bmod m, \\ H_i = h(s[0..i-1]) &= (s[0] \cdot p^{i-1} + s[1] \cdot p^{i-2} + ... + s[i-1]) \bmod m. \end{aligned}$$

При том сматрамо да је $h(s[0..-1]) = 0$ (што може да се деси за $i = 0$). Приметимо да је:

$$\begin{aligned} h(s[i..j]) &= (s[i] \cdot p^{j-i} + s[i+1] \cdot p^{j-(i+1)} + ... + s[j]) \bmod m \\ &= (h(s[0..j]) - p^{j-(i-1)} \cdot h(s[0..i-1])) \bmod m \\ &= (H_{j+1} - p^{j+1-i} \cdot H_i) \bmod m \end{aligned} \tag{4.4}$$

Пошто променљиве i и j узимају вредности од 0 до $|s|-1$, израз $j+1-i$ узима вредности од 1 до $|s|$, те је у ствари потребно израчунати вредност p^k за све вредности $0 \leq k \leq |s|$. Дакле, ако за дату ниску знамо хеш-вредности свих њених префикса и вредности p^k за $k = 0, 1, ..., |s|$, онда на основу једнакости (4.4) алгоритмом сложености $O(1)$ можемо израчунати хеш-вредност произвољног сегмента те ниске. Хеш-вредности свих префикса ниске s и вредности $p^k \bmod m$ за свако $k = 1, 2, ..., |s|$ могу се израчунати на почетку рада програма и то инкрементално, алгоритмом чија је временска сложеност $O(|s|)$, па је укупна сложеност рачунања хеш-вредности q различитих сегмената $O(|s| + q)$.

Пример 4.1.1

Желимо да рачунамо хеш-вредности сегмената ниске s *која је једнака* ananas. *Нека је* $p = 31$ *и* $m = 997$. *Важи да је* $|s| = 6$, *па рачунамо степене броја* p *по модулу* m *и добијамо низ степена* 1, 31, 961, 878, 299, 296 *и* 203. *Кодови карактера ниске* s *су редом* 1, 14, 1, 14, 1, 19. *Зато су хеш-кодови префикса* H_k *једнаки редом* 0, 1, 45, 399, 419, 29, 918.

За вредности $i = 1$ *и* $j = 4$, *потребно је израчунати хеш-вредност ниске* nana, *чији су кодови карактера редом* 14, 1, 14, 1. *На основу дефиниције хеш-функције, она је једнака*

$(14 \cdot 878 + 1 \cdot 961 + 14 \cdot 31 + 1 \cdot 1) \bmod 997 = 727$. *Са друге стране, ова вредност се може израчунати и као* $(H_{j+1} - p^{j+1-i} \cdot H_i) \bmod m = (H_5 - p^5 \cdot H_1) \bmod m = (29 - 299 \cdot 1) \bmod 997 = 727$.

Имплементација у језику C++ је приказана у наставку.

```
int p = 31;
long long m = 1e9 + 9;

// svi stepeni broja p
vector<long long> stepeniBrojaP(int n) {
  vector<long long> pStepen(n+1);
  pStepen[0] = 1;
  for (int i = 1; i <= n; i++)
    pStepen[i] = (pStepen[i-1] * p) % m;
  return pStepen;
}

// hesevi svih prefiksa date niske
vector<long long> heseviPrefiksa(const string& s) {
  int n = s.size();
  vector<long long> hesPrefiksa(n+1);
  hesPrefiksa[0] = 0;
  for (int i = 0; i < n; i++)
    hesPrefiksa[i+1] = (hesPrefiksa[i] * p + kod(s[i])) % m;
  return hesPrefiksa;
}

// hes vrednost segmenta s[i..j]
// racunamo koriscenjem poznatih hes-vrednosti prefiksa i stepena broja p
long long hesSegmenta(const string& s, int i, int j,
                      const vector<long long>& pStepen,
                      const vector<long long>& hesPrefiksa) {
  return
    (hesPrefiksa[j+1] - (hesPrefiksa[i] * pStepen[j+1-i]) % m + m) % m;
}
```

Приметимо да се приликом рачунања хеш-вредности сегмента извршава множење хеш-вредност префикса одговарајућим степеном броја p по модулу m. Да овај производ не би довео до прекорачења, потребан је услов да $m \cdot m$ не изазива прекорачење.

4.1.4.2 Полиномска хеш-функција здесна-налево

Размотримо сада коришћење друге предложене хеш-функције. Тада је хеш-вредност сегмента $s[i..j]$ једнака:

$$\begin{aligned} h'(s[i..j]) &= (s[i] + s[i+1] \cdot p + ... + s[j] \cdot p^{j-i}) \bmod m \\ &= \left(\sum_{k=i}^{j} s[k] \cdot p^{k-i} \right) \bmod m \end{aligned}$$

Ако обе стране ове једнакости помножимо са p^i добијамо:

$$\begin{aligned} h'(s[i..j]) \cdot p^i &= \left(\sum_{k=i}^{j} s[k] \cdot p^k \right) \bmod m \\ &= (h'(s[0..j]) - h'(s[0..i-1])) \bmod m \\ &= (H'_{j+1} - H'_i) \bmod m \end{aligned} \tag{4.5}$$

где важи да је $h'(s[0.. -1]) = 0$ (што може да се деси за $i = 0$) и где смо низ H' дефинисали тако да је $H'_0 = 0$, а $H'_{k+1} = h'(s[0..k])$.

Приметимо да је за рачунање вредности хеш-функције неког сегмента према формули (4.5) потребно извршити дељење израза $H'_{j+1} - H'_i$ вредношћу p^i по модулу m. За то је потребно одредити мултипликативни инверз броја p^i по модулу m, односно број x тако да важи $p^i \cdot x \equiv 1 (\bmod m)$, а затим извршити множење израза $H'_{j+1} - H'_i$ бројем x. Пошто сва израчунавања вршимо по модулу m, а с обзиром на то да смо на почетку претпоставили да су бројеви m и p прости, на основу мале Фермаове теореме за сваки број a који је узајамно прост са m важи $a^{m-1} \equiv 1 (\bmod m)$, односно мултипликативни инверз броја a по модулу m је $a^{m-2} \bmod m$. Ако у ову једнакост уместо a уврстимо вредност p^i (који јесте узајамно прост са m), добијамо мултипликативни инверз за произвољно p^i по модулу m. Још ефикасније, вредност $(p^k)^{-1} \bmod m$ можемо израчунати као $(p^{-1})^k \bmod m$.

Дакле, ако су унапред познате хеш-вредности свих префикса ниске s и вредности мултипликативног инверза броја $p^i, 1 \leq i \leq |s|$ по модулу m, израчунавање хеш-вредности произвољног сегмента полазне ниске може се на основу формуле (4.5) извршити алгоритмом сложености $O(1)$. Приметимо да је фаза предобраде која се састоји од рачунања хеш-вредности свих префикса ниске и вредности инверза броја p^i за свако i временске сложености $O(n \cdot \log m)$.

Пример 4.1.2

Желимо да рачунамо хеш-вредности сегмената ниске s која је једнака ananas. *Нека је* $p = 31$ *и* $m = 997$. *Важи да је* $|s| = 6$, *па рачунамо степене броја* p *по модулу* m *и добијамо низ степена* $1, 31, 961, 878, 299, 296$ *и* 203. *Кодови карактера ниске* s *су редом* $1, 14, 1, 14, 1, 19$. *Зато су хеш-кодови префикса* H_k *једнаки редом* $0, 1, 435, 399, 727, 29$ *и* 668. *Модуларни инверз броја* p *по модулу* m *можемо израчунати на основу мале Фермаове теореме као* $p^{m-2} \bmod m = 193$.

За вредности $i = 1$ *и* $j = 4$, *потребно је израчунати хеш-вредност ниске* nana, *чији су кодови карактера редом* $14, 1, 14, 1$. *На основу дефиниције хеш-функције, она је једнака* $(14 \cdot 1 + 1 \cdot 31 + 14 \cdot 961 + 1 \cdot 878) \bmod 997 = 419$.

Са друге стране, ова вредност се може израчунати и као $(p^{-i}) \cdot (H_{j+1} - H_i) \bmod m = p^{-1} \cdot (H_5 - H_1) \bmod m = 193 \cdot (29 - 1) \bmod 997 = 419$.

Имплементација у језику C++ је приказана у наставку.

```
int p = 31;
int m = 1e9 + 9;

// racunanje inverza prostog broja p po modulu m
// koriscenjem male Fermaove teoreme
long long modInverz(int p, int m) {
  return stepen_mod(p, m - 2, m);
}

// stepeni broja x po modulu m (1, ..., x^n mod m)
vector<long long> stepeniBrojaX(int x, int n, int m) {
  vector<long long> xStepen(n+1);
  xStepen[0] = 1;
  for (int i = 1; i <= n; i++)
    xStepen[i] = (xStepen[i-1] * x) % m;
  return pStepen;
}

// stepeni broja p po modulu m
vector<long long> stepeniBrojaP(int n) {
  return stepeniBrojaX(p, n, m);
}
```

```
// inverzi stepena broja p po modulu m
vector<long long> stepeniInverzaBrojaP(int n) {
  // inverzi stepena su stepeni inverza
  return stepeniBrojaX(modInverz(p, m), n, m);
}

// heseve svih prefiksa date niske s racunamo koriscenjem poznatih
// stepena broja p
vector<long long> heseviPrefiksa(const string& s,
                                 const vector<long long>& pStepen) {
  int n = s.size();
  vector<long long> hesPrefiksa(n+1, 0);
  for (int i = 0; i < n; i++)
    hesPrefiksa[i+1] = (hesPrefiksa[i] + kod(s[i]) * pStepen[i]) % m;
}

// hes segmenta s[i..j] racunamo koriscenjem poznatih
// heseva prefiksa niske s i inverza stepena broja p
long long hesSegmenta(const string& s, int i, int j,
                      const vector<long long>& hesPrefiksa,
                      const vector<long long>& invpStepen) {
  int n = s.size();
  // hes vrednost segmenta racunamo preko hash vrednosti prefiksa
  return
    (((hesPrefiksa[j+1] - hesPrefiksa[i] + m) % m) * invpStepen[i]) % m;
}
```

Задатак: Груписање једнаких ниски

Дато је n ниски $s_1, s_2, \ldots, s_n$. Пронаћи све дупликате међу њима и поделити овај низ ниски у групе једнаких ниски. За сваку групу одредити индексе ниски те групе у полазном низу.

Опис улаза

Са стандардног улаза се учитава $n \leq 10000$ ниски дужине највише $m \leq 10000$. Ниске су састављене само од малих слова енглеске абецеде.

Опис излаза

За сваку групу исписати индексе ниски из те групе у полазном низу.

Пример

Улаз	*Излаз*
ana	2
a	1 4 6
dvorana	5 8
ana	3
banana	7
ana	
kopakabana	
banana	

Решење

Директан приступ састоји се у поређењу сваке ниске са сваком другом: поређење две ниске максималне дужине m карактер по карактер је у најгорем случају је сложености $O(m)$ (додуше, може се очекивати да се најгори случај не дешава често тј. да ће се различитост две ниске установити већ након поређења њихових неколико почетних карактера). Укупан број поређења ниски је реда $O(n^2)$, те је укупна сложеност одговарајућег алгоритма $O(n^2 m)$.

Ефикасније решење добија се сортирањем свих ниски и тражењем дупликата у сортираном низу. Сортирање ниски укључује $O(n \log n)$ упоређивања ниски, а свако поређење ниски је у најгорем случају сложености $O(m)$, те је укупна сложеност сортирања ниски једнака $O(nm \log n)$. Додатно, пролазак кроз скуп ниски у сортираном редоследу и идентификовање истих ниски је сложености $O(nm)$, те је укупна сложеност овог алгоритма $O(nm \log n)$.

Приступ заснован на хеширању ниски редукује време поређења две ниске на $O(1)$ (под претпоставком да не проверавамо да ли је дошло до колизије). На тај начин добијамо алгоритам сложености $O(nm + n \log n)$, где сложеност $O(nm)$ потиче од рачунања хеш-вредности свих ниски, а $O(n \log n)$ од сортирања ниски на основу њихових хеш-вредности. Додатно, пролазак кроз хеш-вредности ниски у сортираном поретку и идентификовање истих ниски је сложености $O(n)$. Ако желимо да будемо сигурни у исправност алгоритма, када су хеш-вредности неких ниски једнаке, можемо експлицитно упоредити те ниске, за шта је у најгорем случају, када су све хеш-вредности једнаке, потребно n^2 поређења ниски, која захтевају време $O(m)$, што захтева додатно време $O(n^2 m)$. Међутим, реално је очекивати да су групе једнаких хеш-вредности много мање и да ће ово време бити много мање. Још боље од тога, можемо елементе сваке групе засебно лексикографски сортирати и затим поредити узастопне ниске.

Имплементација у језику C++ је дата у наставку. У њој се не проверава постојање колизија тј. претпоставља се да једнаким хеш-вредностима одговарају једнаке ниске. Имплементацију додатне фазе у којој би се извршило директно поређење ниски унутар

сваке групе и додатно раздвајање групе на подгрупе ако се утврди да садржи различите ниске препуштамо читаоцу.

```
vector<vector<int>> grupisiIsteNiske(const vector<string> &niske) {
  // broj niski
  int n = niske.size();
  // vektor parova hash vrednosti i pozicije niske u polaznom nizu
  vector<pair<long long, int>> h(n);
  // izracunavamo hash vrednost svake niske;
  // uz hes vrednost niske cuvamo i njen indeks u polaznom nizu
  for (int i = 0; i < n; i++)
    h[i] = {izracunajHesVrednost(niske[i]), i+1};

  // sortiramo niz hes vrednosti
  sort(h.begin(),h.end());

  // svaki element vektora sadrzi niz indeksa niski
  // koje su medjusobno jednake
  vector<vector<int>> grupe;

  // prolazimo skupom svih niski u sortiranom poretku
  for (int i = 0; i < n; i++){
    // ukoliko se radi o prvoj niski u sortiranom poretku
    // ili o niski koja nije jednaka prethodnoj u sortiranom poretku
    // onda je potrebna nova grupa
    if (i == 0 || h[i].first != h[i-1].first) {
      vector<int> novaGrupa;
      grupe.push_back(novaGrupa);
    }
    // u poslednju (tekucu) grupu dodajemo na kraj indeks niske
    // koja ima tekucu hash vrednost
    grupe.back().push_back(h[i].second);
  }
  return grupe;
}
```

Задатак: Број различитих сегмената

Дата је ниска s која се састоји само од малих слова енглеске абецеде. Израчунати број различитих непразних сегмената (подниски узастопних карактера) ове ниске.

Опис улаза

Са стандардног улаза се учитава ниска s, дужине највише 10^5 карактера.

Опис излаза

На стандардни излаз исписати број различитих подниски ниске s.

Пример

Улаз	*Излаз*	*Објашњење*
`ananas`	`15`	• Постоје три различите подниске дужине 1: `a`, `n` и `s`. • Постоје три различите подниске дужине 2: `an`, `na` и `as`. • Постоје три различите подниске дужине 3: `ana`, `nan` и `nas`. • Постоје три различите подниске дужине 4: `anan`, `nana` и `anas`. • Постоје три различите подниске дужине 5: `anana` и `nanas`. • Постоји једна подниска дужине 6: `ananas`. Ниска `ananas` има укупно $3 + 3 + 3 + 3 + 2 + 1 = 15$ различитих подниски.

Решење

Директан приступ би био да се све подниске убаце у скуп (било уређен или неуређен) и да се провери број елемената тог скупа. У наставку ћемо видети да се ова идеја може оптимизовати (пре свега по питању заузећа меморије).

За почетак, довољно је поредити само сегменте истих дужина јер само они могу бити међусобно једнаки, па уместо коришћења јединственог скупа у који смештамо све сегменте, сегменте можемо обрађивати у редоследу њихових дужина и у скупу можемо истовремено чувати само сегменте исте дужине.

Даље решење ћемо засновати на хеширању, при чему ћемо претпоставити да је хеш-функција таква да је вероватноћа колизије јако мала (тј. претпоставићемо да различите подниске исте дужине увек дају различите хеш-вредности). Користићемо полиномијалну хеш-функцију $h'(s) = (s[0] + s[1] \cdot p + ... + s[n-1] \cdot p^{n-1}) \bmod m$. Хеш-вредност сваког сегмента се тада може рачунати коришћењем разлике хеш вредности одговарајућих префикса $h'(s[i..j]) = (p^{-i} \cdot (H'_{j+1} - H'_i)) \bmod m$. Уместо да рачунамо и поредимо тачне хеш-вредности два сегмента исте дужине коришћењем модуларних мултипликативних инверза, довољно је израчунати хеш-вредности сегмената помножене неким (истим) степеном броја p. Претпоставимо да већ имамо израчунате хеш-вредности два сегмента x и y, једног помноженог са p^i, а другог са p^j. Без нарушавања

општости можемо претпоставити да је $i < j$; да бисмо добили обе хеш-вредности помножене истим степеном броја p, довољно је да хеш-вредност сегмента x помножимо са p^{j-i}.

Пример 4.1.3

Размотримо, на пример, сегменте $s[2..4]$ *и* $s[5..7]$ *дужине* 3 *ниске* s *која је дужине* 10. *Из једнакости* $p^i \cdot h'(s[i..j]) = (H'_{j+1} - H'_i) \bmod m$:

$$\begin{aligned} h_1 &= h(s[2..4]) \cdot p^2 = h(s[0..4]) - h(s[0..1]) \bmod m \\ h_2 &= h(s[5..7]) \cdot p^5 = h(s[0..7]) - h(s[0..4]) \bmod m \end{aligned}$$

Уместо да рачунамо мултипликативне инверзе бројева p^2 *и* p^5, *да бисмо добили тачне хеш-вредности сегмената* $s[2..4]$ *и* $s[5..7]$, *можемо поредити вредности* $h_1 \cdot p^{5-2}$ *и* h_2.

Дакле, пролазићемо редом кроз све могуће дужине $l = 1, 2, \ldots, n$ сегмената ниске s и конструисати серију хеш-вредности свих сегмената дужине l који су помножени неким (истим, максималним) степеном броја p. Број различитих елемената те серије (што је, под претпоставком да нема колизија, број различитих сегмената дужине l) можемо одредити коришћењем скупа (на пример, уређеног). Укупан број различитих сегмената једнак је збиру броја различитих сегмената дужине l у ниски, за свако могуће l. Имплементација овог приступа у језику C++ је дата у наставку.

Приметимо да се у претходној имплементацији приликом рачунања производа хеш-вредности сегмента и вредности p^{n-1} по модулу m множе две вредности из опсега $[0, m)$, те да не би било прекорачења приликом рачунања овог производа, вредност параметра m бирамо тако да $m \cdot m$ стаје у 64-битни цео број (узели смо највећи прост број који се може записати са 32 бита).

Операције за рад са уређеним скупом од k елемената су сложености $O(\log k)$, а сегмената има укупно $O(n^2)$, те сложеност овог алгоритма износи $O(n^2 \log n)$.

Уметањем хеш-вредности у скуп, уместо уметања самих подниски у скуп значајно је смањена меморијска сложеност скупа, која уз хеширање износи $O(n)$, а без хеширања би износила $O(n^2)$.

Истакнимо и то да се у овом контексту, када радимо са различитим сегментима ниске, исплати уложити додатно време за рачунање хеш-вредности свих префикса дате ниске, јер ћемо израчунате хеш-вредности већи број пута користити.

```
int brojRazlicitihPodniski(const string &s) {
  typedef unsigned long long ull;

  int p = 31;
  ull m = 4294967291ul; // najveci prost broj koji staje u unsigned long

  int n = s.size();
  // racunamo stepene broja p po modulu m
  vector<ull> pStepen(n);
  pStepen[0] = 1;
  for (int i = 1; i < n; i++)
    pStepen[i] = (pStepen[i-1] * p) % m;

  // racunamo hes-vrednosti svih prefiksa date niske
  vector<ull> h(n+1, 0);
  for (int i = 0; i < n; i++)
    h[i+1] = (h[i] + (s[i] - 'a' + 1) * pStepen[i]) % m;

  // broj razlicitih podniski
  int brojPodniski = 0;
  // za svaku mogucu duzinu segmenta
  for (int l = 1; l <= n; l++) {
    // skup hes vrednosti segmenata duzine l pomnozenih sa p^{n-1}
    set<ull> heseviDuzineL;
    // prolazimo kroz sve segmente duzine l
    for (int i = 0; i <= n - l; i++) {
      // hes-vrednost segmenta racunamo
      // kao razliku hes-vrednosti odgovarajucih prefiksa
      ull hTekuce = (h[i+l] - h[i] + m) % m;
      // racunamo hes vrednost segmenta pomnozenu sa p^{n-1} po modulu m
      // tako sto je mnozimo sa p^{n-i-1}
      hTekuce = (hTekuce * pStepen[n - i - 1]) % m;
      // hes-vrednost dodajemo u skup,
      // isti segmenti imace istu hesh vrednost pa se nece dva puta racunati
      heseviDuzineL.insert(hTekuce);
    }
    brojPodniski += heseviDuzineL.size();
  }
```

```
    return brojPodniski;
}
```

Задатак: Лексикографски најмања ротација

Написати програм који одређује лексикографски најмању ротацију дате ниске.

Опис улаза

Са стандардног улаза се уноси ниска дужине највише 10^6 карактера.

Опис излаза

На стандардни излаз исписати лексикографски најмању ниску која се добија ротирањем карактера унете ниске.

Пример

Улаз	*Излаз*	*Објашњење*
`atlas`	`asatl`	Ротације ниске `atlas` су `atlas`, `tlasa`, `lasat`, `asatl` и `satla`. Лексикографски најмања ниска од њих је `asatl`.

Решење

До решења можемо доћи тако што ћемо одређивати једну по једну ротацију ниске и поредити их лексикографски са до тада најмањом. Ако лексикографско поређење вршимо директним алгоритмом, чија је сложеност $O(n)$ у најгорем случају, добија се алгоритам сложености $O(n^2)$ (додуше, ако су ниске насумично генерисане или садрже текст на природном језику, лексикографски редослед се може одредити већ након анализе првих неколико карактера, па ће алгоритам радити ефикасније). Да би се избегло грађење нових ниски (што је скупа операција услед алокације меморије), ротације можемо добити тако што анализирамо све подниске дужине n ниске ss која се добија надовезивањем учитане ниска s два пута.

Лексикографско поређење се може убрзати коришћењем хеширања. Претпоставимо, једноставности ради, да неће бити колизија. Лексикографски редослед ниски се одређује тако што се пронађе први карактер који се разликује, тј. тако што се пронађе најдужи заједнички префикс две ниске које се пореде. Он се може пронаћи бинарном претрагом. У сваком кораку се испитује једнакост неких префикса две ниске које се пореде. Пошто су они сегменти ниске ss, поређење можемо извршити тако што израчунамо њихове хеш-вредности (као хеш-вредности сегмената ниске ss) и упоредимо их. Имплементација је олакшана, јер су ниске које се пореде исте дужине. Када се нађе дужина најдужег заједничког префикса, пореди се наредни карактер (осим у случају када су ниске једнаке, што се лако препознаје јер је тада дужина заједничког префикса

једнака дужини ниски које се пореде).

Пошто се бинарна претрага врши у времену $O(\log n)$, а ротација постоји n, сложеност најгорег случаја овог алгоритма је $O(n \log n)$. Напоменимо да се коришћењем напреднијих техника (нпр. коришћењем суфиксног дрвета) овај проблем може решити и ефикасније, алгоритмом линеарне временске сложености.

```
// znajuci heseve H svih prefiksa niske i stepene broja P
// izracunavamo najduzi zajednicki prefiks podniski na pozicijama
// [i1, i1+n) i [i2, i2+n)
int najduziZajednickiPrefiks(const vector<long long>& H,
                             const vector<long long>& P,
                             int i1, int i2, int n) {
  // binarna pretraga
  int l = 0, d = n-1;
  while (l <= d) {
    int m = l + (d - l) / 2;
    // poredimo da li se prefiksi niski dužine m poklapaju
    // pretpostavljamo da nema kolizija
    if (hesSegmenta(H, P, i1, i1+m) == hesSegmenta(H, P, i2, i2+m))
      l = m + 1;
    else
      d = m - 1;
  }
  return l;
}

bool leksikgrafskiManjaNiska(const vector<long long>& H,
                             const vector<long long>& P,
                             int i1, int i2, int n,
                             const string& ss) {
  int k = najduziZajednickiPrefiks(H, P, i1, i2, n);
  return k < n && ss[i1 + k] < ss[i2 + k];
}

string leksikografskiNajmanjaRotacija(const string& s) {
  int n = s.length();
  string ss = s + s;
  vector<long long> H = heseviPrefiksa(ss);
  vector<long long> P = stepeniBrojaP(n);
```

```
  int iMin = 0;
  for (int i = 1; i + n < ss.length(); i++)
    if (leksikgrafskiManjaNiska(H, P, i, iMin, n, ss))
      iMin = i;
  return ss.substr(iMin, n);
}
```

4.2 Z-низ

Многи проблеми над нискама могу се ефикасно решити коришћењем *z-низа* (енгл. z-array, z-function); на пример, испитивање да ли се једна ниска јавља унутар друге, одређивање периода ниске (у смислу одређивања најкраће ниске такве да се полазна ниска може представити надовезивањем те ниске одређен број пута), итд. z-низ ниске s дужине n је низ дужине n који на позицији $k = 0, 1, \ldots, n-1$ садржи дужину z_k најдуже подниске (тј. најдужег сегмента) ниске s, која почиње на позицији k и префикс је ниске s. Дакле, $s[0..z_k - 1]$ се поклапа са $s[k..k + z_k - 1]$, а карактери $s[z_k]$ и $s[k + z_k]$ су различити или је пак ниска s дужине $k + z_k$.

Пример 4.2.1

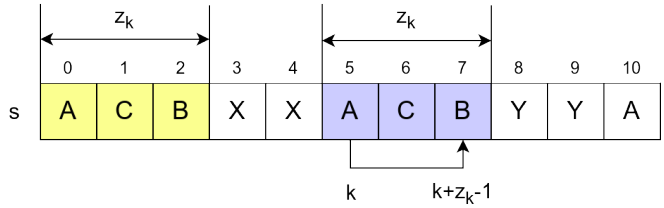
За ниску `alfalfa` *је* $z_3 = 4$*, јер подниска* `alfa` *која почиње на позицији* 3 *и завршава се на крају ниске има дужину* 4 *и поклапа са префиксом ниске дужине* 4*.*

Пример 4.2.2

За ниску `photophosphorescent` *важи* $z_5 = z_9 = 3$*, јер и на позицији* 5 *и на позицији* 9 *почиње подниска* `pho` *која има дужину* 3*, поклапа се префиксом ниске и не може да се продужи тако да се и даље поклапа са одговарајућим префиксом.*

Подниску која почиње на позицији k и преклапа се са неким префиксом дужине z_k називамо и *z-кутија* (енгл. z-box). Пример z-кутије илустрован је на слици 4.1. Ако је вредност $z_k = 0$, тада на позицији k не постоји z-кутија.

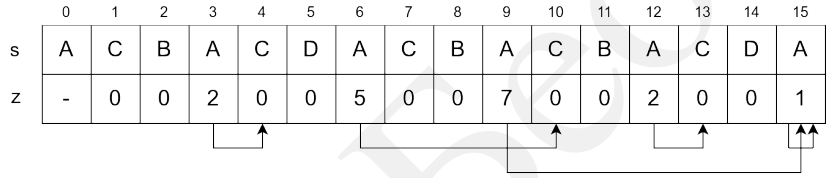
Вредност низа z на позицији 0 једнака је дужини ниске јер је комплетна ниска увек префикс саме себе; међутим та вредност се никада не користи.



Слика 4.1: Пример z-кутије која почиње на позицији 5 и завршава се на позицији 7.

Пример 4.2.3

z-низ ниске `ACBACDACBACBACDA` *приказан је на слици 4.2. Вредност $z_6 = 5$ означава да је подниска* `ACBAC` *(која почиње на позицији 6 и дужине је 5) префикс ниске s, док подниска* `ACBACB` *(која почиње на истој позицији и дужине је 6) није.*



Слика 4.2: Илустрација z-низа. z-кутије су означене стрелицама.

4.2.1 Конструкција z-низа

Једно важно питање је како што ефикасније конструисати z-низ за дату ниску.

Проблем

За дату ниску s конструисати низ z тако да се на свакој позицији k низа z налази вредност z_k једнака највећој дужини подниске ниске s која почиње на позицији k и уједно је и префикс ниске s (тј. карактери ниске s на позицијама $[0, z_k)$ и $[k, k + z_k)$ се поклапају).

4.2.1.1 Алгоритам грубе силе

Директно решење се састоји у томе да се за сваки индекс изнова тражи најдужи поклапајући префикс ниске који почиње на текућој позицији у ниски.

```
vector<int> izracunajZNizTrivijalno(string s) {
  int n = s.size();
```

```
  // inicijalizujemo sve vrednosti z-niza na 0
  vector<int> z(n, 0);

  for (int k = 1; k < n; k++) {
    // sve dok ne izađemo iz opsega niske
    // i odgovarajući karakteri se poklapaju
    while (k+z[k] < n && s[z[k]] == s[k+z[k]]) {
      // inkrementiramo vrednost z-niza na odgovarajucoj poziciji
      z[k]++;
    }
  }
  return z;
}
```

Ово решење садржи две угнежђене петље, а сложеност у најгорем случају је $O(n^2)$. Заиста, за ниску AAA...AAA, алгоритам грубе силе за рачунање z-низа извршава $\Theta(n^2)$ корака. У пракси, очекује се да се на разлике одговарајућих карактера наиђе доста брже.

4.2.1.2 z-алгоритам

z-алгоритам (енгл. z-algorithm) је алгоритам за конструкцију z-низа сложености $O(n)$. Добио је име по структури података која се њоме конструише, тј. по z-низу. До њега је први дошао Густав Хартман 1975. године, али је своју популарност стекао тек касније, након што су Мартин Ескардо и Рајнард Вилхелм представили ефикаснију варијанту алгоритма. Идеја z-алгоритма јесте да се вредности z-низа израчунавају итеративно, слева надесно, на основу претходно израчунатих вредности z-низа.

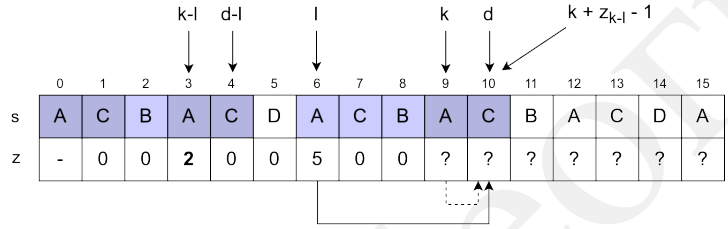
У алгоритму се одржава пар $[l, d]$ који ограничава z-кутију са највећом десном границом d од свих до сада пронађених z-кутија (тада се сегмент $s[l..d]$ поклапа са префиксом $s[0..d-l]$ ниске s). Чињеницу да је $s[0..d-l] = s[l..d]$ користићемо за ефикасније рачунање вредности z-низа на позицијама $l+1, l+2, \ldots, d$.

Дакле, за текући индекс k за који треба израчунати вредност z_k могућа су два случаја:

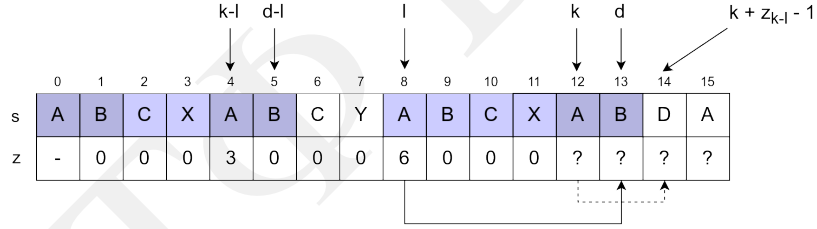
- $l < k \leq d$ – текућа позиција је унутар опсега $[l, d]$, па можемо искористити већ израчунате вредности z-низа за иницијализацију вредности z_k (уместо да крећемо од вредности 0). Наиме, сегменти $s[l..d]$ и $s[0..d-l]$ су једнаки, па пошто је $l < k \leq d$, једнаки су и сегменти $s[k..d]$ и $s[k-l..d-l]$ па приликом рачунања вредности z_k можемо кренути од вредности z_{k-l} која је већ израчуната (слика 4.3). Међутим, вредност z_{k-l} у неким случајевима може бити превелика, јер када се дода позицији k може да буде већа од вредност индекса d, а то није добро јер ми не знамо ништа о карактерима ниске s након позиције d. Рецимо, у примеру

приказаном на слици 4.4 вредност z_k не би било исправно поставити на $z_{k-l} = 3$ јер би нас то извело ван граница опсега $[l, d]$. Дакле, максимална дужина поклопљеног дела може бити једнака броју карактера од текуће позиције k до последње прегледане позиције d, а то је $d - k + 1$. Стога се почетна вредност z_k поставља на $z_k^0 = \min\{d - k + 1, z_{k-l}\}$. Након тога утврђујемо да ли се вредност z_k може повећати покретањем тривијалног алгоритма од позиције $k + z_k^0$.

- $k > d$ – текућа позиција k је ван опсега $[l, d]$, па немамо никаквих информација о позицији k. Стога вредност z_k рачунамо тривијалним алгоритмом, тј. поређењем карактер по карактер ниске s почев од позиција 0 и k;



Слика 4.3: Случај када се z_k иницијализује на z_{k-l}. Провера се наставља поређењем позиција z_{k-l} и $k + z_{k-l}$.



Слика 4.4: Случај када се z_k иницијализује на $d - k + 1$, јер је $k + z_{k-l} > d$. Провера се наставља поређењем позиција $d - k + 1$ и $d + 1$.

Дакле, алгоритам разматра два случаја који се разликују само по иницијализацији вредности z_k, након чега се поступак наставља применом тривијалног алгоритма. Додатно, ако се након иницијализације ниска $s[k..k + z_k]$ цела налази у делу текуће z-кутије без њеног последњег карактера ($s[l, d)$), вредност z_k је већ коначно одређена, па даље карактере не треба поредити. Уколико дође до поклапања дела ниске почев од позиције k са неким префиксом дате ниске и ако је десна граница преклопљеног сегмента већа од претходне вредности десне границе ($k + z_k - 1 > d$), ажурирамо границе $[l, d]$ z-кутије која се простире највише надесно од свих до тада пронађених z-кутија новим границама $[k, k + z_k - 1]$.

Размотримо имплементацију z-алгоритма.

```
// funkcija koja izracunava sve elemente z-niza
vector<int> izracunajZNiz(const string &s) {
  int n = s.size();
  // inicijalizujemo sve vrednosti z-niza na 0
  vector<int> z(n,0);
  int l = 0, d = -1;

  for (int k = 1; k < n; k++) {
    // ako je tekuca pozicija unutar opsega [l,d]
    // koristimo prethodno izracunatu vrednost za inicijalizaciju
    if (k <= d)
      z[k] = min(d-k+1, z[k-l]);

    // ako niska s[k..k+z[k]) pripada delu z-kutije bez njenog poslednjeg
    // karaktera, vrednost z[k] je vec odredjena
    if (z[k] < d-k+1)
       continue;

    // preskacemo proveru karaktera od pozicije k do pozicije k+z[k]-1;
    // od pozicije k+z[k] poredimo karakter po karakter u niski
    // i sve dok se karakteri poklapaju povecavamo vrednost z[k]
    while (k+z[k] < n && s[z[k]] == s[k+z[k]])
      z[k]++;

    // ako je nova vrednost desnog kraja najdesnije z-kutije
    // veca od prethodne vrednosti, azuriramo interval [l,d]
    if (k + z[k] - 1 > d) {
      l = k;
      d = k + z[k] - 1;
    }
  }
  return z;
}
```

Приметимо да се `while` петља може успешно извршити само када је $d - k + 1 \leq z_{k-l}$, односно када постоји поклапање дела ниске од позиције k до позиције d (укључујући и њих) са неким префиксом ниске. Када је $z_{k-l} < d - k + 1$, тада се петља `while` може прескочити. То је урађено у претходном коду (наредбом `continue`). У тим случајевима

услов петље `while` није никада испуњен, па би алгоритам исправно радио чак и када би се наредба `continue` изоставила.

Покажимо да је приказани алгоритам линеарне временске сложености, иако садржи две угнежђене петље. Наиме, након сваког неуспешног поређења карактера у петљи `while`, текућа итерација петље `for` се завршава. Пошто је укупан број итерација спољашње петље `for` једнак n, укупно можемо имати највише n непоклапања карактера у петљи `while`. Сваки карактер ниске s на позицији $k+z_k$ који се успешно поклопи у унутрашњој `while` петљи, више се никада не пореди, те је и број успешно поклопљених карактера у петљи `while` максимално n. Дакле, у свим итерацијама спољашње петље `for`, укупан број корака петље `while` (провера њеног услова и извршавања њеног тела) је $O(n)$, те је укупна сложеност алгоритма $O(n)$.

Може се показати и да свако извршавање тела петље `while` повећава десну границу сегмента d „најдеснијe” z-кутије. Наиме, свака успешна итерација петље `while` поклапа неки нови карактер ниске, десно од границе „најдеснијe” z-кутије, те се вредност променљиве d увећава за онолико колики је број успешних итерација петље `while`. Пошто је максимална вредност за d једнака $n-1$, а почетна је једнака 0, добијамо да се тело петље `while` у свим итерацијама спољашње петље заједно укупно неће извршити више од $n-1$ пута. И одавде следи да је сложеност алгоритма $O(n)$.

Пример 4.2.4

Размотримо конструкцију z-низа за ниску `ACBACDACBACBACDA`.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
s	A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
z	-	0	0	2	?	?	?	?	?	?	?	?	?	?	?	?

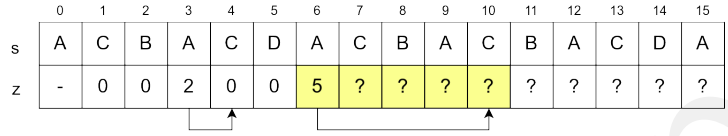
Почетно стање z-низа.

На почетку је $[l,d]=[0,0]$, па вредности z-низа на позицијама 1, 2 и 3 рачунамо тривијалним алгоритмом и добијамо $z_1=z_2=0, z_3=2$. Након израчунате вредности z_3 ажурирамо опсег $[l,d]=[3,4]$.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
s	A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
z	-	0	0	2	?	?	?	?	?	?	?	?	?	?	?	?

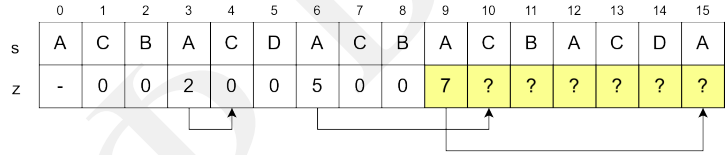
Померање „најдеснијe” z-кутије на позицију $l=3, d=4$.

Након тога, вредност z-низа на позицији $k = 4$ добијамо на основу вредности $z_{k-l} = z_{4-3} = z_1 = 0$. Вредности z_5 и z_6 добијамо покретањем тривијалног алгоритма (јер за $k = 5$ и $k = 6$ и $d = 4$ важи $k > d$) и добијамо $z_5 = 0$, а $z_6 = 5$. Пошто за $k = 6$ важи $k + z_k - 1 = 6 + 5 - 1 = 10$, десна граница текуће z-кутије је већа од претходне вредности 4, па текући $[l, d]$ опсег „најдеснијe" z-кутије постаје $[6, 10]$.



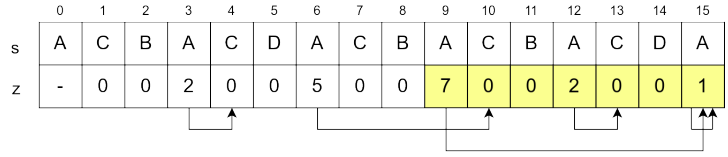
Померање „најдеснијe" z-кутије на позицију $l = 6$, $d = 10$.

Након овог корака, можемо ефикасно израчунати наредне вредности z-низа, јер знамо да су ниске $s[0..4]$ и $s[6..10]$ једнаке. Најпре, пошто је $z_1 = z_2 = 0$ добијамо и да је $z_7 = z_8 = 0$. Након тога, пошто је $z_3 = 2$ и $d - k + 1 = 10 - 9 + 1 = 2$, знамо да је $z_9 \geq 2$. Пошто немамо информације о карактерима ниске након позиције 10, поредимо сегменте даље карактер по карактер. Испоставља се да је $z_9 = 7$. Пошто за $k = 9$ важи $k + z_k - 1 = 9 + 7 - 1 = 15$, десна граница текуће z-кутије је већа од претходне вредности 10, па је нови $[l, d]$ опсег „најдеснијe" z-кутије једнак $[9, 15]$.



Померање „најдеснијe" z-кутије на позицију $l = 9$, $d = 15$.

Након овога, све преостале вредности z-низа могу се одредити на основу информација које се већ налазе у z-низу. Обратимо пажњу само на то да је код последњег елемента z_{15} вредност $z_{k-l} = z_6 = 5$, међутим, она је већа од $d - k + 1 = 1$, па се зато z_{15} иницијализује на $d - k + 1 = 1$ уместо на $z_6 = 5$ и, пошто се дошло до краја низа, на тој вредности и остаје.



Израчунати z-низ.

4.2.2 Претрага текста применом z-низа

У овом поглављу ћемо приказати како се z-низ употребљава за тражење ниске у тексту (тражење подниске узастопних карактера унутар дуже ниске).

Некада се приликом обраде ниски конструише једна дужа ниска која се састоји од већег броја ниски раздвојених специјалним карактерима. То је случај и код алгоритма тражења ниске у тексту заснованог на z-низу. Наиме, у овом случају можемо конструисати ниску $p\#t$, где су ниска која се тражи p и текст t раздвојени специјалним карактером $\#$ који се не јавља ни у p ни у t. Z-низ ниске $p\#t$ нам може указати на то где се у тексту t појављује ниска p јер ће такве позиције имати z-вредност једнаку дужини ниске p. Добијене вредности z-низа на позицијама 0 до m, где је $m = |p|$ дужина ниске p, нису нам битне, јер се односе на позиције унутар ниске p, али их је неопходно израчунати у оквиру z-алгоритма. Могуће је и изоставити специјални карактер $\#$ и тада би се у z-низу тражиле позиције на којима је вредност већа или једнака дужини ниске коју тражимо.

Пример 4.2.5

Размотримо случај када је ниска $p = AABA$, а текст $t = AABAACAADAABAABA$. Формирамо z-низ ниске $p\#t$ (слика 4.5).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	A	B	A	#	A	A	B	A	A	C	A	A	D	A	A	B	A	A	B	A
-	1	0	1	0	4	1	0	2	1	0	2	1	0	4	1	0	4	1	0	1

Слика 4.5: Тражење ниске $AABA$ у тексту $AABAACAADAABAABA$ коришћењем z-низа.

Приметимо да су вредности z-низа на позицијама 5, 14 и 17 једнаке 4, што је дужина ниске p. Пошто је испред текста t додато 5 карактера (4 карактера ниске p и специјални карактер $\#$), те позиције одговарају позицијама 0, 9 и 12 у тексту и на тим позицијама у тексту се јавља ниска p.

Сложеност овог алгоритма је $O(m + n)$, где је $m = |p|$ дужина ниске p, а $n = |t|$ дужина текста t, јер се алгоритам своди на конструкцију z-низа ниске $p\#t$ (за коју смо видели да је линеарне сложености у односу на дужину низа) и пролазак кроз његове вредности.

```
vector<int> traziZNiz(const string& niska, const string& tekst) {
  vector<int> pojavljivanja;
  string zajedno = niska + "#" + tekst;
```

```
    int n = zajedno.size();
    int m = niska.size();
    vector<int> zNiz = izracunajZNiz(zajedno);
    for (int i = 0; i < n; i++)
      if (zNiz[i] == m)
         pojavljivanja.push_back(i - m - 1);
    return pojavljivanja;
  }
```

У овој имплементацији је у првом пролазу одређен садржај z-низа, након чега је претрага вршена у посебном пролазу кроз низ. Приметимо да је могуће ова два пролаза објединити.

4.3 Алгоритам KMP

Директни алгоритам за тражење ниске p дужине $|p| = m$ у тексту t дужине $|t| = n$ пореди ниску p са свим могућим сегментима $t_k t_{k+1} \dots t_{k+m-1}$ текста t, за $k = 0, 1, \dots, n-m+1$. Упоређивање ниске p са текућим сегментом врши се карактер по карактер слева удесно, све док се не установи да су сви карактери ниске једнаки одговарајућим карактерима сегмента (у том тренутку прекида се даље прегледање сегмента) или док се не наиђе на неслагање карактера, односно када се деси $p_i \neq t_{k+i-1}$ за неко i, $0 \leq i \leq m-1$. У другом случају ниска која се тражи се „помера" за један карактер удесно, односно наставља се са провером једнакости карактера p_0 ниске p са карактером t_{k+1} текста t. Број упоређивања карактера је у најгорем случају реда mn (мада ће се у пракси често много брже наићи на неслагање ниске и текућег сегмента), па је сложеност овог алгоритма $O(mn)$ у најгорем случају.

У описаном директном алгоритму за тражење ниске у тексту након померања ниске за једно место удесно игноришу се све информације о претходно поклопљеним карактерима. Стога је могуће да се један исти карактер текста више пута обрађује тј. да се пореди са различитим карактерима ниске.

Пример 4.3.1

Размотримо проблем тражења ниске $p = aaaab$ у тексту $t = aaaaaaaaaa$. Након успешног поређења четири карактера a текста t и ниске p и неподударања карактера a текста и карактера b ниске p, ниску p бисмо померили удесно за једну позицију. Након тога бисмо поново (успешно) поредили четири карактера a текста t и ниске p (иако смо за три од њих већ могли да закључимо да ће се ти карактери поклопити) а затим након неподударања карактера a са карактером b опет померили ниску p за једну позицију удесно итд.

Алгоритам који су независно осмислили Кнут[4], Морис[5] и Прат[6], а који је познат под називом *Кнут-Морис-Пратов алгоритам* (или скраћено *алгоритам КМР*), заснива се на идеји да се искористе информације добијене претходним поређењима карактера и да се никада изнова не пореде карактери текста t који су се претходно успешно поклопили са ниском p. Видећемо да се на овај начин тражења ниске у тексту има сложеност $O(n)$. Да би ово било изводљиво, на почетку алгоритма врши се предобрада ниске p у циљу анализе њене структуре. Видећемо и да предобрада има сложеност $O(m)$, те је укупна сложеност алгоритма КМР $O(m+n)$.

Пре описа алгоритма КМР уведимо основну терминологију која се у њему користи. Нека је $x = x_0 \dots x_{k-1}$ ниска дужине k. Важи следеће:

- ниска y дужине l $(0 \leq l \leq k)$ је *префикс* ниске x ако је $y = x_0 \dots x_{l-1}$;
- ниска y дужине l $(0 \leq l \leq k)$ је *суфикс* ниске x ако је $y = x_{k-l} \dots x_{k-1}$;
- за префикс (суфикс) y кажемо да је *прави префикс* (*прави суфикс*) ниске x ако је $y \neq x$, односно ако је његова дужина строго мања од дужине ниске x;
- ниска y дужине l је *префикс-суфикс* ниске x ако је $0 \leq l < k$, $y = x_0 \dots x_{l-1}$ и истовремено $y = x_{k-l} \dots x_{k-1}$, односно ако је ниска y истовремено и прави префикс и прави суфикс ниске x.

Пример 4.3.2

Нека је $x = abacab$. Означимо са ϵ празну ниску. Прави префикси ниске x су $\epsilon, a, ab, aba, abac, abaca$, а прави суфикси $\epsilon, b, ab, cab, acab, bacab$. Дакле, префикс-суфикси ниске x су ϵ (дужине 0) и ab (дужине 2).

Пример 4.3.3

Префикс-суфикси ниске $x = aaaaa$ су $\epsilon, a, aa, aaa, aaaa$.

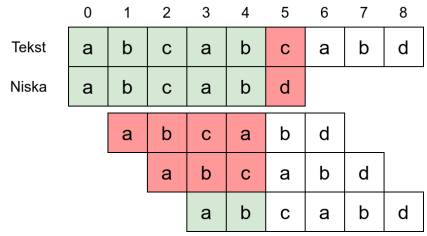
Празна ниска је увек префикс-суфикс ниске x, осим ако је ниска x празна (празна ниска нема префикс-суфикс).

У наредном примеру илустроваћемо како појам префикс-суфикса може помоћи приликом израчунавања вредности за коју треба померити ниску p у односу на текст t, када дође до непоклапања карактера ниске p и текста t.

[4] Доналд Кнут (енгл. Donald Knuth), рођен 1938. године, амерички информатичар и математичар.
[5] Џејмс Морис (енгл. James Morris), рођен 1941. године, амерички информатичар.
[6] Вон Прат (енгл. Vaughan Pratt), рођен 1944, аустралијски информатичар.

Пример 4.3.4

На слици 4.6 приказан је почетак тражења ниске $abcabd$ унутар текста $abcabcabd$.



Слика 4.6: Пример претраге ниске.

Карактери $abcab$ на позицијама од 0 до 4 у ниски p и тексту t су се поклопили, док се карактери на позицији 5 разликују (у тексту t се налази карактер c, а у ниски p карактер d). Интересује нас за колико најмање треба померити ниску p удесно у односу на текст t, тако да се поново неки (краћи) префикс ниске p поклопи са текстом t.

- *Да би се поклапање догодило након померања за једно место удесно, потребно је да се део текста $bcab$ поклопи са делом ниске $abca$, што није случај. Приметимо да је $bcab$ суфикс, а $abca$ префикс дела ниске (префикса ниске) $abcab$ за који смо утврдили да се поклапа са текстом.*
- *Да би се поклапање догодило након померања за два места удесно, потребно је да се део текста cab поклопи са делом ниске abc, што није случај. Опет можемо приметити да је cab суфикс, а abc префикс дела ниске (префикса ниске) $abcab$ за који смо утврдили да се поклапа са текстом.*
- *Да би се поклапање догодило након поменрања за три места удесно, потребно је да се део текста ab поклопи са делом ниске ab, што овај пут јесте случај. Ниска ab је и суфикс и префикс дела ниске (префикса ниске) $abcab$ и заправо је најдужи префикс-суфикс тог поклопљеног дела ниске.*

Дакле, неколико последњих карактера поклопљеног префикса ниске p треба да се поклопе са неколико првих карактера тог префикса ниске p, те је нама, у ствари, потребан неки префикс-суфикс поклопљеног префикса ниске p. С обзиром на то да желимо да се померимо удесно што је мање могуће, ми у ствари тражимо најдужи префикс-суфикс поклопљеног префикса ниске p.

У овом примеру, поклопљени префикс ниске p је $abcab$ и његова дужина је 5. Његов најдужи префикс-суфикс је ab и дужине је 2. Дакле, ниску померамо удесно у односу на текст за $5 - 2 = 3$ карактера.

Ако је ниска $abcabd$ и текст почиње са $abcdxy$, поклопљени префикса ниске је abc и дужина му је 3. Његов најдужи префикс-суфикс је празна ниска ε и дужина му је 0, те ниску померамо удесно за $3 - 0 = 3$ карактера.

Пошто дужина поклопљеног дела ниске p и текста t зависи и од самог текста t, односно не знамо унапред за које префиксе ниске p ће нам бити потребни најдужи префикси-суфикси, у фази предобраде потребно је одредити дужину најдужег префикс-суфикса сваког префикса дате ниске p. Након тога, у фази претраге, на основу префикса ниске p који се поклопио са текстом t, тј. дужине његовог најдужег префикс-суфикса, једноставно се утврђује за колико места треба померити ниску p у односу на текст t.

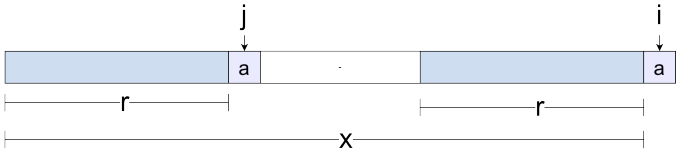
Приметимо велику сличност низа дужина префикс-суфикса и z-низа (описаног у поглављу 4.2). На свакој позицији у z-низу се налази дужина најдуже подниске која *почиње* на тој позицији и једнака је неком префиксу ниске, док се код алгоритма KMP за сваку позицију одређује најдужи прави префикс-суфикс, што је дужина најдуже подниске која се *завршава* на тој позицији и једнака је неком префиксу ниске.

4.3.1 Предобрада ниске која се тражи

Посветимо се сада питању како се за дату ниску $p = p_0 \dots p_{m-1}$ дужине m могу ефикасно одредити дужине најдужих префикс-суфикса свих њених префикса. У фази предобраде ниске p израчунаћемо вредности низа π дужине $m + 1$ који ће чувати вредности *префиксне функције* (енгл. prefix function) ниске p, тако да је π_i дужина најдужег префикс-суфикса префикса $p_0 \dots p_{i-1}$ дужине i дате ниске p. Применићемо индуктивни приступ, тј. дужине најдужих префикс-суфикса ћемо израчунавати инкрементално.

Базу индукције чини случај $i = 0$, тј. случај празног префикса ϵ (префикса дужине 0) ниске p. Он нема префикс-суфиксе, а вредност π_0 ћемо поставити на -1.

Претпоставимо да смо одредили вредности $\pi_0, \dots, \pi_i$ и размотримо како се на основу тих вредности може одредити вредност π_{i+1}. Нека је $x = p_0 \dots p_{i-1}$. Вредност π_{i+1} једнака је дужини најдужег префикс-суфикса ниске $p_0 \dots p_{i-1}p_i = xp_i$. Сваки префикс-суфикс ниске xp_i добија се тако што се неки префикс-суфикс r (дужине $j < i$) ниске x прошири карактером $a = p_i$ (слика 4.7).



Слика 4.7: Проширивање префикс-суфикса.

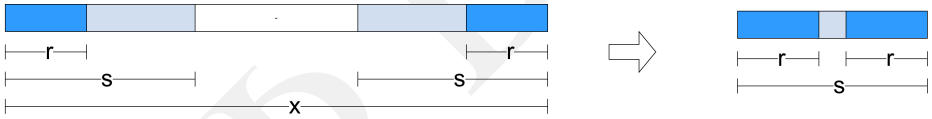
Да би префикс-суфикс ниске xp_i био што дужи, потребно је да и префикс-суфикс r буде што дужи. Дакле, до најдужег префикс-суфикса ниске xp_i можемо доћи тако што анализирамо редом префикс-суфиксе ниске x од најдужег, ка најкраћем (а то је ε). Ако за неки префикс-суфикс $r = p_0 \dots p_{j-1}$ дужине j важи $p_j = p_i$, тада је најдужи префикс-суфикс ниске xp_i ниска rp_i, па је $\pi_{i+1} = j+1$. Ако ни за један префикс-суфикс r ниске x то не важи, тада је једини префикс-суфикс ниске xp_i празна ниска ϵ, па је $\pi_{i+1} = 0$.

Остаје још питање како набројати све префикс-суфиксе ниске $x = p_0 \dots p_{i-1}$. Дужина најдужег од њих је позната и једнака је π_i, па је најдужи префикс-суфикс ниске x ниска $p_0 \dots p_{\pi_i - 1}$. Наредна лема нам даје начин да познајући један префикс-суфикс ниске x одредимо и све остале.

Лема 4.3.1

Нека су r и s префикс-суфикси ниске x, при чему важи $|r| < |s|$. Тада је ниска r префикс-суфикс ниске s.

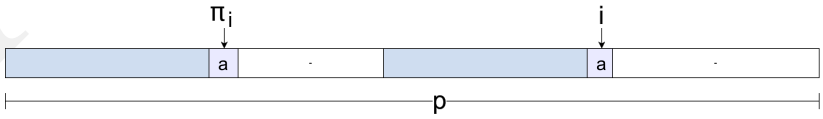
Доказ. Ниска r је префикс ниске x, па је и прави префикс ниске s, јер је краћа од ње. Ниска r је истовремено суфикс ниске x, те је стога и прави суфикс ниске s. Стога је r префикс-суфикс ниске s (слика 4.8). □



Слика 4.8: Префикс-суфикси r и s ниске x.

Дакле, ако је ниска r_1 најдужи префикс-суфикс ниске x, следећи најдужи префикс-суфикс r_2 ниске x се добија као најдужи префикс-суфикс ниске r_1. Наредни најдужи префикс-суфикс r_3 ниске x је најдужи префикс-суфикс ниске r_2 и тако даље.

Пошто је π_i дужина најдужег префикс-суфикса ниске x, карактери $p_0, p_1, \dots, p_{\pi_i - 1}$ и $p_{i-\pi_i}, \dots, p_{i-1}$ ниске се поклапају и треба проверити да ли је $p_{\pi_i} = p_i$ (слика 4.9).



Слика 4.9: Префикс дужине i ниске са префикс-суфиксом дужине π_i.

Ако се не поклапају, разматра се први наредни краћи префикс-суфикс ниске x. Он ће бити једнак најдужем префикс-суфиксу најдужег префикс-суфикса префикса ниске

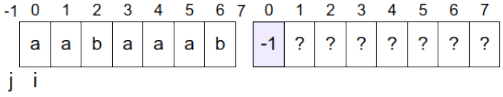
дужине i. Дакле, он ће бити једнак префиксу ниске x дужине π_{π_i}, те је потребно упоредити карактере $p_{\pi_{\pi_i}}$ и p_i. Ако се ти карактери не поклопе, поступак се наставља на исти начин.

Дакле, приликом рачунања вредности π_{i+1}, променљива j ће у петљи узимати редом вредности $\pi_i, \pi_{\pi_i}, \pi_{\pi_{\pi_i}}$ ... Префикс-суфикс дужине j може се проширити карактером p_i ако је $p_j = p_i$. Ако то није случај, наредни најдужи префикс-суфикс ниске x се одређује постављањем $j = \pi_j$. Петља се прекида ако се ниједан префикс-суфикс не може проширити. Захваљући томе што смо поставили $\pi_0 = -1$, када се то деси, петља се прекида када је $j = -1$, јер услов $j \geq 0$ није испуњен. На крају извршавања петље вредност променљиве j увећана за 1 садржаће дужину најдужег префикс-суфикса ниске $x = p_0 \dots p_i$ и њу треба сместити на позицију π_{i+1} (ово важи и у случају када је дошло до поклапања $p_j = p_i$, али, опет захваљујући одлуци да је $\pi_0 = -1$, и када је петља прекинута јер је $j = -1$, што значи да је празна реч најдужи префикс-суфикс).

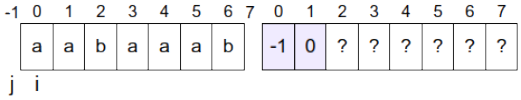
```
vector<int> kmpIzracunajPrefiksSufikse(const string &p) {
  int m = p.size() + 1;
  vector<int> pi(m);
  int i = 0;
  int j = -1;
  // prazna niska nema prefiks-sufikse
  pi[i] = j;
  // za prefiks duzine i polazne niske racunamo najduzi prefiks-sufiks
  while (i + 1 < m) {
    // proveravamo da li se najduzi prefiks-sufiks prefiksa uzorka duzine i
    // moze prosiriti: to vazi ako je p[i] = p[j]
    while (j >= 0 && p[i] != p[j])
      j = pi[j];
    pi[++i] = ++j;
  }
  return pi;
}
```

Пример 4.3.5

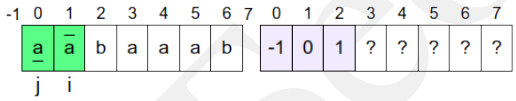
Прикажимо предобраду, тј. израчунавање најдужих префикс-суфикса ниске aabaaab. Након иницијализације променљивих, важи $j = -1$, $i = 0$, $\pi_0 = -1$.



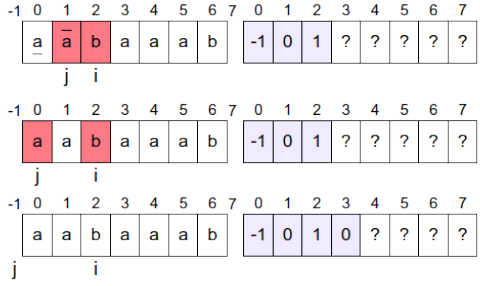
- *Сада одређујемо вредност π_1 тј. дужину најдужег префикс-суфикса ниске a (тј. префикса ниске p дужине 1, који се завршава на позицији $i = 0$). Њен једини префикс-суфикс је празна ниска, па је $\pi_1 = 0$ (у унутрашњу петљу се не улази јер је $j = -1 < 0$). Ово је илустровано на наредној слици.*



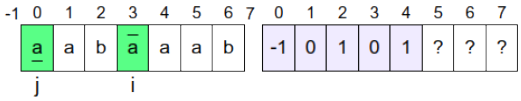
- *Након тога се увећавају вредности променљивих i и j и постају $i = 1$ и $j = 0$ (тада се врши додела `pi[1] = 0`) и прелази се на одређивање вредности π_2 тј. дужине најдужег префикс-суфикса ниске aa (тј. префикса ниске p дужине 2, који се завршава на позицији $i = 1$). Најдужи префикс-суфикс ниске a је празна ниска, а пошто је $p_j = p_0 = a = p_1 = p_i$, он се може продужити карактером a, па је a најдужи префикс-суфикс ниске aa и $\pi_2 = 1$. Ово је илустровано на наредној слици.*



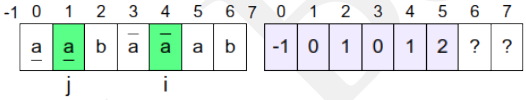
- *Након тога се увећавају вредности променљивих i и j и постају $i = 2$ и $j = 1$ (тада се врши додела `pi[2]=1`) и прелази се на одређивање вредности π_3 тј. дужине најдужег префикс-суфикса ниске aab (тј. префикса ниске p дужине 3, који се завршава на позицији $i = 2$). Најдужи префикс-суфикс ниске aa је a, али пошто је $p_j = p_1 = a \neq b = p_2 = p_i$, он се не може продужити. Зато анализирамо краћи префикс-суфикс ниске aa, а то је празна реч (постављајући вредност j на $\pi_j = \pi_1 = 0$). Пошто је $p_j = p_0 = a \neq p_2 = p_i$, ни он се не може продужити. Постављајући j на вредност $\pi_j = \pi_0 = -1$, прекидамо унутрашњу петљу и закључујемо да је најдужи префикс-суфикс ниске aab празна ниска и $\pi_3 = 0$. Ово је илустровано на наредној слици.*



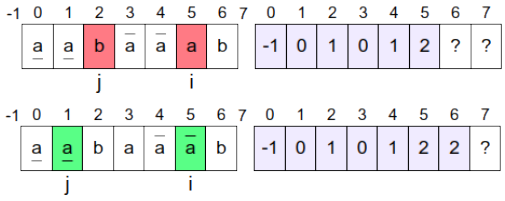
- *Након тога се увећавају вредности променљивих* i *и* j *и постају* $i = 3$ *и* $j = 0$ *(тада се врши додела* `pi[3]=0`*) и прелази се на одређивање вредности* π_4 *тј. дужине најдужег префикс-суфикса ниске* $aaba$ *(тј. префикса ниске* p *дужине* 4*, који се завршава на позицији* $i = 3$*). Најдужи префикс-суфикс ниске* aab *је празна ниска, а пошто је* $p_j = p_0 = a = p_3 = p_i$*, он се може продужити карактером* a*, па је* a *најдужи префикс-суфикс ниске* $aaba$ *и* $\pi_4 = 1$*. Ово је илустровано на наредној слици.*



- *Након тога се увећавају вредности променљивих* i *и* j *и постају* $i = 4$ *и* $j = 1$ *(тада се врши додела* `pi[4]=1`*) и прелази се на одређивање вредности* π_5 *тј. дужине најдужег префикс-суфикса ниске* $aabaa$ *(тј. префикса ниске* p *дужине* 5*, који се завршава на позицији* $i = 4$*). Најдужи префикс-суфикс ниске* $aaba$ *је ниска* a*, а пошто је* $p_j = p_1 = a = p_4 = p_i$*, он се може продужити карактером* a*, па је* aa *најдужи префикс-суфикс ниске* $aabaa$ *и* $\pi_5 = 2$*. Ово је илустровано на наредној слици.*

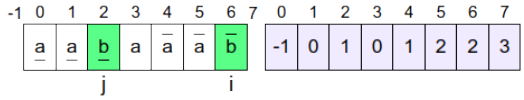


- *Након тога се увећавају вредности променљивих* i *и* j *и постају* $i = 5$ *и* $j = 2$ *(тада се врши додела* `pi[5]=2`*) и прелази се на одређивање вредности* π_6 *тј. дужине најдужег префикс-суфикса ниске* $aabaaa$ *(тј. префикса ниске* p *дужине* 6*, који се завршава на позицији* $i = 5$*). Најдужи префикс-суфикс ниске* $aabaa$ *је ниска* aa*, али пошто је* $p_j = p_2 = b \neq a = p_5 = p_i$*, он се не може продужити. Зато анализирамо краћи префикс-суфикс ниске* $aabaa$*, а то је* a *(постављајући вредност* j *на* $\pi_j = \pi_2 = 1$*). Пошто је сада* $p_j = p_1 = a = p_5 = p_i$*, овај префикс-суфикс се може продужити карактером* a*, па је* aa *најдужи префикс-суфикс ниске* $aabaaa$ *и* $\pi_6 = 2$*. Ово је илустровано на наредној слици.*



- *Након тога се увећавају вредности променљивих* i *и* j *и постају* $i = 6$ *и* $j = 2$

(тада се врши додела `pi[6]=2`) и прелази се на одређивање вредности π_7 тј. дужине најдужег префикс-суфикса ниске $aabaaab$ (тј. префикса ниске p дужине 7, који се завршава на позицији $i = 6$). Најдужи префикс-суфикс ниске $aabaaa$ је ниска aa, а пошто је $p_j = p_2 = b = p_6 = p_i$, он се може продужити карактером b, па је aab најдужи префикс-суфикс ниске $aabaaab$ и $\pi_7 = 3$. Ово је илустровано на наредној слици.



Овим су одређени најдужи префикс-суфикси свих префикса ниске p, па се алгоритам завршава (јер након увећавања вредности променљивих на $j = 3$ и $i = 7$, вредност i достиже дужину ниске $m = 7$).

Дужине свих префикс-суфикса неке ниске можемо пронаћи и коришћењем z-низа. Наиме, ако за неку позицију k важи $k + z_k = n$, где је n дужина ниске, тада се подниска која почиње на позицији k и једнака је неком префиксу ниске простире до самог краја ниске и она је уједно и суфикс ниске. И други смер важи: за сваки префикс-суфикс дужине d важи $z_{n-d} = d$ тј. $n - d + z_{n-d} = n$. На пример, у наредној табели је приказан z-низ за ниску $abacabacaba$.

k	0	1	2	3	4	5	6	7	8	9	10
s_k	a	b	a	c	a	b	a	c	a	b	a
z_k	-	0	1	0	7	0	1	0	3	0	1
π_{k+1}	0	0	1	0	1	2	3	4	5	6	7

Важи $4 + z_4 = 8 + z_8 = 10 + z_{10} = 11$, па су прави префикс-суфикси ниске $abacaba$, aba и a.

На основу познатог z-низа могуће је одредити и цео KMP-низ π. Ако је $z_k > 0$, тада је $s[0..z_k) = s[k..k + z_k)$. Зато и за свако $j < z_k$ важи да је $s[0..j] = s[k..k + j]$, па постоји прави префикс-суфикс дужине $j + 1$ ниске која се завршава на позицији $k + j$. На пример, пошто је у текућем примеру $z_4 = 7$, сигурно постоји префикс-суфикс дужине 1 ниске која се завршава на позицији 4 (ово се добија за $k = 4$, а $j = 0$ и тада је a префикс-суфикс ниске $abaca$). Слично, сигурно постоји префикс-суфикс дужине 7 ниске која се завршава на позицији 10 (ово се добија за $k = 4$, $j = 6$ и тада је $abacaba$ префикс-суфикс целе ниске). Овако одређен префикс-суфикс не мора бити и најдужи. На пример, пошто је $z_8 = 3$, постоји и префикс-суфикс дужине 3 ниске која се завршава на позицији 10 (ово се добија за $k = 8$, $j = 2$ и тада је aba

префикс-суфикс целе ниске). Ипак, ако се вредности k обрађују редом, први пронађен префикс-суфикс за сваку позицију биће и најдужи, па се обрада префикс-суфикса за ту позицију добијених на основу наредних позиција k може прескочити. Ово доводи до алгоритма линеарне сложености (за свако наредно k довољно је обрађивати само оне вредности $j < z_k$, за које је $k + j$ десно од последње до тада одређене вредности низа π).

Могуће је и на основу познатих вредности низа π добити вредности низа z.

4.3.2 Претраживање текста

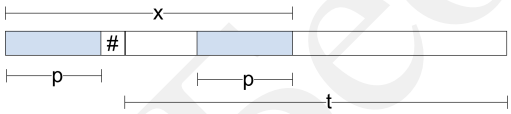
Основна варијанта претраге текста се може добити прилагођавањем алгоритма грубе силе. Спољашња петља обилази позиције i у тексту са којих се почиње претрага ниске. У унутрашњој петљи се пролази кроз карактере ниске (од почетка) и карактере текста (од позиције i) све док се не дође до краја ниске или до различитог карактера. Ако се дође до краја ниске, пронађено је поклапање. Ако није, алгоритам грубе силе увећава позицију i за 1, док се код алгоритма KMP позиција i увећава за $j - \pi_j$, где је j позиција у ниски на којој се наишло на непоклапајући карактер (тј. дужина успешно поклопљеног префикса ниске), а π_j дужина најдужег префикс-суфикса тог дела ниске. Пошто је већ установљено да се првих π_j карактера ниске p након померања поклапају са текстом, поређење се наставља од карактера π_j те ниске тј. променљива j се поставља на π_j (при чему треба обратити пажњу на специјални случај $j = 0$ тј. $\pi_0 = -1$ треба посебно обрадити).

```
vector<int> traziKMP(const string &p, const string &t) {
  // duzine prefiks-sufiksa svih prefiksa niske p
  vector<int> b = kmpIzracunajPrefiksSufikse(p);
  // pozicije poklapanja u tekstu
  vector<int> pozicije;
  // proveravamo pojavljivanje nniske krenuvsi od pozicije i u tekstu
  int i = 0, j = 0;
  while (i + p.size() <= t.size()) {
    // analiziramo jedan po jedan karakter niske
    while (j < p.size()) {
      if (t[i + j] != p[j])
        break;
      j++;
    }
    // ako smo stigli do kraja niske, ona je pronadjena na poziciji i u tekstu
    if (j == p.size())
```

```
      pozicije.push_back(i);
    // pomeramo se udesno na osnovu algoritma KMP i znamo da se prvih b[j]
    // karaktera vec poklapaju
    i += j - b[j];
    j = max(b[j], 0);
  }
  return pozicije;
}
```

Могуће су и другачије имплементације.

Слично као код z-алгоритма, проналазак свих појављивања ниске p у тексту t се може свести на предобраду ниске $p\#t$, где је са $\#$ означен специјални карактер који се не јавља ни у тексту t ни у ниски p. Тада сваком појављивању ниске p у тексту t одговара по један префикс-суфикс дужине m (слика 4.10).

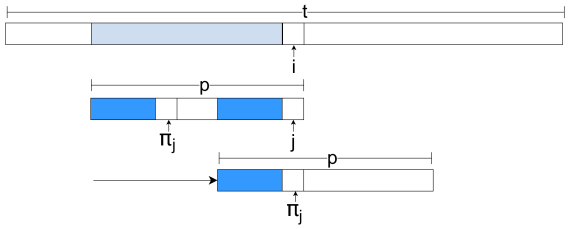


Слика 4.10: Префикс-суфикс дужине $m = |p|$ префикса x ниске $p\#t$ одговара појављивању префикса p у тексту t.

Алтернативно, другу фазу, тј. алгоритам којим се тражи ниска p у тексту t на основу познатих дужина префикс-суфикса добијених у фази предобраде ниске, можемо имплементирати у виду засебне функције. Тај алгоритам је врло сличан алгоритму предобраде ниске, једино што се пореде карактери t_i (карактер на позицији i у тексту t) и p_j (карактер на позицији j у ниски p).

Када се у унутрашњој `while` петљи наиђе на непоклапање, разматра се најдужи префикс-суфикс поклапајућег префикса дужине j ниске (слика 4.11) и ниска се помера тако што се вредност j замени њеном дужином π_j. То се ради све док се карактери p_j и t_i не поклопе или док се закључи да крађи префикс-суфикс префикса ниске не постоји ($j = -1$). Након тога имамо нови поклапајући префикс ниске и настављамо са спољашњом `while` петљом. Уколико смо наишли на поклапање свих m карактера ниске (када важи $j = m$), евидентирамо да смо пронашли ниску у тексту почев од позиције $i - j$. Након тога, ниска се помера удесно у односу на текст на основу свог најдужег префикс-суфикса.

Имплементација претраге у језику C++ може бити следећа:



Слика 4.11: Када се нађе на непоклапање карактера p_j (карактера на позицији j у ниски) и карактера t_i (карактера на позицији i у тексту), прелази се на поређење карактера p_{π_j} (карактера на позицији π_j у ниски) и карактера t_i, што одговара померању ниске удесно за $j - \pi_j$.

```
vector<int> traziKMP(const string& p, const string& t) {
  vector<int> pozicije;
  vector<int> pi = kmpIzracunajPrefiksSufikse(p);
  int n = t.size();
  int m = p.size();
  int i = 0;
  int j = 0;
  // proveravamo tekuci karakter teksta sve dok postoji mogućnost da se
  // do kraja teksta pronadje niska
  while (i + m - j <= n) {
    // proveravamo da li se najduzi prefiks-sufiks
    // prefiksa niske p koji se zavrsava na prethodnoj poziciji
    // moze prosiriti: to vazi ako je t[i] = p[j]
    while (j >= 0 && t[i] != p[j])
      j = pi[j];
    i++; j++;
    // ako smo poklopili svih m karaktera niske pamtimo poziciju poklapanja
    if (j == m)
      pozicije.push_back(i - m);
  }
  return pozicije;
}
```

Пример 4.3.6

Вратимо се на пример 4.3.1 и размотримо како би претрага изгледала применом алгоритма КМР.

Дужине префикс-суфикса ниске $p = aaaab$ су одређене следећом табелом.

	0	1	2	3	4	5
π	-1	0	1	2	3	0

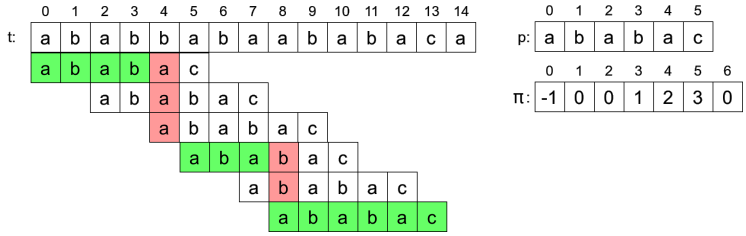
- *Претрага текста $t = aaaaaaaaaa$ почиње од позиције 0 и наилази се на неподударање на позицији 4 у ниски p. Зато се тражи најдужи префикс-суфикс прва 4 карактера ниске p (тј. њеног префикса $aaaa$). Његова дужина је 3, па се у тексту померамо надесно за $4-3$ тј. за један карактер. Пошто знамо да се префикс узорка дужине 3 већ поклапа са текстом, поређење може да се настави на позицији $j = \pi_j = 3$ узорка (који се пореди са карактером на позицији $i+j = 1+3 = 4$ у тексту).*
- *Пошто је $p_3 = t_4 = a$, увећавају се i и j и наилази се на следеће неподударање $p_4 = b \neq a = t_5$. До неподударања је дошло поново на позицији 4 у ниски p, па се текст поново помера за једну позицију и поступак се на практично исти начин наставља до краја текста.*

Приметимо да смо у сваком (осим првом) кораку спољашње петље вршили само два поређења карактера (једно успешно и једно неуспешно). Исто би се дешавало и да је ниска p много дужа, што је значајно унапређење у односу на наивни алгоритам који би у сваком кораку унутрашње петље поредио све карактере ниске p.

Пример 4.3.7

Размотримо која се поређења извршавају у алгоритму КМР, приликом тражења ниске $ababac$ у тексту $ababbabaabababaca$. Успешна поређења карактера су приказана зеленом, а поређења код којих се није нашло на поклапање, црвеном бојом (слика 4.12).

На почетку се успешно поклапају карактери $abab$ након чега се наилази на разлику између карактера $t_4 = b$ и $p_4 = a$. Пошто је ab најдужи префикс-суфикс поклопљеног дела $abab$ и његова дужина је 2 (тј. пошто је $\pi_4 = 2$), вредност променљиве j се поставља на 2 и пореде се карактери $t_4 = b$ и $p_2 = a$. Поново нема поклапања. Пошто је сада поклопљена ниска ab, а њен најдужи префикс-суфикс је празна реч (тј. пошто је $\pi_2 = 0$), прелази се на поређење карактера $t_4 = b$ и $t_0 = a$. И они су различити, важи $\pi_0 = -1$, што значи да се i увећава, j поставља на нулу и прелази се на карактер $t_5 = a$. Успешно се поклапа префикс aba, након чега се наилази на разлику $t_8 = a$,



Слика 4.12: Друга фаза алгоритма KMP.

а $p_3 = b$. Пошто је a најдужи префикс-суфикс поклопљеног дела aba (тј. пошто је $\pi_3 = 1$), променљива j се поставља на 1 и пореде се карактери $t_8 = a$ и $p_1 = b$. Пошто су и они различити, а пошто је празна реч најдужи префикс-суфикс поклопљеног дела a (тј. пошто је $\pi_1 = 0$), променљива j се поставља на 0 и пореде се карактери $t_8 = a$ и $p_0 = a$. Ти карактери су једнаки, па се са поређењем наставља даље и успешно се пронализи подниска. Пошто је празна реч једини префикс суфикс целе ниске (тј. пошто је $\pi_6 = 0$), наредни карактер би требало поредити са карактером 0 ниске, међутим, дошло се до краја текста и друга појављивања ниске не постоје.

Која је сложеност функције за предобраду ниске p дужине $|p| = m$, приказане у поглављу 4.3.1? Унутрашња петља смањује вредност промењиве j бар за 1, јер је $\pi_j < j$. Петља се завршава најкасније када вредност j постане -1 (а на почетку је j једнако -1), те се стога може смањити највише онолико пута колико је претходно била повећана наредбом `j++`. С обзиром на то да се наредба `j++` извршава у спољашњој петљи тачно m пута, укупан број извршавања унутрашње `while` петље је ограничен са m, те је укупна сложеност предобраде ниске $O(m)$. Потпуно аналогно се може закључити да је сложеност алгоритма за тражење ниске p у тексту t дужине $|t| = n$ једнака $O(n)$, те је укупна сложеност алгоритма KMP $O(m + n) = O(|p| + |t|)$.

У примеру 4.3.7 смо имали фину илустрацију сложености фазе претраге, с обзиром на то да поређења (успешна и неуспешна) формирају „степенице” које у најгорем случају могу бити једнако високе и дуге, те је у најгорем случају потребно $2n$ поређења приликом тражења ниске у тексту.

4.3.3 Испитивање периодичности ниске

Проналажење најдужих префикс-суфикса ниски има и друге примене. Илуструјмо како се они могу употребити да би се испитало да ли је ниска периодична и како би се пронашао њен најкраћи период.

Реч w је *периодична* ако постоји непразна реч $p = p_1 p_2$ и природан број $n \geq 2$, тако да је $w = p^n p_1$. На пример, реч $abacabacabacab$ је периодична јер се у њој понавља реч

$abac$, при чему се последње понављање не завршава цело већ се зауставља са ab, тј. реч је једнака $(abac)^3ab$. Ако је p_1 празна ниска, тада можемо рећи да је ниска *строго периодична*. Ако бисмо допустили да је $n = 1$, тада би свака ниска била периодична.

Испитивање периодичности ниске игра важну улогу у различитим доменима. Некада се користи као мера самосличности у применама које се тичу обраде текста, анализе података, рачунарске биологије и сл.

Проблем

Написати алгоритам који проверава да ли је реч w периодична.

Проблем можемо решити грубом силом, тако што ћемо за сваку вредност d такву да је $2d \leq |w|$ проверити да ли је реч периодична, при чему је период префикс речи w дужине d. Једноставно се доказује да је реч w периодична са периодом p чија је дужина d ако и само ако за свако i за које је $0 \leq i$ и $i + d < |w|$ важи $w_i = w_{i+d}$. Да бисмо проверили да ли је ниска строго периодична, довољно је додатно проверити да ли дужина речи w дељива бројем d. Проблем се онда решава са две угнежђене линеарне претраге – у спољној проверавамо све потенцијалне вредности дужине d, а у унутрашњој проверавамо да ли постоји вредност i таква да је $w_i \neq w_{i+d}$. Ако у унутрашњој петљи утврдимо да такво i не постоји, тада је ниска периодична. Ако пронађемо такво i, можемо прекинути унутрашњу петљу (реч није периодична са периодом дужине d) и прећи на следеће d (за један веће). Ако такво d не постоји, тада можемо констатовати да реч није периодична.

```
// provera da li niska stri ima period duzine p
bool proveriPeriod(const string& str, int p) {
    for (int i = 0; i + p < str.size(); i++)
       if (str[i] != str[i + p])
          return false;
    return true;
}

// provera da li je niska periodicna
bool periodicna(const string &str) {
  // proveravamo za svaku mogucu duzinu perioda
  for (int p = 1; 2 * p <= str.size(); p++) {
      if (proveriPeriod(str, p))
         return true;
  return false;
}
```

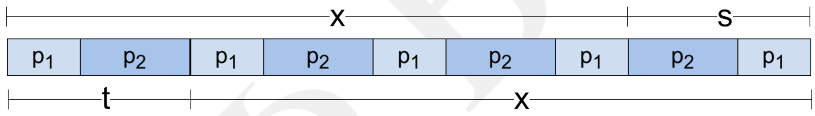
Сложеност најгорег случаја овог алгоритма је квадратна. Заиста, унутрашња линеарна претрага може у најгорем случају захтевати $O(|w|)$ итерација, и она се понавља $O(|w|)$ пута. Ипак, ако је ниска насумична, реално је очекивати да ће се за већину вредности d веома брзо установљавати да је $w_i \neq w_{i+d}$, па програм може радити доста брже од најгорег случаја.

Ефикасније решење се заснива на наредној леми:

Лема 4.3.2 **[Карактеризација периодичне ниске]**

Ниска w је периодична ако и само ако има префикс-суфикс x чија је дужина најмање половина ниске тј. ако постоје непразни x, s и t такви да је $w = xs = tx$ и $2|x| \geq |w|$, тј. ако је $\pi_{|w|} \geq |w|/2$.

Доказ. Претпоставимо да је ниска w периодична и покажимо да она има префикс-суфикс дужине бар половине ниске. Из услова да је ниска w периодична следи да постоји непразна реч $p = p_1p_2$ тако да је $w = p^np_1$, за неко $n \geq 2$. Тада је $t = p_1p_2 = p$, $x = p^{n-1}p_1$, док је $s = p_2p_1$ (слика 4.13). Заиста, важи $xs = p^{n-1}p_1 \cdot p_2p_1 = p^np_1 = w$ и $tx = p \cdot p^{n-1}p_1 = p^np_1 = w$.



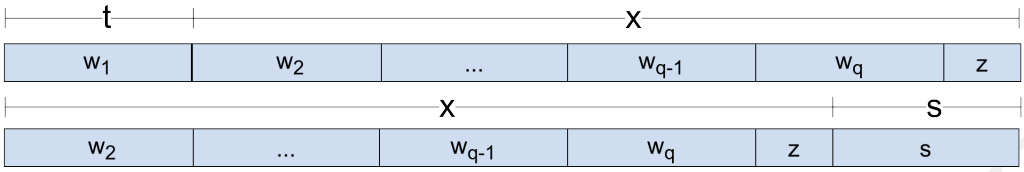
Слика 4.13: x је префикс-суфикс периодичне ниске који се простире преко њене средине.

Покажимо и да је $2|x| \geq |w|$. Важи $|x| = (n-1)|p| + |p_1|$, а из $n \geq 2$ следи $(n-1) \cdot |p| \geq |p|$, па је $|x| = (n-1) \cdot |p| + |p_1| \geq |p| + |p_1| = |t| + |p_1| \geq |t|$, те је $2 \cdot |x| \geq |x| + |t| = |w|$.

Докажимо супротну импликацију. Претпоставимо да ниска w има префикс-суфикс x дужине бар $|w|/2$ и да важи $w = xs = tx$. Покажимо да је ниска w периодична и да јој је t период.

Јасно је да су дужине ниски s и t једнаке и непразне (јер јер сваки префикс-суфикс x краћи од целе речи w).

Нека је $|w| = q|t| + r, 0 \leq r < |t|$. Пошто је $|t| \leq |w|$, важи да је $q > 0$. Нека је $w = w_1w_2 \ldots w_qz$, где је $|w_1| = |w_2| = \ldots = |w_q| = |t|$ и $|z| = r$. Из $w = tx$ следи $w_1 = t$ и $w_2 \ldots w_qz = x$. Зато је и $w = xs = w_2 \ldots w_qzs$, па важи редом $w_2 = w_1, w_3 = w_2, \ldots, w_{q-1} = w_q$ (слика 4.14).



Слика 4.14: Одређивање периоде када је познат префикс-суфикс који се простире преко средине ниске.

Из услова $zs = w_q z$ и $|z| = r < q = |w_q|$, следи да је z префикс ниске $w_q = t$. Одавде добијамо $w = t^q z$, при чему је z префикс непразне ниске t.

Да бисмо доказали да је ниска периодична, потребно је још да докажемо да је $q \geq 2$. Из услова $|x| \geq |w|/2$ следи да је $|s| = |t| \leq |x|$. Ако би важило $q < 2$, важило би $|w| \leq |t| + r < 2|t|$, што заједно са претпоставком $|w| \leq 2|x|$ даје $2|w| < 2|x| + 2|t| \leq 2|w|$ и добили бисмо контрадикцију. □

Пример 4.3.8

На пример, ако је ниска abacabacaba, тада је тражени префикс-суфикс x једнак abacaba, суфикс s једнак је caba, док је префикс t једнак abac. Важи да је $\pi_{|w|} = \pi_{11} = 7 \geq 11/2$.

Дакле, решење овог проблема се своди на проналажење дужине најдужег префикс-суфикса речи w (то је вредност $\pi_{|w|}$) и проверу да ли важи $2\pi_{|w|} \geq |w|$. То можемо урадити већ приказаном функцијом `kmpIzracunajPrefiksSufikse`.

```
bool periodicna(const string& w) {
  int m = w.size();
  // racunamo duzinu najduzeg prefiks-sufiksa niske w
  vector<int> pi = kmpIzracunajPrefiksSufikse(w);
  // proveravamo duzinu najduzeg prefiks-sufiksa
  return 2 * pi[m] >= m;
}
```

Сложеност овог алгоритма једнака је сложености алгоритма за предобраду у алгоритму KMP, односно једнака је $O(|w|)$.

Алгоритам KMP се може употребити и за одређивање најкраћег периода речи w тј. најкраћег префикса p речи w таквог да је $p = p_1 p_2$ и $w = p^n p_1$. Ако реч није периодична, могуће је да је $w = pp_1$ или да је $w = p$.

Лема 4.3.3 **[Најкраћа периода]**

Дужина најкраћег периода ниске w је $|w| - \pi_{|w|}$.

Доказ. Ако је $w = p^n p_1$ и $p = p_1 p_2$, тада је $w = p_1 p_2 p_1 p_2 \dots p_1 p_2 p_1$, па је $p^{n-1} p_1 = p_1 p_2 \dots p_1 p_2 p_1$ префикс-суфикс речи w. Важи и обратно. Ако је x префикс-суфикс, тада је $px = xq$, па се на исти начин као у леми 4.3.2 доказује да је $w = p^n p_1$ и $p = p_1 p_2$, за одговарајуће n. Дакле, постоји бијекција између периода ниске и њених префикс-суфикса.

Ако је p најкраћи префикс који задовољава услов периодичности, онда је $|p| \leq |p'|$ за све друге периоде. Ако периоду p одговара префикс-суфикс x, а периоду p' префикс-суфикс x', тада је $|x'| = |w| - |p'| \leq |w| - |p| = |x|$. Зато најкраћем периоду одговара најдужи префикс-суфикс ниске w. Дакле, дужина $\pi_{|w|}$ најдужег префикс-суфикса речи w одређује дужину $|w| - \pi_{|w|}$ њеног најкраћег периода. □

Да би реч била периодична, дужина њеног најкраћег периода мора да буде краћа од половине ниске, односно мора да важи $|w| - \pi_{|w|} \leq |w|/2$, тј. $2\pi_{|w|} \geq |w|$, тј. $\pi_{|w|} \geq |w|/2$, што је потпуно у складу са лемом 4.3.2.

На крају напоменимо да овакве карактеризације периодичности нису својствене само КМП низу π. На пример, лако се може доказати да се дужина најкраћег периода може одредити и помоћу z-низа као најмања вредност k таква да је $k + z_k \geq |w|$.

Задатак: Најкраћа допуна до палиндрома

Ниска $abaca$ није палиндром (не чита се исто слева и сдесна), али ако јој на почетак допишемо карактере ac, добијамо ниску $acabaca$ која јесте палиндром (чита се исто и слева и сдесна). Напиши програм који за дату ниску одређује дужину најкраћег палиндрома који се може добити дописивањем карактера с леве стране дате ниске.

Опис улаза

Са стандардног улаза се уноси ниска састављена од N ($1 \leq N \leq 50000$) малих слова енглеске абецеде.

Опис излаза

На стандардни излаз исписати тражену дужину најкраће проширене ниске која је палиндром.

Пример 1

Улаз	*Излаз*
`abacba`	`9`

Пример 2

Улаз	*Излаз*
`anavolimilovana`	`15`

Пример 3

Улаз	*Излаз*
`anavolimilovanakapak`	`25`

Решење

Претпоставимо да се најкраћи могући палиндром може добити дописивањем ниске p на ниску s тј. да је ps најкраћи палиндром коме је s суфикс. Тада је $ps = s'p'$, где је s' ниска која се добија обртањем карактера ниске s, док је p' ниска која се добија обртањем карактера ниске p. Дакле, s се завршава са p' и постоји неки њен префикс t такав да је $s = tp'$. Зато је $ptp' = pt'p'$, где је t' ниска која се добија обртањем карактера ниске t, па је $t = t'$, тј. t мора бити палиндром. Да би дужина префикса p била што мања, дужина палиндрома t мора бити што већа, па је t најдужи могући палиндром којим почиње ниска s. Дакле, проблем проналажења најкраће допуне p се своди на проблем одређивања најдужег палиндромског префикса t.

Пример 4.3.9

На пример, да бисмо одредили најдужи палиндром којим се допуњује ниска $s = abacba$, примећујемо да је њен најдужи палиндромски префикс ниска $t = aba$, одакле следи $p' = cba$, па је $p = abc$ и резултујући палиндром је $ps = abcabacba$.

Ако је $|t|$ дужина палиндрома t, а $|s|$ дужина ниске s, дужина дела p једнака је $|s| - |t|$, па је укупна дужина траженог палиндрома једнака $(|s| - |t|) + |s| = 2|s| - |t|$.

Палиндром t можемо одредити грубом силом, тако што редом проверавамо све префиксе ниске s и тражимо најдужи палиндром међу њима.

Задатак ефикасније можемо решити алгоритмом KMP. Ако уместо ниске s посматрамо ниску ss' добијену надовезивањем ниске добијене обртањем редоследа карактера ниске s на ниску s, најдужи палиндромски префикс ниске s је најдужи префикс ниске ss' који је уједно њен суфикс и који има највише n карактера, где је n дужина ниске s. Да би се спречило да се приликом попуњавања KMP-низа π (низа дужина префикс-суфикса) урачунају и они суфикси и префикси који садрже целу полазну ниску, довољно је да се између ниски s и s' постави неки карактер који оне не садрже (пошто се ниске састоје само од малих слова енглеске абецеде, можемо, на пример, употребити карактер #). Када изградимо ниску $s\#s'$, тада попуњавамо KMP-низ π и његов последњи елемент представља дужину најдужег палиндромског префикса. Сложеност овог алгоритма је $O(|s|)$.

```
// izracunava duzinu najduzeg prefiksa niske s koji je palindrom
int duzinaNajduzegPalindromskogPrefiksa(const string& s) {
  string sObratno(s.rbegin(), s.rend());
  string str = s + "#" + sObratno;
  vector<int> kmp = kmpIzracunajPrefiksSufikse(str);
```

```
  return kmp[kmp.size() - 1];
}

// odredjuje duzinu najkraćeg palindroma koji se može dobiti
// dopisivanje slova na pocetak niske s
int najkraciPalindrom(const string& s) {
  // s razlazemo na prefiks + sufiks tako da je prefiks sto duzi
  // palindrom. Tada je trazeni palindrom dobijen sa sto manje
  // dopisivanja slova na pocetak jednak:
  //    obrni(sufiks) + prefiks + sufiks
  int duzinaPrefiksa = duzinaNajduzegPalindromskogPrefiksa(s);
  int duzinaSufiksa = s.size() - duzinaPrefiksa;
  int duzinaNajkracegPalindroma =
          duzinaSufiksa + duzinaPrefiksa + duzinaSufiksa;
  return duzinaNajkracegPalindroma;
}
```

Задатак: Домине

Домине се слажу једна уз другу, тако што се поља на доминама постављеним једну уз другу морају поклапати. Домине обично имају само два поља, међутим, наше су домине специјалне и имају више различитих поља (означених словима). Ако све домине које слажемо имају исту ниску, написати програм који одређује како је задати број домина могуће сложити тако да заузму што мање простора по дужини (свака наредна домина мора бити смакнута бар за једно поље). На пример, ако на доминама пише `ababcabab`, најмање простора заузимају ако се сложе на следећи начин:

```
ababcabab
    ababcabab
        ababcabab
```

Опис улаза

Први ред стандардног улаза садржи ниску малих слова енглеске абецеде које представљају ниске на доминама. Дужина ниске је између 2 и 10^5 карактера. У наредном реду се налази цео број n $(1 \leq n \leq 5 \cdot 10^4)$ који представља број домина.

Опис излаза

На стандардни излаз исписати цео број који представља укупну дужину сложених домина.

Пример 1		**Пример 2**		**Пример 3**	
Улаз	*Излаз*	*Улаз*	*Излаз*	*Улаз*	*Излаз*
`ababcabab`	`19`	`abc`	`15`	`aa`	`11`
`3`		`5`		`10`	

Решење

Приликом слагања домина потребно је да се поклопе неки прави префикс и прави суфикс текста на доминама. Да би домине заузимале што мање простора, потребно је да преклапање буде што дуже. Дакле, потребно је пронаћи дужину k најдужег префикс-суфикса ниске s. Укупна дужина домина је тада $k + n \cdot (|s| - k)$. Наиме, након прве домине дужине $|s|$, свака наредна домина додаје још $|s| - k$ карактера.

```
vector<int> kmp = kmpIzracunajPrefiksSufikse(s);
cout << kmp[s.size()] + n * (s.size() - kmp[s.size()]) << endl;
```

4.4 Најдужи палиндромски сегмент - Маначеров алгоритам

Проблем

Дата је ниска s која садржи само мала слова. Одредити најдужи сегмент ниске s који је палиндром. Ако има више најдужих сегмената који су палиндроми, приказати сегмент чији почетак има најмањи индекс.

Пример 4.4.1

За ниске `ananas`, `najjaci` и `list` најдужи сегменти који су палиндроми су редом `anana`, `ajja` и `l`.

4.4.1 Провера свих сегмената

Проблем је могуће решити анализом свих сегмената, провером да ли је текући сегмент палиндром и одређивањем најдужег пронађеног палиндрома. Пошто је провера да ли је ниска дужине n палиндром сложености $O(n)$, а сегмената ниске дужине n укупно има $O(n^2)$, сложеност овог приступа је $O(n^3)$.

4.4.2 Провера свих сегмената редом према опадајућим дужинама

Имплементација се може мало поједноставити и дугачки палиндроми се могу пронаћи брже, ако се примети да сегменте можемо анализирати редом почев од најдужег сегмента па уназад до сегмента дужине 1. Приметимо да ће са овим редоследом обиласка

први сегмент за који се утврди да је палиндром управо бити најдужи палиндром који тражимо. Ако сегменте исте дужине разматрамо у растућем редоследу индекса леве границе, у случају када постоји више палиндрома исте највеће дужине, као први ће бити пронађен онај који има најмању вредност индекса леве границе, као што је и тражено. Сложеност најгорег случаја овог приступа је и овде $O(n^3)$.

4.4.3 Провера центара

Палиндроми поседују одређено својство инкременталности које нам може помоћи да пронађемо ефикаснији алгоритам. Наиме, ако је познат центар палиндрома (то може бити било неко слово, било позиција тачно између два суседна слова) и ако знамо да се k слова око тог центра сликају као у огледалу и на тај начин граде палиндром, онда за проверу да ли се $k+1$ слова око тог центра сликају као у огледалу не треба проверавати све из почетка, већ је довољно само проверити да ли су два слова на спољним позицијама ($(k+1)$-во слово лево тј. десно од центра) једнака. Зато ефикасније решење добијамо ако:

- за свако слово речи одредимо најдужи палиндром непарне дужине такав да му је изабрано слово центар и
- за сваку позицију између два суседна слова одредимо најдужи палиндром парне дужине коме је та позиција центар.

Да бисмо одредили најдужи палиндром са центром у слову s_i, ширимо палиндром s_i у десно и у лево за k слова док се налазимо унутар речи ($i-k \geq 0$ и $i+k < n$) и док су одговарајућа слова једнака ($s_{i-k} = s_{i+k}$). У тренутку када се то први пут наруши, пронађен је најдужи палиндром са центром у слову s_i (ако се изађе из речи даље проширивање није могуће, а ако се пронађе различит пар слова даља проширивања не могу више да дају палиндром).

Одређивање најдужег палиндрома са центром између слова s_i и s_{i+1} вршимо на аналоган начин: док смо унутар речи ($i-k \geq 0$ и $i+k+1 < n$) ширимо палиндром све док важи $s_{i-k} = s_{i+k+1}$.

За сваку позицију која може бити центар палиндрома (а њих има n за слова ниске и $n-1$ за позиције између два слова ниске, тј. укупно $2n-1$, односно $O(n)$) налазимо најдужи палиндром са центром у њој ширећи текући палиндром налево и надесно. Глобално најдужи палиндром налазимо као најдужи од добијених палиндрома.

Ширење за фиксирани центар палиндрома се обавља у једном проласку и захтева време $O(n)$, па је укупна сложеност овог алгоритма $O(n^2)$.

```
string najduzaPalindromskaPodniska(const string& s) {
  // duzina reci
  int n = s.size();
  // duzina i pocetak najduzeg palindroma
  int maxDuzina = 0, maxPocetak;

  // prolazimo kroz sva slova reci
  for (int i = 0; i < n; i++) {
    int duzina, pocetak;

    // nalazenje najduzeg palindroma neparne duzine ciji je centar
    // slovo s[i]
    int k = 1;
    while (i - k >= 0 && i + k < n && s[i - k] == s[i + k])
      k++;
    // duzina i pocetak maksimalnog palindroma
    duzina = 2 * k - 1;
    pocetak = i - k + 1;
    // azuriramo maksimum ako je to potrebno
    if (duzina > maxDuzina) {
      maxDuzina = duzina;
      maxPocetak = pocetak;
    }

    // nalazenje najduzeg palindroma parne duzine ciji je centar
    // izmedju slova s[i] i s[i+1]
    k = 0;
    while (i - k >= 0 && i + k + 1 < n && s[i - k] == s[i + k + 1])
      k++;
    // duzina i pocetak maksimalnog palindroma
    duzina = 2 * k;
    pocetak = i - k + 1;
    // azuriramo maksimum ako je to potrebno
    if (duzina > maxDuzina) {
      maxDuzina = duzina;
      maxPocetak = pocetak;
    }
  }
```

```
  // izdvajamo i ispisujemo odgovarajuci palindrom
  return s.substr(maxPocetak, maxDuzina);
}
```

4.4.4 Експлицитна допуна речи и позиција

Постоји неколико техника које могу да упросте имплементацију претходног алгоритма, обједињавајући случајеве палиндрома парне и непарне дужине. Замислимо да се пре првог, након последњег и између свака два суседна слова речи постави специјални карактер `|`. На пример, реч `aabcbab` допуњавамо до речи `|a|a|b|c|b|a|b|`. На тај начин постиже се да су сви центри палиндрома карактери проширене речи и довољно је анализирати само палиндроме непарне дужине у проширеној речи. Ово допуњавање полазне речи је могуће физички реализовати тако што се у програму експлицитно креира допуњена ниска. Ово упрошћава имплементацију по цену нешто споријег извршавања (додуше не асимптотски) и додатног заузећа меморије.

Напоменимо још и да се и провера припадности индекса тј. позиција опсегу речи може елиминисати ако се полазна реч прошири додатним специјалним карактерима на почетку и на крају: ти карактери морају бити међусобно различити и различити од осталих карактера у ниски. У ту сврху могу да се искористе `^` и `$` (ти карактери користе за означавање почетка и краја у регуларним изразима).

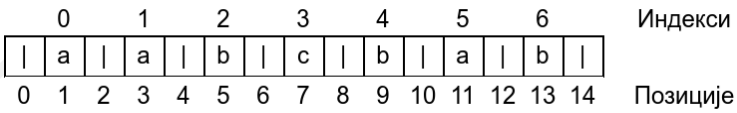
Дужину палиндрома разматраћемо у односу на полазну (а не допуњену) реч.

```
// da bismo uniformno posmatrali palindrome i parne i neparne duzine,
// prosirujemo nisku dodajuci ^ i $ oko njega i umecuci | izmedju
// svih slova, na primer abc -> ^|a|b|c|$
string dopuni(const string &s) {
  string rez = "^";
  for (int i = 0; i < s.size(); i++)
    rez += "|" + s.substr(i, 1);
  rez += "|$";
  return rez;
}

string najduzaPalindromskaPodniska(const string& s) {
  string t = dopuni(s);

  // dovoljno je pronaci najveci palindrom neparne duzine u prosirenoj reci
  int maxDuzina = 0, maxCentar;
  // proveravamo sve pozicije u dopunjenoj reci
```

```
  for (int i = 1; i < t.size() - 1; i++) {
    // prosirujemo palindrom sa centrom na poziciji i dokle god je to
    // moguce
    int d = 0;
    while (t[i - d - 1] == t[i + d + 1])
      d++;

    // azuriramo maksimum ako je potrebno
    if (d > maxDuzina) {
      maxDuzina = d;
      maxCentar = i;
    }
  }

  // ispisujemo konacan rezultat, odredjujuci pocetak najduzeg palindroma
  int maxPocetak = (maxCentar - maxDuzina) / 2;
  return s.substr(maxPocetak, maxDuzina);
}
```

4.4.5 Имплицитна допуна речи и позиција

Да бисмо олакшали наредно излагање, индексе у допуњеној речи (без додатих ознака почетка и краја речи) називаћемо позиције, а у оригиналној речи само индекси. На слици 4.15 приказани су индекси и позиције за реч `aabcbab`.



Слика 4.15: Индекси и позиције за реч `aabcbab`.

Приметимо неколико чињеница. Ако је полазна реч дужине n, укупно имамо $N = 2n+1$ позиција у допуњеној речи. Слова полазне речи се налазе на непарним позицијама, док се на парним позицијама налази специјални карактер `|`. Слово са индексом k се налази на позицији $p = 2k + 1$, што значи да се на непарној позицији p налази слово полазне речи са индексом $k = \lfloor \frac{p}{2} \rfloor$.

За сваку позицију i, $0 \leq i < N$, дужину палиндрома са центром на тој позицији можемо одредити на исти начин, без обзира на то да ли је на тој позицији слово или специјални карактер `|`. Разматраћемо дужину палиндрома у полазној речи (а не допуњеној) и она је једнака броју карактера са леве стране дате позиције у допуњеној речи који су једнаки

одговарајућим карактерима са десне стране те позиције у допуњеној речи. На пример, у претходном примеру за позицију 7 то је 5, јер је палиндром `abcba` дужине 5, што одговара томе да пет карактера `|a|b|` са леве стране карактера `c` у допуњеној речи одговарају карактерима `|b|a|` са десне стране карактера `c`. Слично важи и за парне позиције (на пример 2).

На почетку, дужину палиндрома d постављамо на 1, ако је позиција i непарна, тј. 0 ако је парна. Заиста, ако је позиција непарна, на њој се налази слово полазне речи које је само за себе палиндром дужине 1 (из другог угла гледано, лево и десно од ове позиције се налазе карактери `|`, па је бар један карактер једнак). Ако је позиција парна, око ње се налазе два слова (осим у случају крајњих позиција) и не знамо унапред да ли су она једнака, тако да дужину палиндрома иницијално постављамо на 0. На овај начин постижемо да су бројеви $i-d$ и $i+d$ парни, што значи да су бројеви $i-d-1$ и $i+d+1$ непарни и ако су у опсегу $[0, N)$, они указују на наредна два карактера полазне ниске чију једнакост треба проверити. Ако су карактери полазне речи на одговарајућим индексима једнаки (то су индекси $\lfloor\frac{i-d-1}{2}\rfloor$ и $\lfloor\frac{i+d+1}{2}\rfloor$), онда се d увећава за 2 (из једног угла гледано, та два једнака карактера се додају текућем палиндрому, па му се дужина повећава за 2, а из другог угла гледано, испред првог и иза другог се налазе специјални знаци `|` који су сигурно једнаки и њихову једнакост није потребно експлицитно проверавати). На тај начин се одржава и инваријанта да су бројеви $i+d$ и $i-d$ парни и петља се може наставити на исти начин све док се не наиђе на два различита слова или се изађе ван опсега допуњене речи.

```
string najduzaPalindromskaPodniska(const string& s) {
  // broj pozicija u reci s
  int N = 2 * s.size() + 1;
  int maxDuzina = 0, maxCentar;
  // za svaki moguci centar palindroma
  for (int i = 0; i < N; i++) {
    // d postavljamo na 0 za parnu, a na 1 za neparnu poziciju
    int d = 0;
    if (i % 2 == 1)
      d = 1;

    // dok god su pozicije u opsegu dozvoljenih indeksa i slova
    // na odgovarajucim indeksima jednaka uvecavamo d za 2
    while (i - d - 1 >= 0 && i + d + 1 < N &&
       s[(i - d - 1) / 2] == s[(i + d + 1) / 2])
      // ukljucujemo dva slova u palindrom, pa se duzina uvecava za 2
      d += 2;
```

```
    if (d > maxDuzina) {
      maxDuzina = d;
      maxCentar = i;
    }
  }

  // ispisujemo konacan rezultat, odredjujuci pocetak najduzeg palindroma
  int maxPocetak = (maxCentar - maxDuzina) / 2;
  return s.substr(maxPocetak, maxDuzina);
}
```

4.4.6 Маначеров алгоритам

Илуструјмо основну идеју Маначеровог алгоритма кроз један пример.

Пример 4.4.2

Посматрајмо како изгледа дужина најдужег палиндрома са центром у свакој од позиција p у речи `s = babcbabcbaccba` *(слика 4.16). Обележимо ову дужину са d_p, а допуњену реч са t.*

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
	|	b	|	a	|	b	|	c	|	b	|	a	|	b	|	c	|	b	|	a	|	c	|	c	|	b	|	a	|
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
	0	1	0	3	0	1	0	7	0	1	0	9	0	1	0	5	0	1	0	1	0	1	2	1	0	1	0	1	0

Слика 4.16: У првом реду дати су индекси i у полазној речи s, у другом проширена реч t, у трећем дате су позиције p у проширеној речи, а у последњем вредност d_p.

Посматрајмо, на пример, најдужи палиндром са центром у позицији 11 *– његова дужина је* $d_{11} = 9$ *и он се простире од позиције* 2 *до позиције* 20.

Посматрајмо сада дужину најдужег палиндрома са центром у позицији 12. *Пошто је претходно одређени палиндром симетричан око позиције* 11, *позицији* 12, *одговара позиција* 10. *Знамо да је* $d_{10} = 0$. *То је зато што је* $t_9 \neq t_{11}$. *Међутим, ми знамо да је* $t_9 = t_{13}$ *(пошто су обе позиције унутар нашег палиндрома), па је зато* $t_{11} \neq t_{13}$ *и важи* $d_{12} = d_{10} = 0$. *Приметимо да ово можемо констатовати без икакве потребе за новим упоређивањем карактера.*

Посматрајмо сада дужину најдужег палиндрома са центром у позицији 13. *Њему одговара палиндром са центром на позицији* 9. *Важи* $d_9 = 1$, *јер је* $t_8 = t_{10}$ *и* $t_7 \neq t_{11}$.

На основу симетрије палиндрома са центром у 11*, важи* $t_8 = t_{14}$*,* $t_{10} = t_{12}$*,* $t_7 = t_{15}$ *и* $t_{11} = t_{11}$*. Зато је* $t_{14} = t_{12}$ *и* $t_{15} \neq t_{11}$*, па је* $d_{13} = d_9 = 1$*.*

Наизглед, важи да је за свако i *унутар ширег палиндрома са центром у некој позицији* C *број* d_i *једнак броју* $d_{i'}$*, где се* i' *одређује као симетрична позиција позицији* i *у односу на позицију* C *(важи да је растојање од* C *до* i *и* i' *једнако, па је* $C - i' = i - C$*, тј.* $i' = C - (i - C)$*). Но, то није увек тачно.*

Посматрајмо сада вредност d_{15} *и њој одговарајућу вредност* d_7*. Оне нису једнаке. Зашто? Посматрајмо шта је оно што можемо закључити из симетрије палиндрома са центром на позицији* 11*. Важи да је* t_2 *до* t_{12} *једнако одговарајућим карактерима* t_{20} *до* t_{10} *– то је гарантовано симетријом и није потребно проверавати. Међутим, важи* $d_7 = 7$*. Знамо зато и да је* $t_1 = t_{13}$*, међутим, не можемо да тврдимо да симетрично у односу на позицију* 11 *важи* $t_{21} = t_9$*, зато што* t_{21} *није више део палиндрома са центром на позицији* C*. Дакле, проверу да ли важи* $t_{21} = t_9$ *је потребно посебно извршити.*

Размотримо сада општи случај. Претпоставимо да је $[L, R]$ палиндром са центром на позицији C (тада је $C - L = R - C$), да је i неки индекс унутар тог палиндрома (нека је $C < i < R$) и да је $i' = C - (i - C)$ њему симетричан индекс. Ако је палиндром са центром у i' у потпуности садржан у палиндрому (L, R) (без урачунатих крајева), тј. ако је $L < i' - d_{i'}$, тј. $d_{i'} < i' - L = (C - (i - C)) - (C - (R - C)) = R - i$, тада је $d_i = d_{i'}$. Докажимо ово.

За сваку вредност $0 \leq j \leq d_{i'}$ треба доказати да је $t_{i-j} = t_{i+j}$. Заиста, пошто важи $L < i' - j$ и $i + j < R$, закључује се да је $t_{i-j} = t_{i'+j}$ и $t_{i+j} = t_{i'-j}$. Међутим, пошто је i' центар палиндрома дужине $d_{i'}$, важи $t_{i'-j} = t_{i'+j}$. Зато се на позицији i налази центар палиндрома дужине бар $d_{i'}$. Докажимо да је ово и горње ограничење, тј. докажимо да је $t_{i-d_{i'}-1} \neq t_{i+d_{i'}+1}$. Пошто је $i + d_{i'} < R$, важи $i + d_{i'} + 1 \leq R$, па је $t_{i-d_{i'}-1} = t_{i'+d_{i'}+1}$ и $t_{i+d_{i'}+1} = t_{i'-d_{i'}-1}$. Међутим, пошто је палиндром са центром у i' дужине $d_{i'}$, важи $t_{i'-d_{i'}-1} \neq t_{i'+d_{i'}+1}$.

Ако је $[L, R]$ палиндром са центром на позицији C и ако је i неки индекс унутар тог палиндрома ($C < i < R$), али такав да је $d_{i'} \geq R - i$, онда можемо само да закључимо да је $d_i \geq R - i$.

Ово инспирише наредни алгоритам, познат под називом *Маначеров*[7] *алгоритам*. Слично као у претходној верзији алгоритма обрађујемо све позиције i од 0 до $N - 1$. При том одржавамо индексе C и R такве да је $[C - (R - C), R]$ најдеснији до сад пронађени палиндром. Ако је $i \geq R$, тада палиндром са центром у i одређујемо из

[7] Глен Маначер (енгл. Glenn Manacher), рођен 1934. године, амерички информатичар.

почетка, повећавајући за два дужину палиндрома d_i која креће од 0 или 1 (у зависности од парности позиције i), све док је то могуће, исто као у претходно описаном алгоритму. Међутим, ако је $i < R$, тада одређујемо вредност $i' = C - (i - C)$ и ако важи $d_{i'} < R - i$, постављамо одмах $d_i = d_{i'}$. Ако је $d_{i'} \geq R - i$, тада дужину d_i постављамо на почетну вредност $R - i$ тј. на $R - i + 1$ тако да су $i - d_i$ и $i + d_i$ парни бројеви, и онда је постепено повећавамо за 2, све док је то могуће (опет, веома слично као у претходно описаном алгоритму). Ако је пронађени палиндром са центром у i такав да му десни крај десно од позиције R, онда њега проглашавамо за нови палиндром $[L, R]$, постављајући му центар $C = i$ и десни крај $R = i + d_i$. На почетку можемо иницијализовати $R = C = 0$ (на тај начин обезбеђујемо да не може да важи $i < R$ и да се на почетку неће користити симетричност окружујућег палиндрома).

```
string najduzaPalindromskaPodniska(const string& s) {
  // broj pozicija u reci s (pozicije su ili slova originalnih reci,
  // ili su izmedju njih)
  int N = 2 * s.size() + 1;
  // d[i] je duzina najduzeg palindroma ciji je centar na poziciji i
  vector<int> d(N);

  // znamo da je [L, R] palindrom sa centrom u C
  int C = 0, R = 0; // L = C - (R - C)
  for (int i = 0; i < N; i++) {
    // karakter simetrican karakteru i u odnosu na centar C
    int i_sim = C - (i - C);
    if (i < R && i + d[i_sim] < R)
      // nalazimo se unutar palindroma [L, R], ciji je centar C
      // palindrom sa centrom u i_sim i palindrom sa centrom u i su
      // celokupno smesteni u palindrom (L, R)
      d[i] = d[i_sim];
    else {
      // ili se ne nalazimo u okviru nekog prethodnog palindroma ili
      // se nalazimo unutar palindroma [L, R], ali je palindrom sa
      // centrom u i_sim takav da nije celokupno smesten u (L, R);
      // u tom slucajmo znamo da je duzina palindroma sa centrom u i bar
      // bar R - i, a da li je vise od toga, treba proveriti
      d[i] = i <= R ? R - i : 0;

      // osiguravamo da je i + d[i] stalno paran broj
```

```
      if ((i + d[i]) % 2 == 1)
        d[i]++;

      // dok god su pozicije u dozvoljenom opsegu i slova na odgovarajucim
      // indeksima jednaka uvecavamo d[i] za 2 (jedno slovo s leva i
      // jedno slovo zdesna)
      while (i - d[i] - 1 >= 0 && i + d[i] + 1 < N &&
         s[(i - d[i] - 1) / 2] == s[(i + d[i] + 1) / 2])
        // ukljucujemo dva slova u palindrom, pa se duzina uvecava za 2
        d[i] += 2;
    }

    // ako palindrom sa centrom u i prosiruje desnu granicu
    // onda njega uzimamo za palindrom [L, R] sa centrom u C
    if (i + d[i] > R) {
      C = i;
      R = i + d[i];
    }
  }

  // pronalazimo najvecu duzinu palindroma i pamtimo njegov centar
  int maxDuzina = 0, maxCentar;
  for (int i = 0; i < N; i++) {
    if (d[i] > maxDuzina) {
      maxDuzina = d[i];
      maxCentar = i;
    }
  }

  // ispisujemo konacan rezultat, odredjujuci pocetak najduzeg palindroma
  int maxPocetak = (maxCentar - maxDuzina) / 2;
  return s.substr(maxPocetak, maxDuzina);
}
```

Сложеност овог алгоритма је линеарна. Интуитивно, проналажење кратких палиндрома захтева мали број извршавања унутрашње петље, док проналажење једног дугачког палиндрома захтева дуже извршавање унутрашње петље, али доводи до тога да ће се у наредним корацима у великом броју случајева у потпуности избегавати њено извршавање. Прецизније, свако извршавање унутрашње `while` петље повећава вредност промењиве R (јер је у случају `else` гране $i + d[i] \geq R$, а свако успешно

извршавање `while` петље увећава вредност $d[i]$ за 2, те ће важити $i + d[i] > R$) која се нигде не смањује, а чија је максимална вредност N, те је укупан број извршавања унутрашње петље ограничен са $O(N)$.

Можемо имплементирати и варијанту која елиминише проверу припадности индекса опсегу речи на тај начин што експлицитно прави допуњену реч (њена имплементација је једноставнија).

```
string najduzaPalindromskaPodniska(const string& s) {
  // jednostavnosti radi dopunjavamo rec s
  string t = dopuni(s);
  // d[i] je najveci broj takav da je [i - d[i], i + d[i]] palindrom
  // to je ujedno i duzina najduzeg palindroma ciji je centar na
  // poziciji i (pozicije su ili slova originalnih reci, ili su
  // izmedju njih)
  vector<int> d(t.size());
  // znamo da je [L, R] palindrom sa centrom na poziciji C
  int C = 0, R = 0; // L = C - (R - C)
  for (int i = 1; i < t.size() - 1; i++) {
    // karakter simetrican karakteru i u odnosu na centar C
    int i_sim = C - (i - C);
    if (i < R && i + d[i_sim] < R)
      // nalazimo se unutar palindroma [L, R], ciji je centar C
      // palindrom sa centrom u i_sim i palindrom sa centrom u i su
      // celokupno smesteni u palindrom (L, R)
      d[i] = d[i_sim];
    else {
      // ili se ne nalazimo u okviru nekog prethodnog palindroma ili
      // se nalazimo unutar palindroma [L, R], ali je palindrom sa
      // centrom u i_sim takav da nije celokupno smesten u (L, R);
      // u tom slucajmo znamo da je duzina palindroma sa centrom u i bar
      // R - i, a da li je vise od toga, treba proveriti
      d[i] = i <= R ? R-i : 0;
      // prosirujemo palindrom dok god je to moguce krajnji karakteri
      // ^$ obezbedjuju da nije potrebno proveravati granice
      while (t[i - d[i] - 1] == t[i + d[i] + 1])
        d[i]++;
    }

    // ako palindrom sa centrom u i prosiruje desnu granicu
```

```
    // onda njega uzimamo za palindrom [L, R] sa centrom u C
    if (i + d[i] > R) {
      C = i;
      R = i + d[i];
    }
  }

  // pronalazimo najvecu duzinu palindroma i pamtimo njegov centar
  int maxDuzina = 0, maxCentar;
  for (int i = 1; i < t.size() - 1; i++)
    if (d[i] > maxDuzina) {
      maxDuzina = d[i];
      maxCentar = i;
    }

  // ispisujemo konacan rezultat, odredjujuci pocetak najduzeg palindroma
  return s.substr((maxCentar - maxDuzina) / 2, maxDuzina);
}
```

5. Геометријски алгоритми

Геометријски алгоритми играју важну улогу у многим областима, попут рачунарске графике, пројектовања помоћу рачунара, пројектовања интегрисаних кола високе резолуције (VLSI), роботике и база података. У рачунарски генерисаној слици може бити на хиљаде или чак на милионе тачака, линија, троуглова, квадрата или кругова; слично, пројектовање рачунарског чипа може да захтева рад са милионима елемената који се представљају геометријским фигурама. Све ове примене захтевају обраду геометријских објеката. Пошто величина улаза за ове проблеме може бити врло велика, веома је значајно развити што ефикасније алгоритме за њихово решавање.

Честа неугодна карактеристика геометријских проблема је постојање многих специјалних случајева. На пример, две праве у равни се секу у једној тачки, сем ако су паралелне или се поклапају. При решавању проблема са две праве, морају се предвидети сва три могућа случаја. Сложеније фигуре проузрокују појаву много већег броја специјалних случајева о којима треба водити рачуна. Обично се већина тих специјалних случајева директно решава, али потреба да се они узму у обзир чини понекад конструкцију алгоритма врло исцрпљујућом. У наставку текста су игнорисани неки детаљи који нису од суштинског значаја за разумевање основних идеја алгоритама, а читаоцу саветујемо да размотри решавање свих специјалних случајева на који се при конструкцији алгоритма наиђе, да би се добили алгоритми који су у потпуности исправни.

- У првом поглављу ћемо обновити основне појмове аналитичке геометрије равни: **тачке**, **векторе**, **координате**, **операције над векторима**, са посебним нагласком на примене **скаларног** и **векторског производа**, рачунање **површине троугла** и **оријентације тројке тачака**.

- Приказаћемо алгоритам одређивања свих пресечних тачака датог скупа хоризонталних и вертикалних дужи.
- Посебан нагласак је стављен на многоуглове. Најпре ће бити размотрен алгоритам за конструкцију **простог многоугла**. Након тога, биће приказан алгоритам за испитивање **припадности тачке простом многоуглу** и биће описан ефикаснији алгоритам за испитивање **припадности тачке конвексном многоуглу**. На крају ће бити дискутовани разни алгоритми за конструкцију **конвексног омотача**: **индуктивни алгоритам** заснован на додавању једне по једне тачке, алгоритам **увијања поклона**, **Грејемов алгоритам** и **брзи алгоритам** за одређивање конвексног омотача (енгл. QuickHull).

5.1 Основе геометријских алгоритама

Приликом рачунарске обраде сви геометријски објекти се представљају аналитички: бројевима (координатама, коефицијентима и слично). Претпоставља се да је читалац упознат са основама аналитичке геометрије. Ипак, у наставку ћемо резимирати неке основне појмове корисне приликом конструкције геометријских алгоритама, при чему ћемо већину операција засновати на векторима и операцијама са њима (алтернативно, у традиционалној аналитичкој геометрији се геометријски објекти обично представљају имплицитним једначинама). Бавићемо се питањима попут израчунавања растојања између две тачке, испитивања колинеарности три тачке или израчунавања пресечне тачке двеју дужи. Све ове операције могу се извести алгоритмима константне временске сложености, коришћењем основних аритметичких операција. Претпостављамо и да се различите елементарне функције (квадратни корен, тригонометријске функције и слично) могу израчунати за константно време. Међутим, пошто је израчунавање тих функција спорије од елементарних рачунских операција, треба их избегавати када год је то могуће.

5.1.1 Тачке, координате

Основни геометријски објекти су *тачке* и већина других објеката се представља помоћу тачака. *Права* је најчешће представљена паром различитих тачака P и Q које јој припадају. *Дуж* се најчешће задаје паром тачака P и Q које представљају крајеве те дужи, и означаваћемо је са PQ.

5.1.1.1 Декартове координате

Тачке се у рачунару најчешће представљају својим *Декартовим координатама*. Тачке у равни се представљају паром Декартових координата (x, y), а тачке у тродимензионом простору тројком Декартових координата (x, y, z). У зависности од примене, координате су најчешће или целобројне или реалне (у рачунару најчешће представљене

бројевима у покретном зарезу).

Тачке се могу представити или структурним типом или уграђеним типом за представљање уређених n-торки (тј. парова ако су тачке у равни). На пример, тачке у равни са целобројним координатама могу се представити на било који од следећа два начина.

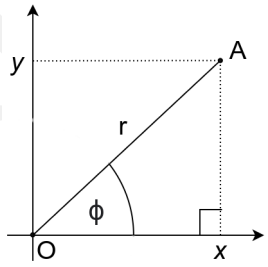
```
struct Tacka {
   int x, y;
};
```

```
typedef pair<int, int> Tacka;
```

Наравно, уместо типа `int` могу се користити и други типови за представљање координата (нпр. `long long`, `float`, `double` итд.).

5.1.1.2 Поларне координате

У неким алгоритмима је погоднија репрезентација тачака помоћу њихових *поларних координата*. Због тога је за дате Декартове координате x и y тачке A понекад потребно одредити њене поларне координате: растојање r од координатног почетка O и угао ϕ који дуж OA захвата са позитивним делом x осе и, обратно, на основу поларних координата одредити Декартове. Угао се често разматра као оријентисани угао из интервала $(-\pi, \pi]$ или $[0, 2\pi)$, где угао 0 представља позитиван смер x осе и угао расте у позитивном математичком смеру.



Слика 5.1: Веза између Декартових и поларних координата тачке.

Да бисмо трансформисали поларне координате у Декартове, можемо искористити следећу везу (слика 5.1):

$$\begin{aligned} x &= r\cos\phi \\ y &= r\sin\phi \end{aligned}$$

За трансформацију Декартових координата у поларне у већини случајева можемо искористити наредне једначине:

$$\begin{aligned} r &= \sqrt{x^2+y^2} \\ \phi &= \arctan\frac{y}{x} \end{aligned}$$

Претходне формуле нису увек исправне, јер вредност $\arctan\frac{y}{x}$ није дефинисана за $x = 0$, тј. за тачке на y-оси. Такође, вредност функције arctan је увек у интервалу $(-\pi/2, \pi/2]$, а не у интервалу $(-\pi, \pi]$ (или $[0, 2\pi)$), како бисмо желели. Због тога се уместо функције arctan дефинише и користи функција arctan2, која има два аргумента: y и x координату тачке чији се поларни угао израчунава као $\phi = \text{arctan2}(y, x)$ (обратити пажњу на редослед аргумената). Ова функција је директно подржана у већини програмских језика (под називом `atan2`) и враћа угао из интервала $(-\pi, \pi]$.

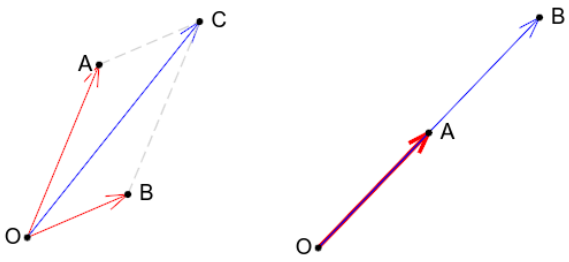
$$\text{arctan2}(y,x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & \text{за } x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & \text{за } x < 0 \text{ и } y \geq 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & \text{за } x < 0 \text{ и } y < 0 \\ +\frac{\pi}{2} & \text{за } x = 0 \text{ и } y > 0 \\ -\frac{\pi}{2} & \text{за } x = 0 \text{ и } y < 0 \\ \text{недефинисано} & \text{за } x = 0 \text{ и } y = 0 \end{cases}$$

5.1.2 Вектори

Поред тачака, основни појам аналитичке геометрије представљају *вектори*. Вектори се представљају уређеним паровима тачака тј. усмереним дужима. Усмерена дуж (A, B), која има почетну тачку A и крајњу тачку B одређује вектор $\overrightarrow{AB}$. Све друге усмерене дужи тј. парови тачака (A', B') таквих да постоји транслација којом се пар (A, B) пресликава на пар (A', B') одређују исти вектор (вектори су, дакле, класе еквиваленције усмерених дужи). Ако су почетна и крајња тачка вектора једнаке, оне одређују тзв. *нула-вектор* (говорићемо и да је у питању вектор једнак нули).

Вектори се могу сабирати и скалирати (множити бројем, тј. скаларом). Израз $\vec{a} + \vec{b}$ означава збир вектора $\vec{a}$ и $\vec{b}$, док израз $k \cdot \vec{a}$ означава скалирање вектора $\vec{a}$ скаларом k. Вектори се могу и одузимати. Разлика вектора $\vec{a}$ и $\vec{b}$ се дефинише као $\vec{a} - \vec{b} = \vec{a} + (-1) \cdot \vec{b}$. Сабирање и скалирање имају своју геометријску интерпретацију (слика 5.2).

Вектори се најчешће представљају аналитички, својим Декартовим координатама (паром координата за векторе у равни и тројком координата за векторе у простору). Сваки



Слика 5.2: Вектори се сабирају на основу правила паралелограма (збир је дијагонала паралелограма који образују вектори сабирци). На слици лево вектор $\overrightarrow{OC}$ је збир вектора $\overrightarrow{OA}$ и $\overrightarrow{OB}$. Скалирање вектора подразумева његово издуживање или скраћивање (а ако је скалар негативан, тада и промену смера). На слици десно вектор $\overrightarrow{OB}$ је $2 \cdot \overrightarrow{OA}$.

вектор у равни се може једнозначно изразити као линеарна комбинација произвољна два неколинеарна јединична вектора $\vec{i}$ и $\vec{j}$, тј. за сваки вектор $\vec{a}$ постоје бројеви x и y такви да је $\vec{a} = x \cdot \vec{i} + y \cdot \vec{j}$. Бројеви (x, y) представљају *координате* вектора $\vec{a}$. Декартов координатни систем је одређен међусобно нормалним јединичним векторима $\vec{i}$ и $\vec{j}$ и координатним почетком O и у наставку ћемо увек подразумевати да радимо са оваквим координатним системом.

Сабирање и скалирање вектора се лако изражавају аналитички. Збир вектора $\vec{a} = (a_x, a_y)$ и $\vec{b} = (b_x, b_y)$ је вектор $\vec{a} + \vec{b} = (a_x + b_x, a_y + b_y)$. Производ вектора $\vec{a} = (a_x, a_y)$ и скалара k је вектор $k \cdot \vec{a} = (ka_x, ka_y)$. Интензитет вектора $\vec{a}$ чије су координате (x, y) се израчунава Питагорином теоремом $|\vec{a}| = \sqrt{x^2 + y^2}$.

Вектор у равни се у језику C++ може представити следећом структуром.

```
struct Vektor {
  double x, y;
};
```

Координате било које тачке A су заправо координате вектора $\overrightarrow{OA}$.

- *Сабирањем тачке и вектора тачка се транслира за дати вектор.* Ако су познате координате тачке A и вектора $\overrightarrow{AB}$, њиховим сабирањем добијају се координате тачке B. Заиста, важи $\overrightarrow{OB} = \overrightarrow{OA} + \overrightarrow{AB}$. Ово ћемо краће записивати као: $B = A + \overrightarrow{AB}$.
- *Одузимањем тачака добија се вектор који их спаја.* Ако су познате координате почетне и крајње тачке вектора $\overrightarrow{AB}$, тада се координате вектора $\overrightarrow{AB}$ могу израчунати одузимањем координата тачке A од координата тачке B. Заиста, важи

$\overrightarrow{AB} = \overrightarrow{OB} - \overrightarrow{OA}$. Ово ћемо краће записивати као: $\overrightarrow{AB} = B - A$.

Задатак: Стрелица

У апликацији за цртање треба имплементирати функционалност исцртавања стрелице. Стрелица има своје тело (дуж AB) и врх (који се црта помоћу две краће дужи BM и BN), који је симетрично постављен у односу на тело. Ако су познате координате крајњих тачака тела стрелице AB, дужина дужи $|BM| = |BN| = d$ којима се црта врх и конвексан угао α који заклапају дужи на врху стрелице, одредити координате крајњих тачака дужи којима се врх представља.

Опис улаза

У првој линији стандардног улаза се налазе координате тачке A (два реална броја), у другој линији координате тачке B (различите од тачке A), у трећој дужина d дужи на врху стрелице (позитиван реалан број) и у четвртој конвексан угао α између дужи на врху стрелице (у степенима).

Опис излаза

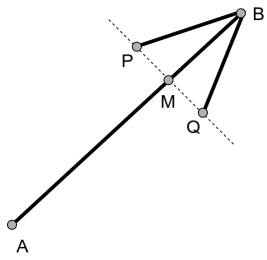
На стандардни излаз исписати координате крајњих тачака M и N дужи при врху стрелице (различите од тачке B).

Пример

Улаз	*Излаз*
`0 0`	`4 3`
`4 4`	`3 4`
`1`	
`90`	

Решење

Нека су тачке A и B крајње тачке тела стрелице, а P и Q крајње тачке дужи на врху стрелице (како је приказано на слици.



Нека је тачка M пресечна тачка дужи AB и PQ. Координате тачака P и Q можемо одредити као $P = M + \overrightarrow{MP}$ и $Q = M + \overrightarrow{MQ}$. Координате тачке M можемо одредити

као $M = B + \overrightarrow{BM}$.

Правац вектора $\overrightarrow{BM}$ је исти као правац вектора $\overrightarrow{BA} = A - B$. Зато је $\overrightarrow{BM} = |BM| \cdot \frac{\overrightarrow{BA}}{|BA|}$.

Интензитет $|BM|$ можемо одредити коришћењем елементарне тригонометрије. Важи да је угао $\angle PBQ = \alpha$, па је угао $\angle PBM = \alpha/2$. Троугао PMB је правоугли, па важи да је $\sin(\alpha/2) = |MP|/|PB|$ и $\cos(\alpha/2) = |BM|/|PB|$. Дужина $|PB|$ је позната и једнака је d, па је $|MP| = |MQ| = d\sin(\alpha/2)$ и $|BM| = d\cos(\alpha/2)$.

Остаје још да одредимо векторе $\overrightarrow{MP}$ и $\overrightarrow{MQ}$. Њихове интенизитете знамо (на основу претходног тригонометријског извођења) и остаје да им одредимо правац. Пошто су они управни на вектор $\overrightarrow{BA}$, њихов правац се може добити ротирањем вектора $\overrightarrow{BA}$ за 90 степени у једном или другом смеру. У општем случају важи да ако су координате вектора (x, y), ротирањем за 90 степени добијају се вектори чије су координате $(-y, x)$ и $(y, -x)$.

```
// vektor BA
Vektor BA = {A.x - B.x, A.y - B.y};
// duzina vektora BA
double BAd = sqrt(BA.x*BA.x + BA.y*BA.y);
// ugao alpha/2 u radijanima
double alpha2 = (alpha/2) / 180 * M_PI;
// duzina vektora BM
double BMd = d * cos(alpha2);
// duzina vektora MP i MQ
double MPd = d * cos(alpha2), MQd = MPd;
// vektor BM
Vektor BM = {BMd * BA.x / BAd, BMd * BA.y / BAd};
// tacka M
Tacka M = {B.x + BM.x, B.y + BM.y};
// vektor MP
Vektor MP = {MPd * (-BA.y / BAd), MPd * (BA.x / BAd)};
// tacka P
Tacka P = {M.x + MP.x, M.y + MP.y};
// vektor MQ
Vektor MQ = {MQd * (BA.y / BAd), MQd * (-BA.x / BAd)};
// tacka Q
Tacka Q = {M.x + MQ.x, M.y + MQ.y};
```

5.1.3 Скаларни производ

Скаларни производ $\vec{a} \circ \vec{b}$ вектора $\vec{a}$ и $\vec{b}$ је скалар (број):

$$\vec{a} \circ \vec{b} = |\vec{a}| \cdot |\vec{b}| \cdot \cos \phi,$$

где је са $|\vec{v}|$ означен интензитет вектора $\vec{v}$, а са ϕ конвексни угао (мањи или једнак π) који образују вектори $\vec{a}$ и $\vec{b}$. Једноставно се доказује да скаларни производ има следеће особине:

$$\begin{aligned}
\vec{a} \circ \vec{b} &= \vec{b} \circ \vec{a} && \text{симетричност} \\
\vec{a} \circ (\vec{b}_1 + \vec{b}_2) &= \vec{a} \circ \vec{b}_1 + \vec{a} \circ \vec{b}_1 && \text{десна дистрибутивност} \\
(\vec{a}_1 + \vec{a}_2) \circ \vec{b} &= \vec{a}_1 \circ \vec{b} + \vec{a}_2 \circ \vec{b} && \text{лева дистрибутивност} \\
\vec{a} \circ \vec{a} &\geq 0 && \text{ненегативност} \\
\vec{a} \circ \vec{a} = 0 &\Leftrightarrow \vec{a} = \vec{0} && \text{позитивна дефинитност} \\
(k \cdot \vec{a}) \circ \vec{b} &= \vec{a} \circ (k \cdot \vec{b}) = k \cdot (\vec{a} \circ \vec{b}) && \text{однос са скалирањем}
\end{aligned}$$

Ако умемо да израчунамо скаларни производ произвољна два вектора, једноставно је израчунати:

- интензитет, тј. дужину вектора $\vec{u}$, на основу формуле $|\vec{u}|^2 = \vec{u} \circ \vec{u}$, јер је $\cos 0 = 1$,
- конвексни угао ϕ између вектора $\vec{u}$ и $\vec{v}$, на основу формуле $\phi = \arccos \frac{\vec{u} \circ \vec{v}}{|\vec{u}| \cdot |\vec{v}|}$. Обратите пажњу на то да је вредност функције arccos увек између 0 и π, па се скаларним производом увек израчунава вредност конвексног угла између вектора.

Скаларни производ два не-нула вектора је нула ако и само ако је $\cos \phi = 0$, па се скаларни производ може користити и за проверу да ли су вектори међусобно нормални (при чему треба бити обазрив ако се ради са бројевима у покретном зарезу, јер се услед нумеричких грешака може добити вредност блиска, али не и једнака нули иако су вектори међусобно нормални). Негативна вредност скаларног производа указује на то да је мањи (конвексни) угао између вектора туп, а позитивна да је тај угао оштар.

Скаларни производ се лако израчунава и када су вектори задати аналитички, Декартовим координатама. Наиме, ако су два вектора $\vec{a}(a_1, \ldots, a_n)$ и $\vec{b}(b_1, \ldots, b_n)$ исте димензије, њихов скаларни производ рачунамо по формули:

$$\vec{a} \circ \vec{b} = \sum_{i=1}^{n} a_i \cdot b_i$$

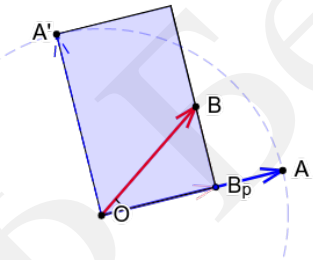
Ово се веома једноставно може доказати изражавањем вектора $\vec{a}$ и $\vec{b}$ преко својих координата и коришћењем основних особина скаларног производа.

Уколико су координате вектора $\vec{a}$ и $\vec{b}$ целобројне, и вредност скаларног производа је целобројна.

У даљем тексту ћемо разматрати векторе одређене тачкама у равни, те ће димензија вектора бити једнака 2. Скаларни производ вектора $\vec{a}(a_x, a_y)$ и вектора $\vec{b}(b_x, b_y)$ димензије 2 једнак је $\vec{a} \circ \vec{b} = a_x \cdot b_x + a_y \cdot b_y$.

5.1.3.1 Пројекција вектора на правац вектора

Аналитичко израчунавање скаларног производа подразумева коришћење координата вектора, које су заправо одређене пројекцијом вектора на координатне осе. Уместо тога, могуће је одредити *пројекцију једног вектора на правац другог*. Тиме се добијају два колинеарна вектора чији се скаларни производ добија множењем њихових интензитета (водећи рачуна о знаку).



Слика 5.3: Веза између скаларног производа и пројекције. Апсолутна вредност скаларног производа једнака је површини правоугаоника одређеног векторима $\overrightarrow{OB_p}$ и $\overrightarrow{OA'}$.

На слици 5.3, тачка B_p је пројекција тачке B (крајње тачке вектора $\overrightarrow{OB}$) на правац вектора $\overrightarrow{OA}$. Дужина OB_p је једнака апсолутној вредности израза $|\overrightarrow{OB}| \cos \phi$ (где је ϕ конвексни угао између вектора $\overrightarrow{OA}$ и $\overrightarrow{OB}$), док негативан знак тог израза указује на то да је вектор $\overrightarrow{OB_p}$ супротно усмерен од вектора $\overrightarrow{OA}$, а позитиван знак на то да су та два вектора исто усмерена. Зато је апсолутна вредност скаларног производа $\overrightarrow{OA} \circ \overrightarrow{OB} = |\overrightarrow{OA}||\overrightarrow{OB}| \cos \phi$ једнака производу дужина OB_p и OA, док је знак скаларног производа негативан ако су вектори $\overrightarrow{OB_p}$ и $\overrightarrow{OA}$ супротно усмерени, а позитиван ако су исто усмерени. Пројекција помаже и да се вредност скаларног производа визуализује. Да бисмо визуелно представили апсолутну вредност скаларног производа, вектор $\overrightarrow{OA}$ ротирамо за 90° тј. $\frac{\pi}{2}$ радијана, чиме се добија вектор $\overrightarrow{OA'}$, који са вектором $\overrightarrow{OB_p}$ гради правоугаоник (приказан плавом бојом на слици 5.3) чија је површина једнака апсолутној вредности скаларног производа.

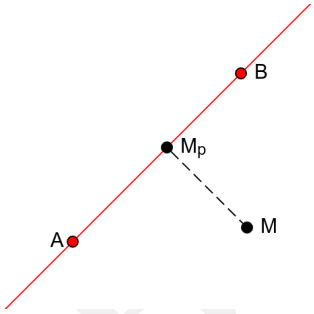
Наравно, пошто је све симетрично, могуће је пројектовати тачку A на правац вектора $\overrightarrow{OB}$.

5.1.3.2 Пројекција тачке на праву

Видели смо да је скаларни производ у тесној вези са пројекцијом тачке на праву, тј. на правац једног од вектора који се множе. Стога се та пројекција веома једноставно одређује коришћењем скаларног производа.

Проблем

Израчунати координате нормалне пројекције M_p тачке M на праву AB (слика 5.4).



Слика 5.4: Пројекција тачке M на праву AB.

За тачку M_p важи $M_p = A + \overrightarrow{AM_p}$. Вектор $\overrightarrow{AM_p}$ колинеаран је са вектором $\overrightarrow{AB}$, а дужина вектора $\overrightarrow{AM_p}$ једнака је апсолутној вредности израза $|AM|\cos\phi$ (где је ϕ угао између вектора $\overrightarrow{AB}$ и $\overrightarrow{AM}$). Вредност скаларног производа вектора $\overrightarrow{AB}$ и $\overrightarrow{AM}$ једнака је $|AB||AM|\cos\phi$, па је дужина вектора $\overrightarrow{AM_p}$ једнака апсолутној вредности израза

$$\frac{\overrightarrow{AB} \circ \overrightarrow{AM}}{|\overrightarrow{AB}|} \tag{5.1}$$

при чему знак ове вредности говори о томе да ли су вектори $\overrightarrow{AB}$ и $\overrightarrow{AM_p}$ исто или супротно усмерени. Вектор $\overrightarrow{AM_p}$ се може добити тако што се јединични вектор $\frac{\overrightarrow{AB}}{|\overrightarrow{AB}|}$ помножи вредношћу израза (5.1). Зато је

$$M_p = A + \overrightarrow{AM_p} = A + \frac{\overrightarrow{AB} \circ \overrightarrow{AM}}{|\overrightarrow{AB}|^2}\overrightarrow{AB}.$$

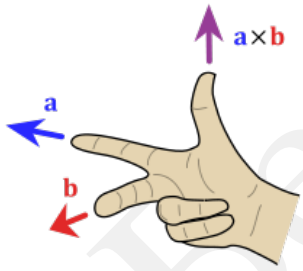
Нагласимо да се ова формула може користити и за тачке у равни и за тачке у простору.

5.1.4 Векторски производ

Векторски производ вектора $\vec{a}(a_x, a_y, a_z)$ и $\vec{b}(b_x, b_y, b_z)$ димензије 3 је вектор управан на раван одређену векторима $\vec{a}$ и $\vec{b}$, чији је смер одређен правилом десне руке (слика[1] 5.5), а интензитет једнак површини паралелограма који одређују вектори $\vec{a}$ и $\vec{b}$, односно може се израчунати коришћењем формуле

$$|\vec{a} \times \vec{b}| = |\vec{a}| \cdot |\vec{b}| \cdot \sin \phi,$$

где је са ϕ означен мањи од углова који образују вектори $\vec{a}$ и $\vec{b}$ (тј. конвексни угао, чији је синус увек позитиван).



Слика 5.5: Правило десне руке: ако кажипрст и средњи прст показују редом у смеру вектора $\vec{a}$ и $\vec{b}$, палац ће одређивати смер њиховог векторског производа $\vec{a} \times \vec{b}$.

Лако се показују основна својства векторског производа:

$$\begin{aligned}
\vec{a} \times \vec{b} &= -\vec{b} \times \vec{a} && \text{антикомутативност} \\
\vec{a} \times (\vec{b}_1 + \vec{b}_2) &= \vec{a} \times \vec{b}_1 + \vec{a} \times \vec{b}_1 && \text{десна дистрибутивност} \\
(\vec{a}_1 + \vec{a}_2) \times \vec{b} &= \vec{a}_1 \times \vec{b} + \vec{a}_2 \times \vec{b} && \text{лева дистрибутивност} \\
\vec{a} \times \vec{a} &= 0 && \text{самопроизвод} \\
(k \cdot \vec{a}) \times \vec{b} = \vec{a} \times (k \cdot \vec{b}) &= k \cdot (\vec{a} \times \vec{b}) && \text{однос са скалирањем}
\end{aligned}$$

Нагласимо и да нам скаларни производ даје начин да лако одредимо косинус угла између два вектора, а одатле и величину конвексног угла између њих. Са друге стране, из векторског производа се лако одређује вредност синуса угла између два вектора, међутим, вредност синуса није довољна да се лако одреди конвексни угао (јер исту вредност

[1] Слика је преузета са Википедије (https://commons.wikimedia.org/wiki/File:Right_hand_rule_cross_product.svg).

синуса дају један оштар и један туп угао). Стога се за одређивање величине конвексног угла између два вектора користи скаларни, а не векторски производ.

Пошто важи да је:

$$\begin{aligned}\vec{a} &= a_x \cdot \vec{i} + a_y \cdot \vec{j} + a_z \cdot \vec{k} \\ \vec{b} &= b_x \cdot \vec{i} + b_y \cdot \vec{j} + b_z \cdot \vec{k}\end{aligned}$$

где су са $\vec{i}$, $\vec{j}$ и $\vec{k}$ означени међусобно нормални јединични вектори у смеру x, y и z координатне осе, и истовремено је:

$$\begin{aligned}\vec{i} \times \vec{j} = \vec{k} = -\vec{j} \times \vec{i} \\ \vec{j} \times \vec{k} = \vec{i} = -\vec{k} \times \vec{j} \\ \vec{k} \times \vec{i} = \vec{j} = -\vec{i} \times \vec{k}\end{aligned}$$

важи да је векторски производ вектора $\vec{a}$ и $\vec{b}$ једнак:

$$\vec{a} \times \vec{b} = (a_y b_z - a_z b_y)\vec{i} + (a_z b_x - a_x b_z)\vec{j} + (a_x b_y - a_y b_x)\vec{k}$$

Ова формула се може записати у облику наредне детерминанте:

$$\vec{a} \times \vec{b} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$$

у чијој се првој врсти налазе јединични вектори у смеру x, y и z осе, у другој врсти координате вектора $\vec{a}$, а у трећој координате вектора $\vec{b}$. Уколико су координате вектора $\vec{a}$ и $\vec{b}$ целобројне, и координате вектора $\vec{a} \times \vec{b}$ су целобројне.

5.1.4.1 Дводимензионални векторски производ

Векторски производ је дефинисан за векторе у тродимензионом простору. Ипак, и вектори у равни се могу векторски множити ако се раван xOy утопи у тродимензиони простор. Тиме се координате сваког од два вектора равни прошире нулом (z координата тих вектора је нула) и изврши се векторско множење. Резултат је вектор који је управан на раван xOy, тј. вектор чије су прве две координате једнаке нули. Заиста

$$\vec{a} \times \vec{b} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & 0 \\ b_x & b_y & 0 \end{vmatrix} = (0, 0, a_x b_y - a_y b_x).$$

Пошто нам је у овом случају једино значајна z-координата, векторски производ два раванска вектора $\vec{a} = (a_x, a_y)$ и $\vec{b} = (b_x, b_y)$ се понекад поистовећује са скаларом $a_x b_y - a_y b_x$ (што може бити збуњујуће, јер називи скаларни и векторски производ потичу од тога што је у првом случају резултат множења скалар, а у другом вектор). Да бисмо избегли забуну, наглашаваћемо увек да је у питању множење дводимензионалних вектора (тј. *дводимензионални векторски производ*[2]) и уместо ознаке $\times$ користићемо $\times_{2d}$. Дакле, за дводимензионалне векторе $\vec{a} = (a_x, a_y)$ и $\vec{b} = (b_x, b_y)$ дефинишемо да је:

$$\vec{a} \times_{2d} \vec{b} = a_x b_y - a_y b_x = \begin{vmatrix} a_x & a_y \\ b_x & b_y \end{vmatrix}.$$

Ако дводимензионални вектори образују угао ϕ, вредност дводимензионалног векторског производа $\vec{a} \times_{2d} \vec{b}$ једнака је $|\vec{a}| \cdot |\vec{b}| \cdot \sin \phi$.

Колинеарност

Вредност $\vec{a} \times_{2d} \vec{b} = 0$ нам говори да су вектори $\vec{a}$ и $\vec{b}$ колинеарни, тј. да је угао који образују 0 или π и то се обично користи као потребан и довољан услов за испитивање колинеарности вектора. Заиста, пошто је $\vec{a} \times_{2d} \vec{b} = |\vec{a}| \cdot |\vec{b}| \cdot \sin \phi$, важи да је:

$$\sin \phi = \frac{\vec{a} \times_{2d} \vec{b}}{|\vec{a}| \cdot |\vec{b}|}.$$

Синус је једнак нули за угао 0 или π.

Зато једна од основних примена векторског производа може бити да се утврди да ли су три тачке колинеарне.

[2]Појам векторског производа (енгл. cross product) се у теоријској математици уопштава и на n-димензионалне векторске просторе, међутим, приказана дефиниција дводимензионалног векторског производа се не уклапа у ту општу дефиницију. Вредност $a \times_{2d} b$ се некада назива и *означена површина* (енгл. signed area) паралелограма одређеног преко векторима a и b у равни. У том светлу, она је аналогна операцији мешовитог производа тродимензионалних вектора, којом се рачуна означена запремина паралелепипеда ког три вектора граде. Вредност $a \times_{2d} b$ се некада назива и *нормални скаларни производ* (енгл. perp dot product), јер је $a \times_{2d} b = a^{\perp} \circ b$, где је са $a^{\perp}$ означен вектор нормалан на a, добијен ротацијом вектора a за 90 степени. Ипак термин векторски производ (енгл. cross product) је заступљен у литератури о геометријским алгоритмима и рачунарској графици, па ћемо га и ми користити.

Проблем

Дате су три тачке у простору (или у равни). Испитати да ли су оне колинеарне.

Тачка M у тродимензионалном простору припада правој која је одређена тачкама A и B ако и само ако је $\overrightarrow{MA} \times \overrightarrow{MB} = \vec{0}$ (или, на пример, $\overrightarrow{AB} \times \overrightarrow{AM} = \vec{0}$). За тачке A, B и M у равни, довољно је проверити услов $\overrightarrow{MA} \times_{2d} \overrightarrow{MB} = 0$. При томе треба бити обазрив ако се ради са реалним координатама (јер је због рачунских грешака мало вероватно да ће се добити вредности које су баш тачно једнаке нули).

Провера колинеарности се веома лако допуњује до провере да ли тачка припада дужи.

Проблем

Дате су различите тачке A, B и тачка M у равни. Испитати да ли тачка M припада затвореној дужи AB.

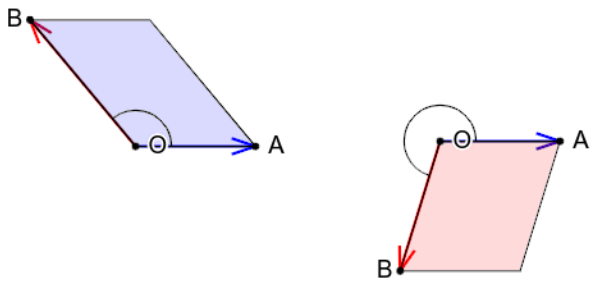
Тачка M припада дужи AB ако и само ако је колинеарна са A и B, ако се њена пројекција на x-осу налази између пројекција тачака A и B на x-осу и ако се њена пројекција на y-осу налази између пројекција тачака A и B на y-осу (тј. ако тачка M припада правоугаонику чија је дијагонала AB). Ако знамо да дуж AB није вертикална, тада је довољно проверити само пројекције на x-осу.

```
bool kolinearne(const Tacka& A, const Tacka& B, const Tacka& M) {
    return (B.x-A.x)*(M.y-A.y) == (M.x-A.x)*(B.y-A.y);
}

bool pripadaDuzi(const Tacka& A, const Tacka& B, const Tacka& M) {
   return kolinearne(A, B, M) &&
          min(A.x, B.x) <= M.x && M.x <= max(A.x, B.x) &&
          min(A.y, B.y) <= M.y && M.y <= max(A.y, B.y);
}
```

У претходној имплементацији се сматра да је дуж AB затворена. Ако се разматра отворена дуж AB, тада неједнакости у провери пројекција треба да буду строге.

Знак дводимензионалног векторског производа

Знак вредности дводимензионалног векторског производа $\vec{a} \times_{2d} \vec{b}$ нам говори о оријентисаном углу између вектора $\vec{a}$ и $\vec{b}$, при чему се сада угао посматра у интервалу $[0, 2\pi)$ (слика 5.6). Позитивне вредности говоре да је угао који се описује ротирањем вектора $\vec{a}$ ка вектору $\vec{b}$ у позитивном математичком смеру конвексан (у интервалу је $(0, \pi)$), а негативне вредности да је тај угао неконвексан (у интервалу је $(\pi, 2\pi)$). Ово, наравно, одговара знаку синуса угла.



Слика 5.6: Позитиван знак векторског производа указује на конвексан, а негативни на неконвексан оријентисани угао између вектора $\overrightarrow{OA}$ и $\overrightarrow{OB}$.

Још једна интерпретација је да позитивни знак дводимензионалног векторског производа $\vec{a} \times_{2d} \vec{b}$ указује да је најближи пут од вектора $\vec{a}$ до вектора $\vec{b}$ „налево" тј. у позитивном математичком смеру, док негативни знак дводимензионалног векторског простора указује на то да је најближи пут „надесно".

Знак дводимензионалног векторског производа се може искористити и да се тачке класификују у две полуравни на које права AB дели раван. Негативна вредност векторског производа $\overrightarrow{OA} \times_{2d} \overrightarrow{OB}$ указује да тачка O припада једној полуравни, а позитивна да припада другој (док вредност 0 указује на то да тачка O припада граничној правој AB).

Да резимирамо, следећи услови су међусобно еквивалентни:

- Знак дводимензионалног векторског производа $\overrightarrow{OA} \times_{2d} \overrightarrow{OB}$ је позитиван.
- Најближи пут од вектора $\overrightarrow{OA}$ до вектора $\overrightarrow{OB}$ је „налево" тј. у позитивном математичком смеру.
- Угао од вектора $\overrightarrow{OA}$ до вектора $\overrightarrow{OB}$ у позитивном математичком смеру је конвексан.

У поглављу 5.1.6 видећемо да се знак векторског производа користи и за одређивање оријентације тројке тачака, што је фундаментална операција у многим геометријским алгоритмима.

5.1.5 Површина и примене

Проблем

Израчунати површину троугла чија су темена три тачке у равни задате својим координатама.

Површина троугла чије су странице два дата вектора $\vec{a}$ и $\vec{b}$ је половина површине паралелограма који они граде и једнака је половини апсолутне вредности њиховог векторског производа. Ако су познате координате темена троугла $A(a_x, a_y)$, $B(b_x, b_y)$ и $C(c_x, c_y)$, вектори страница су $\overrightarrow{AB} = \overrightarrow{OB} - \overrightarrow{OA} = (b_x - a_x, b_y - a_y)$ и $\overrightarrow{AC} = \overrightarrow{OC} - \overrightarrow{OA} = (c_x - a_x, c_y - a_y)$, па је површина једнака

$$P_{ABC} = \frac{|\overrightarrow{AB} \times_{2d} \overrightarrow{AC}|}{2} = \frac{|(b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)|}{2}.$$

```
double PovrsinaTrougla(const Tacka& A, const Tacka& B, const Tacka& C) {
   return abs((B.x - A.X)*(C.y - A.y) - (B.y - A.y)*(C.x - A.x) / 2.0;
}
```

Површина не мора бити цео број чак ни када су координате целобројне (због дељења са 2). Да би се избегао рад са реалним бројевима, некада се у коду дефинише функција која израчунава двоструку површину а касније формуле се, ако је могуће, модификују тако да уместо површине користе двоструку површину.

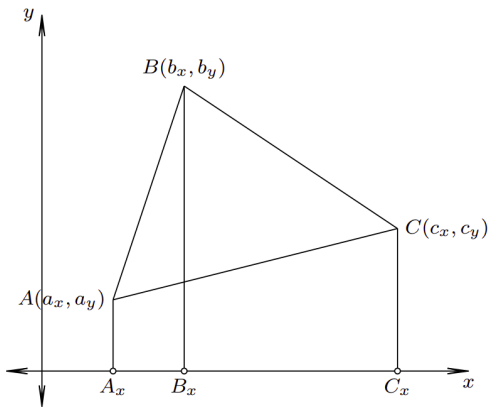
Изразом $\frac{|\overrightarrow{AB} \times \overrightarrow{AC}|}{2}$ се може рачунати и површина троугла у простору (али је тада израз којим се површина изражава помоћу координата тачака компликованији).

Површину је могуће израчунати и на друге начине, али је то често или неефикасно или се коначна формула своди на претходну. На пример, Херонов образац $s = (a + b + c)/2$, $P = \sqrt{s(s-a)(s-b)(s-c)}$ укључује кореновање, што овај присуп чини неефикасним.

Још један начин доласка до решења је да се површина троугла искаже као разлика површина одређених трапеза. На пример, ако важи распоред $a_x < b_x < c_x$, и ако су A_x, B_x и C_x редом пројекције тачака A, B и C на x-осу, тада је површина троугла ABC једнака разлици између збира површина правоуглих трапеза AA_xB_xB, BB_xC_xC и површине правоуглог трапеза AA_xC_xC (слика 5.7).

Површина трапеза AA_xB_xB једнака је производу дужине његове средње линије и дужине његове висине. Дужина основице A_xA једнака је апсолутној вредности координате a_y, дужина основице B_xB једнака је апсолутној вредности координате b_y док је дужина висине једнака апсолутној вредности разлике координата b_x и a_x. Тако да је површина тог трапеза једнака

$$\frac{|b_x - a_x|(|a_y| + |b_y|)}{2} \tag{5.2}$$



Слика 5.7: Површина троугла се може изразити помоћу површина трапеза.

На сличан начин могу се извести и формуле за друга два трапеза.

Анализом могућих распореда тачака, може се показати да ће се исправан резултат добити и ако се посматра такозвана означена површина трапеза, која искључује рачунање апсолутне вредности у формули (5.2) и допушта негативне дужине и површине. Наиме, означена површина троугла ABC биће једнака збиру означених површина три поменута правоугла трапеза (AA_xB_xB, BB_xC_xC и CC_xA_xA), а права површина троугла биће једнака њеној апсолутној вредности. Означене површине ових трапеза добијају се тако што се у претходно изведеним формулама за њихову површину занемаре апсолутне вредности, тако да се површина троугла добија следећом формулом.

$$P_{ABC} = \frac{|(b_x - a_x)(b_y + a_y) + (c_x - b_x)(c_y + b_y) + (a_x - c_x)(a_y + c_y)|}{2}$$

Након сређивања добија се формула

$$P_{ABC} = \frac{|a_x b_y + b_x c_y + c_x a_y - a_x c_y - c_x b_y - b_x a_y|}{2}$$

Ова формула се некада назива *правило пертле* (енгл. shoelace formula). Заиста, ако се координате тачака запишу у наредном облику

$$\begin{matrix} a_x & a_y \\ b_x & b_y \\ c_x & c_y \\ a_x & a_y \end{matrix}$$

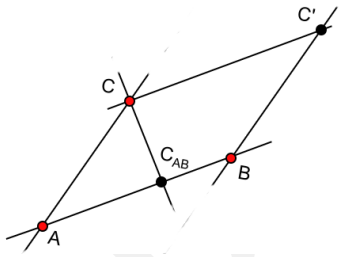
формула се гради тако што се са једним знаком узимају производи одозго-наниже, слева-удесно (то су $a_x b_y$, $b_x c_y$ и $c_x a_y$), док се са супротним знаком узимају производи одоздо-навише, слева удесно (то су $a_x c_y$, $c_x b_y$ и $b_x a_y$), што, када се нацрта, заиста подсећа на цик-цак везивање пертли.

Приметимо да се након сређивања формуле добијене када се површина троугла рачуна помоћу векторског производа такође добија формула пертле. Видећемо да се формула пертле може уопштити и на израчунавање површине одређених врста многоуглова.

5.1.5.1 Растојање тачке од праве

Проблем

За дате тачке A, B и C у равни, одредити растојање од C до праве AB.



Слика 5.8: Израчунавање растојања тачке C од праве AB.

Површина паралелограма одређеног векторима $\overrightarrow{AB}$ и $\overrightarrow{AC}$ (слика 5.8) је са једне стране једнака апсолутној вредности дводимензионалног векторског производа вектора његових страница, док је са друге стране једнака производу дужине његове основе AB и висине CC_{AB}. Ово се може употребити да би се израчунало растојање тачке C од праве одређене тачкама A и B, јер то растојање представља висину CC_{AB} паралелограма одређеног векторима $\overrightarrow{AB}$ и $\overrightarrow{AC}$. Пошто је дужина основице једнака $|\overrightarrow{AB}|$, висина тј. растојање тачке C до праве AB је једнако

$$\frac{|\overrightarrow{AB} \times_{2d} \overrightarrow{AC}|}{|\overrightarrow{AB}|} = \frac{|(b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)|}{\sqrt{(b_x - a_x)^2 + (b_y - a_y)^2}}$$

Наредном функцијом се одређује растојање дате тачке од дате праве.

```
// rastojanje tacke C od prave odredjene tackama A i B
double rastojanje(const Tacka& A, const Tacka& B, const Tacka& C) {
  double dx = B.x - A.x;
```

```
    double dy = B.y - A.y;
    return abs(dx*(C.y-A.y) - dy*(C.x-A.x)) / sqrt(dx*dx + dy*dy);
}
```

Изразом $\frac{|\overrightarrow{AB}\times\overrightarrow{AC}|}{|\overrightarrow{AB}|}$ се може рачунати и растојање од тачке до праве у простору.

Проблем

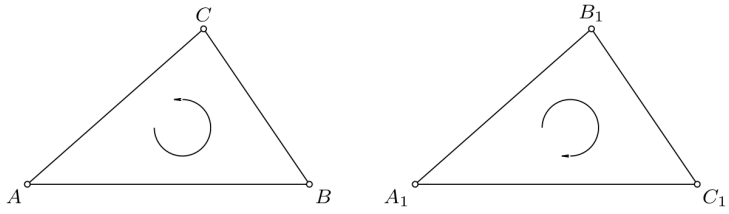
За дати скуп тачака S у равни, одредити најкраће растојање од неке тачке $C \in S$ до праве AB.

Ако је потребно одредити најближу или најдаљу тачку неког скупа од праве AB (а не и њихова растојања), довољно је израчунати и поредити само вредности бројиоца, јер ће за све тачке вредност имениоца бити иста.

5.1.6 Оријентација тројке тачака и примене

Уређена тројка неколинеарних тачака у равни може имати:

- оријентацију у смеру супротном од кретања казаљке на часовнику – математички позитивна оријентација (слика 5.9 (а))
- оријентацију у смеру кретања казаљке на часовнику – математички негативна оријентација (слика 5.9 (б))



Слика 5.9: (а) Троугао ABC има математички позитивну оријентацију. (б) Троугао $A_1B_1C_1$ има математички негативну оријентацију.

Важе наредна својства:

- ако уређена тројка ABC има позитивну (негативну) оријентацију, онда и уређена тројка BCA има позитивну (негативну) оријентацију (па и тројка CAB);
- ако уређена тројка ABC има позитивну (негативну) оријентацију, онда уређена тројка BAC има негативну (позитивну) оријентацију (па и тројке ACB и CBA).

Оријентација се може засновати и аксиоматски[3], али се лако може аналитички израчунати када су познате координате тачака $A(a_x, a_y)$, $B(b_x, b_y)$, $C(c_x, c_y)$. Оријентацију тројке ABC дефинишемо као знак дводимензионалног векторског производа:

$$\overrightarrow{AB} \times_{2d} \overrightarrow{AC} = (b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x).$$

Пошто постоје три могуће оријентације, у циљу постизања читљивог програма могуће је дефинисати набројиви тип за представљање оријентације.

```
// moguce orijentacije trojke tacaka
enum Orijentacija {POZITIVNA, KOLINEARNE, NEGATIVNA};
```

У неким случајевима можемо сматрати и да је оријентација цео број и то 1 за позитивну оријентацију, -1 за негативну и 0 за колинеарне тачке.

Када су координате тачака целобројне, функција за израчунавање оријентације се веома једноставно дефинише.

```
// orijentacija trojke tačka sa koordinatama A, B i C
// orijentacija je pozitivna akko je orijentisani ugao ABC konveksan
Orijentacija orijentacija(const Tacka& A, const Tacka& B, const Tacka& C) {
  int d = (B.x-A.x)*(C.y-A.y) - (C.x-A.x)*(B.y-A.y);
  if (d > 0)
    return POZITIVNA;
  else if (d < 0)
    return NEGATIVNA;
  else
    return KOLINEARNE;
}
```

Ствар је компликованија када се ради са координатама које су записане помоћу бројева у покретном зарезу. Наиме, класификовање оријентације, а посебно испитивање колинеарности може бити веома проблематично услед грешака у запису бројева. Наиме, уместо да вредност дводимензионалног векторског производа буде тачно нула, она може да буде неки веома мали број близак нули. Најједноставније је да се тада фиксира нека вредност ε и да се, када год је вредност производа у интервалу $[-\varepsilon, \varepsilon]$, тачке прогласе за колинеарне. Међутим, није унапред јасно колика би требало да буде та вредност ε. Вредност векторског производа не зависи само од угла између вектора, већ и од њихове дужине, па би прецизнији приступ би био да се на основу вредности

[3] Аксиоматско заснивање оријентације детаљно је описано у књизи Доналда Кнута „Аксиоме и омотачи”.

дводимензионалног векторског производа израчуна синус угла између вектора и да се за проверу колинеарности захтева да он припада неком малом интервалу, међутим, то је рачунски захтевније. Приметимо да у неким применама није превише битно како ће се класификовати тачке које су скоро колинеарне. На пример, у рачунарској графици, како год да се класификују тачке које су готово колинеарне, људско око неће приметити разлику.

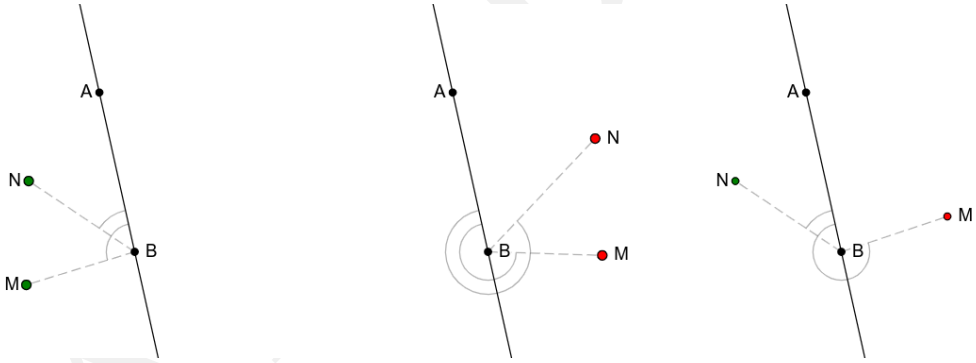
5.1.6.1 Провера да ли су тачке са исте стране праве

Проблем

За дате тачке M и N утврдити да ли су са исте стране праве p одређене тачкама A и B ($A \neq B$).

Једноставности ради претпоставићемо да ниједна од тачака M и N не припада правој AB.

Тачке M и N су са исте стране праве p ако и само ако су троуглови ABM и ABN, $A, B \in p$ исте оријентације (слика 5.10). Дакле, довољно је да испитамо да ли су одговарајући дводимензионални векторски производи истог знака.



Слика 5.10: У првом случају тачке M и N се налазе са исте стране праве AB јер су обе оријентације ABM и ABN позитивне (па су оба угла ABM и ABN конвексна). У другом случају су тачке поново са исте стране јер су обе оријентације ABM и ABN негативне (па су оба та угла неконвексна). У трећем случају тачке се налазе са разних страна, јер је једна оријентација позитивна, а једна негативна (па је један угао конвексан, а други неконвексан).

Када имамо на располагању функцију за рачунање оријентације, имплементација провере да ли су две тачке са исте стране праве је веома једноставна.

```
// provera da li su tačke M i N sa iste strane prave
// određene tačkama A i B.
// Pretpostavlja se da nijedna od dve tačke ne pripada toj pravoj.
bool saIsteStrane(const Tacka& A, const Tacka& B,
                  const Tacka& M, const Tacka& N) {
  return orijentacija(A, B, M) == orijentacija(A, B, N);
}
```

Проширимо сада захтев тиме да желимо да ако су тачке са разних страна праве одредимо и пресечну тачку дужи која их спаја и праве (при чему је потребно размотрити и могућност да M или N припада правој AB).

Проблем

За дате тачке M и N утврдити да ли су са различитих страна праве p одређене тачкама A и B ($A \neq B$) и ако јесу, одредити координате пресечне тачке дужи MN и праве p.

Наредни метод је заснован на коришћењу дводимензионалног векторског производа, али не за израчунавање оријентације тројке тачака, јер ћемо користити производе вектора облика $\overrightarrow{XY} \times_{2d} \overrightarrow{ZW}$ (који, подсетимо се, дају вредност означене површине паралелограма одређеног помоћу та два вектора).

Дегенерисани случај наступа када су вектори $\overrightarrow{AB}$ и $\overrightarrow{MN}$ колинеарни, што се дешава ако и само ако је $\overrightarrow{AB} \times_{2d} \overrightarrow{MN} = 0$. Ако су у том случају тачке M, A и B колинеарне тј. ако је $\overrightarrow{MA} \times_{2d} \overrightarrow{MB} = 0$, тада дуж припада правој, а ако нису колинеарне, онда пресек не постоји.

У недегенерисаном случају је $\overrightarrow{AB} \times_{2d} \overrightarrow{MN} \neq 0$. Тада постоји пресечна тачка X правих AB и MN и важи да је:

$$X = M + k\overrightarrow{MN}, \quad \text{тј.} \quad \overrightarrow{MX} = k\overrightarrow{MN}.$$

Тачка X припада дужи MN ако и само ако је $0 \leq k \leq 1$. Пошто је $\overrightarrow{MX} = \overrightarrow{MA} + \overrightarrow{AX}$, множењем претходне једначине векторски са $\overrightarrow{AB}$ слева, добија се:

$$\overrightarrow{AB} \times_{2d} (\overrightarrow{MA} + \overrightarrow{AX}) = k \cdot (\overrightarrow{AB} \times_{2d} \overrightarrow{MN})$$

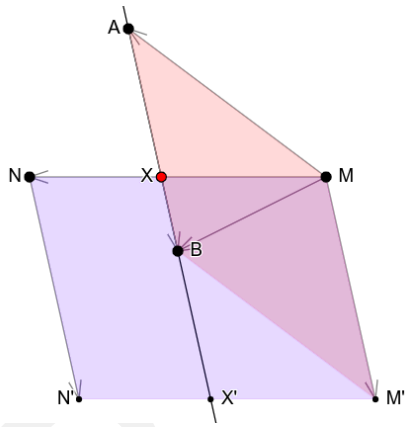
Пошто важи да је $\overrightarrow{AB} \times_{2d} \overrightarrow{AX} = 0$, јер тачка X припада правој AB, па су ти вектори колинеарни, можемо израчунати коефицијент k.

$$k = \frac{\overrightarrow{AB} \times_{2d} \overrightarrow{MA}}{\overrightarrow{AB} \times_{2d} \overrightarrow{MN}}$$

Да би се проверило да ли тачка X припада дужи MN довољно је проверити:

1. да ли су бројеви $\overrightarrow{AB} \times_{2d} \overrightarrow{MA}$ и $\overrightarrow{AB} \times_{2d} \overrightarrow{MN}$ истог знака и
2. да ли је $|\overrightarrow{AB} \times_{2d} \overrightarrow{MA}| \leq |\overrightarrow{AB} \times_{2d} \overrightarrow{MN}|$, што се све може урадити и у целобројној аритметици (ако су почетне координате тачака целобројне).

Ова техника има и јасну геометријску интерпретацију (слика 5.11).



Слика 5.11: Геометријска интерпретација одређивања коефицијента k.

Нека је $M' = M + \overrightarrow{AB}$ и $N' = N + \overrightarrow{AB}$. Вредност $\overrightarrow{AB} \times_{2d} \overrightarrow{MA}$ представља означену површину паралелограма одређеног векторима $\overrightarrow{AB}$ и $\overrightarrow{MA}$ (он је обојен на слици 5.11). Та је површина једнака означеној површини паралелограма P_1 одређеног векторима $\overrightarrow{MX}$ и $\overrightarrow{XX'}$. Вредност $\overrightarrow{AB} \times_{2d} \overrightarrow{MN}$ представља означену површину паралелограма P_2 одређеног векторима $\overrightarrow{AB} = \overrightarrow{MM'}$ и $\overrightarrow{MN}$ (и он је обојен на слици 5.11). Пошто паралелограми P_1 и P_2 имају исту висину, однос њихових површина једнак је односу дужина њихових страница MX и MN.

Наравно, да би се дужи секле, потребно је и да коефицијент

$$k = \frac{\overrightarrow{AM} \times_{2d} \overrightarrow{MN}}{\overrightarrow{AB} \times_{2d} \overrightarrow{MN}}$$

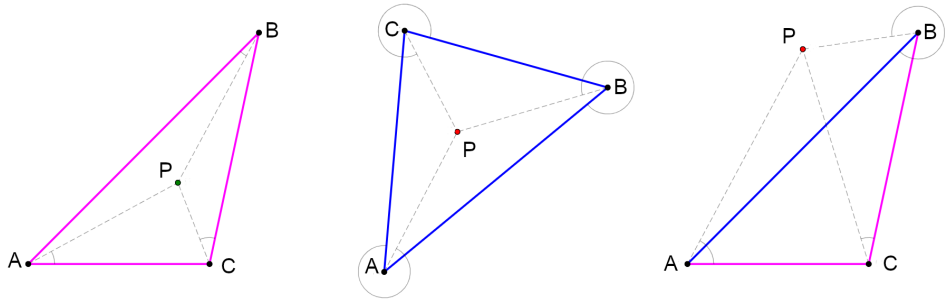
припада интервалу $[0, 1]$, што посебно треба проверити.

5.1.6.2 Испитивање да ли тачка припада унутрашњости троугла

Проблем

Дат је троугао ABC и тачка P. Испитати да ли тачка P припада унутрашњости троугла ABC или не.

Као што је то обично случај у геометрији, треба обратити пажњу на дегенерисане и граничне случајеве. На пример, ми ћемо, једноставности ради, претпостављати да је троугао недегенерисан, тј. да су тачке A, B и C неколинеарне. Сматраћемо да трогуао чине темена и тачке на страницама. Поред њих, разликоваћемо тачке у унутрашњости и спољашњости троугла. Под *отвореним троуглом* подразумеваћемо само његову унутрашњост, а под *затвореним троуглом* унију троугла (његових темена и страница) са његовом унутрашњошћу. У неким проблемима (на пример, провери припадности тачке троуглу) је веома важно разликовати отворене и затворене троуглове и тада ћемо јасно наглашавати са каквим ликом радимо, док је у неким проблемима то ирелевантно (на пример, за израчунавање површине троугла).



Слика 5.12: У првом случају тачка P припада унутрашњости троугла ABC и све тројке ABP, BCP и CAP су негативно оријентисане (сви углови ABP, BCP и CAP су конвексни). У другом случају тачка P припада унутрашњости троугла ABC, а све три тројке тачака су позитивно оријентисане (сви углови су неконвексни). У трећем случају тачка P не припада унутрашњости троугла ABC и тројка ABP је позитивно оријентисана (угао ABP је неконвексан), док су тројке BCP и CAP негативно оријентисане (углови BCP и CAP су конвексни).

Тачка припада унутрашњости троугла ако и само ако су тачке C и P са исте стране праве AB, тачке A и P са исте стране праве BC и тачке B и P са исте стране праве CA. Ако то важи, тада је иста оријентација тројки ABC и ABP, иста оријентација тројки BCA и BCP и тројки CAB и CAP. Пошто су оријентације тројки ABC, BCA и CAB једнаке, једнаке су и оријентације тројки ABP, BCP и CAP. Показује се да је ово довољан услов да би тачка P припадала отвореном троуглу ABC (слика 5.12).

Обратимо пажњу да овај услов не обухвата припадност страницама троугла: наиме, ако би тачка припадала некој страници, нека оријентација би била једнака нули и не би било могуће да су све оријентације међусобно једнаке.

Када имамо функцију за израчунавање оријентације на располагању, имплементација провере да ли тачка припада отвореном троуглу је веома једноставна.

```
bool tackaUOtvorenomTrouglu(const Tacka& T,
                            const Tacka& A, const Tacka& B, const Tacka& C) {
  Orijentacija o1 = orijentacija(A, B, T);
  Orijentacija o2 = orijentacija(B, C, T);
  Orijentacija o3 = orijentacija(C, A, T);
  return o1 == o2 && o2 == o3;
}
```

Функција за проверу да ли тачка припада затвореном троуглу треба да узме у обзир и да ли тачка припада страницама троугла у случају када је нека оријентација нула.

```
bool tackaUZatvorenomTrouglu(const Tacka& T,
                             const Tacka& A, const Tacka& B, const Tacka& C) {
  Orijentacija o1 = orijentacija(A, B, T);
  Orijentacija o2 = orijentacija(B, C, T);
  Orijentacija o3 = orijentacija(C, A, T);
  if (o1 == 0 && pripadaDuzi(A, B, T)) return true;
  if (o2 == 0 && pripadaDuzi(B, C, T)) return true;
  if (o3 == 0 && pripadaDuzi(C, A, T)) return true;
  return o1 == o2 && o2 == o3;
}
```

Функција која проверава припадност тачке дужи је приказана у поглављу 5.1.4.1. Пошто се та провера врши само када се већ зна да су одговарајуће тачке колинеарне (јер је оријентација 0), довољно је само проверити пројекције на x и на y-осу.

Још један приступ провери да ли тачка припада затвореном троуглу је да се провери да ли је збир површина троуглова ABP, BCP и CAP једнак површини троугла ABC. Овај приступ може бити рачунски неефикаснији (нарочито ако се површине рачунају Хероновим обрасцем, који укључује кореновање). Додатно, ако су тачке представљене реалним координатама, овај приступ може да буде нумерички нестабилан. Наиме, и за тачку која је унутар троугла, може да се деси да збир површина три мања троугла не буде тачно једнак, већ само близак површини већег троугла. Зато је једнакост потребно проверавати до на неку тачност ε (при чему остаје осетљиво питање како одредити ту вредност ε). Из свих наведених разлога, овај приступ је пожељно избегавати.

```
bool tackaUZatvorenomTrouglu(const Tacka& T,
                             const Tacka& A, const Tacka& B, const Tacka& C) {
  // racunamo povrsinu trougla ABC
  double P = povrsinaTrougla(A, B, C);
  // racunamo povrsine trouglova TBC, TAC i TAB
  double PA = povrsinaTrougla(T, B, C);
  double PB = povrsinaTrougla(A, T, C);
  double PC = povrsinaTrougla(A, B, T);
  // proveravamo da li one u zbiru daju povrsinu trougla ABC
  // poredimo dve realne vrednosti na jednakost
  return abs(P - (PA + PB + PC)) < EPS;
}
```

5.1.6.3 Пресек дужи

Проблем

За дужи AB и CD задате координатама тачака A, B, C и D одредити да ли се секу и, ако се секу, одредити бар једну пресечну тачку. Претпоставити да су дужи затворене, тј. да садрже своје крајње тачке.

Ако су све тачке у општем положају, тј. све тачке су различите и никоје три нису колинеарне, дужи се секу ако и само ако дуж AB сече праву CD и дуж CD сече праву AB, па се испитивање пресека две дужи може лако свести на испитивање пресека дужи и праве (што је описано у поглављу 5.1.6.1). Дуж сече праву ако и само ако су њене крајње тачке са разних страна те праве, а то је проблем који се лако решава помоћу испитивања оријентације тачака. Довољно је проверити да ли су оријентације тројки CDA и CDB различитог знака и да ли су оријентације тројки ABC и ABD различитог знака, тј. да ли им је производ негативан (при чему се за оријентацију тројке XYZ може узети знак дводимензионалног векторског производа $\overrightarrow{XY} \times_{2d} \overrightarrow{XZ}$). Овај тест се лако прилагођава случајевима у којима су неке три тачке колинеарне. Наиме, уместо да се захтева да су две тачке са стриктно различитих страна праве, могуће је укључити и случај да нека од њих припада правој, што се своди на испитивање да ли је оријентација непозитивна (вредност јој је мања или једнака од нуле).

Проблем представља дегенерисани случај када су све четири тачке колинеарне. Наиме, ако су све тачке колинеарне, све оријентације тројки биће једнаке 0 и из тога нећемо моћи да извучемо никакав закључак о међусобном односу две дужи (знамо да оне припадају једној правој, али су сви њихови међусобни односи могући). У том случају њихов однос је могуће испитати испитивањем њихових пројекција на x-осу (тј. њихових x-координата). Нажалост, ни ово није довољно у случају када све четири тачке имају исту x координату, но тада можемо испитати њихове пројекције на y-осу (тј. њихове

y-координате). Да би се пројекције на x-осу $[a_x, b_x]$ и $[c_x, d_x]$ секле довољно је да не важи да је један од интервала лево од другог тј. не сме да важи $b_x < c_x$ нити $d_x < a_x$, што је еквивалентно томе да је $b_x \geq c_x \wedge d_x \geq a_x$. Приликом примене овога услова потребно је обезбедити да се зна шта је леви, а шта десни крај интервала тј. да је $a_x \leq b_x$ и $c_x \leq d_x$. Аналогно се поставља услов који мора да важи и за пројекције на y-осу.

Следи имплементација провере да ли се две дужи секу. Функција која израчунава оријентацију враћа вредности -1, 0, 1, чиме је кôд можда мало мање читљив, али је омогућено да се провера да ли су два броја истог знака изврши тако што се провери да ли је производ две оријентације ненегативан. Обратимо пажњу на то да уместо вредности векторског производа функција враћа њен знак, чиме се избегава опасност од прекорачења приликом множења две оријентације.

```
struct Tacka {
  int x, y;
};

// 1 za pozitivne vrednosti, -1 za negativne i 0 za nulu
int znak(long long x) {
  return (x > 0) - (x < 0);
}

// orijentacija tj. znak vektorskog proizvoda AB x AC
int orijentacija(const Tacka& A, const Tacka& B, const Tacka& C) {
  return znak((B.x - A.x) * (C.y - A.y) - (C.x - A.x) * (B.y - A.y));
}

// provera da li prava AB sece duz MN
bool pravaSeceDuz(const Tacka& A, const Tacka& B,
                  const Tacka& M, const Tacka& N) {
  return orijentacija(A, B, M) * orijentacija(A, B, N) <= 0;
}

// provera da li se intervali realne prave [a, b] i [c, d] seku
// (pri cemu se ne zna odnos vrednosti a i b tj. c i d)
bool projekcijeSeSeku(long long a, long long b, long long c, long long d) {
  return max(a, b) >= min(c, d) && max(c, d) >= min(a, b);
}

// provera da li se duzi AB i CD seku
```

```
bool duziSeSeku(const Tacka& A, const Tacka& B,
                const Tacka& C, const Tacka& D) {
  return pravaSeceDuz(A, B, C, D) &&
         pravaSeceDuz(C, D, A, B) &&
         // seku se projekcije na x osu
         projekcijeSeSeku(A.x, B.x, C.x, D.x) &&
         // seku se projekcije na y osu
         projekcijeSeSeku(A.y, B.y, C.y, D.y);
}
```

Функција коју смо имплементирали исправно ради у свим случајевима (и недегенерисаним и дегенерисаним), али у многим случајевима врши неке непотребне провере услова (на пример, ако нису све четири тачке колинеарне, нема потребе проверавати пројекције на осе). Уз то, након што се детектује да пресек постоји, потребно је израчунати га сасвим изнова. Прикажимо сада другачији метод, који је мало ефикаснији и којим се може одмах одредити и пресечна тачка.

На почетку рачунамо $\overrightarrow{AB} \times_{2d} \overrightarrow{CD}$ и проверавамо да ли је тај број различит од нуле. Тиме смо ефективно испитали да ли су вектори $\overrightarrow{AB}$ и $\overrightarrow{CD}$ колинеарни.

1. Ако је $\overrightarrow{AB} \times_{2d} \overrightarrow{CD} \neq 0$, у питању је недегенерисани случај и испитујемо да ли постоји пресечна тачка праве AB и дужи CD, као и да ли постоји пресечна тачка праве CD и дужи AB, што можемо урадити на начин који је описан у поглављу 5.1.6.1. Пошто смо вредност $\overrightarrow{AB} \times_{2d} \overrightarrow{CD}$, већ израчунали, довољно је да се израчунају још само вредности $\overrightarrow{AC} \times_{2d} \overrightarrow{CD}$ и $\overrightarrow{AB} \times_{2d} \overrightarrow{CA}$. Приметимо да је потребно израчунати вредности само 3 дводимензионална векторска производа.
2. Ако је $\overrightarrow{AB} \times_{2d} \overrightarrow{CD} = 0$, у питању је дегенерисани случај. Ако тачке A, B и C нису колинеарне, тј. ако је $\overrightarrow{AB} \times_{2d} \overrightarrow{AC} \neq 0$, тада су праве AB и CD паралелне и пресек не постоји. У супротном су све ћетири тачке колинеарне и потребно је испитати пројекције на x-осу, осим у случају када су све на вертикалној правој, када можемо испитати пројекције на y-осу.

```
// z koordinata vektorskog proizvoda vektora MN i PQ
long long vektorski_proizvod_2d(const Tacka& M, const Tacka& N,
                                const Tacka& P, const Tacka& Q) {
   return (N.x - M.x)*(Q.y - P.y) - (N.y - M.y)*(Q.x - P.x);
}

// provera da li je 0 <= brojilac/imenilac <= 1
bool kolicnikUIntervalu01(int brojilac, int imenilac) {
```

```
  return brojilac == 0 ||
         (istogZnaka(brojilac, imenilac) && abs(brojilac) <= abs(imenilac));
}

// provera da li se intervali realne prave [a, b] i [c, d] seku
// (pri cemu se ne zna odnos vrednosti a i b tj. c i d)
bool projekcijeSeSeku(long long a, long long b, long long c, long long d) {
   return max(a, b) >= min(c, d) && max(c, d) >= min(a, b);
}

// provera da li se duzi AB i CD seku
bool duziSeSeku(const Tacka& A, const Tacka& B,
                const Tacka& C, const Tacka& D) {
   int ABCD = vektorski_proizvod_2d(A, B, C, D);
   if (ABCD != 0) {
      int ABCA = vektorski_proizvod_2d(A, B, C, A);
      int ACCD = vektorski_proizvod_2d(A, C, C, D);
      // presecna tacka se moze odrediti kao: C + (ABCA/ABCD)CD
      return kolicnikUIntervalu01(ACCD, ABCD) &&
             kolicnikUIntervalu01(ABCA, ABCD);
   } else {
     int ABAC = vektorski_proizvod_2d(A, B, A, C);
     if (ABAC != 0)
        return false;
     if (A.x == B.x && C.x == D.x)
        return projekcijeSeSeku(A.y, B.y, C.y, D.y);
     else
        return projekcijeSeSeku(A.x, B.x, C.x, D.x);
   }
}
```

Претходни програм је веома једноставно прерадити тако да израчунава и пресечну тачку дужи ако она постоји.

Треће могуће решење би било да за тачке A и B испитамо да ли припадају дужи CD, а за тачке C и D да ли припадају дужи AB (ако је било шта од овога задовољено, дужи се секу). Да би се испитало да ли тачка P припада дужи MN довољно је проверити да ли су тачке M, N и P колинеарне (тј. да ли је оријентација тројке MNP једнака нули) и да ли је x-координата тачке P између x-координата тачака M и N као и да ли је y-координата тачке P између y-координата тачака M и N.

5.1.7 Многоуглови

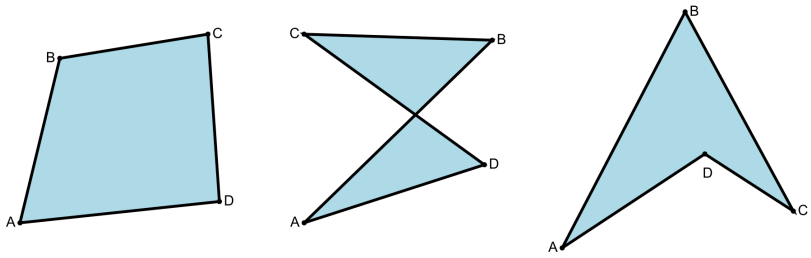
Објекти који се често разматрају у рачунарској геометрији су и многоуглови.

- *Пут* P је низ тачака $P_1, P_2, \ldots, P_k$ и дужи $P_1P_2, P_2P_3, \ldots, P_{k-1}P_k$ које их повезују. Дужи које чине пут су његове *странице* (*ивице*). Приметимо да је у општем случају допуштено да тачке на путу буду и колинеарне.

- *Затворени пут* је пут чија се последња тачка поклапа са првом и називамо га *многоугао* или *полигон*. Тачке које дефинишу многоугао су његова *темена*.

  На пример, троугао је многоугао са три темена. Многоугао се представља низом, а не скупом тачака, јер је битан редослед којим се тачке задају; променом редоследа тачака из истог скупа у општем случају добија се други многоугао.

- *Прост многоугао* је онај код кога одговарајући пут нема пресека са самим собом; другим речима, једине странице које имају заједничке тачке су суседне странице са својим заједничким теменом. Прост многоугао дели раван на две области: *унутрашњост* и *спољашност*. На слици 5.13, лево приказан је један прост многоугао, док је на средини приказан један многоугао који није прост (тј. многоугао који је самопресецајући).



Слика 5.13: Врсте многоуглова: прост, самопресецајући и неконвексан.

Отворен многоугао садржи само тачке у унутрашњости многоугла (без тачака на страницама). *Затворен* многоугао поред тачака у унутрашњости садржи и тачке на страницама многоугла.

- *Конвексни многоугао* је многоугао чија унутрашњост са сваке две тачке које садржи, садржи и све тачке дужи које те тачке одређују (еквивалентно, многоугао је конвексан ако су му сви унутрашњи углови мањи или једнаки $180°$ тј. π радијана). Многоугао је конвексан ако и само ако за произвољну праву која садржи неку његову страницу важи да су сва остала темена са исте стране те праве. *Конвексни пут* је пут састављен од тачака $P_1, P_2, \ldots, P_k$ такав да је многоугао $P_1P_2 \ldots P_k$

конвексан. Многоугао на слици 5.13 лево је конвексан, док је многоугао на тој слици десно неконвексан.

Природно се поставља питање испитивања да ли је многоугао задат низом својих тачака прост и да ли је конвексан.

5.1.7.1 Испитивање да ли је многоугао прост

Испитивање да ли је дати полигон прост грубом силом подразумева да се провери постојање пресека сваког пара страница. Проблем испитивања пресека дужи је описан у поглављу 5.1.6.3 и провера да ли се две дужи секу се може извршити у времену $O(1)$, међутим, многоугао са n темена има n страница и $\frac{n(n+1)}{2}$ парова страница, па је сложеност овог алгоритма $O(n^2)$.

Постоје и ефикаснији начини да се овај проблем реши. Бентли-Отманов[4] алгоритам проналази све пресеке који постоје у скупу од n дужи у времену $O((n+k)\log n)$, где је k број пресека који постоје (овај алгоритам има *излазно зависну сложеност*). Ако је број пресека k знатно мали (што је чест случај), сложеност је значајно боља од алгоритма грубе силе. Опис овог алгоритма превазилази оквире овог уџбеника. Ипак, у поглављу 5.2 описаћемо једноставнију варијанту у којој се зна да су дужи чији се пресеци траже хоризонталне или вертикалне. Да би се проверило да ли је многоугао прост, Бентли-Отманов алгоритам се примењује на скуп свих страница многоугла. Ако пресека нема, сложеност испитивања ће бити $O(n\log n)$, а ако постоје пресеци, алгоритам може да се заустави већ након налажења првог пресека, што опет гарантује сложеност $O(n\log n)$, чак и када пресека постоји пуно.

5.1.7.2 Испитивање да ли је многоугао конвексан

У простом конвексном многоуглу су све тројке тачака $P_iP_jP_k$ за $0 \leq i < j < k < n$ исто оријентисане. То је и довољан услов да би многоугао био прост и конвексан. Алгоритмом грубе силе би се израчунале и упоредиле све ове оријентације, што се може урадити у времену $O(n^3)$ (јер толико тројки постоји) и веома је неефикасно.

У конвексном многоуглу све тројке узастопних тачака имају исту оријентацију (на пример, када се тачке обилазе редом, до наредне тачке се увек долази правећи леви заокрет тј. заокрет у позитивном смеру), тј. све тројке $P_iP_{i+1}P_{i+2}$ за $0 \leq i < n$ и $P_n = P_0$, $P_{n+1} = P_1$ имају исту оријентацију. Овај услов није довољан да би многоугао био конвексан. На пример, пентаграм (звезда петокрака) задовољава овај услов, сви углови су оштри, али она није конвексан многоугао – није чак ни прост, јер се несуседне странице секу. Ипак, ако се унапред зна да је многоугао прост, овај услов је довољан и да би био конвексан и може се користити као алгоритам за проверу, чија је сложеност $O(n)$.

[4] Алгоритам су 1979. године описали Џон Бентли (енгл. Jon Bentley) и Томас Отман (енгл. Thomas Ottmann), амерички информатичари.

5.1.7.3 Површина простог многоугла

Слично као што је у поглављу 5.1.5 показано за троугао, и многоугао се може раставити на трапезе, тако да се означена површина простог многоугла може добити сабирањем означених површина појединачних трапеза. Трансформацијом тако добијене формуле добија се поново формула пертле којом се израчунава површина простог многоугла чија су темена тачке са координатама (x_1, y_1), (x_2, y_2), ..., (x_n, y_n).

$$P = \frac{1}{2} \sum_{i=1}^{n} (x_i y_{i+1} - x_{i+1} y_i)$$

При том се подразумева да је $(x_{n+1}, y_{n+1}) = (x_1, y_1)$.

Ако је многоугао конвексан, тада се његова површина може израчунати као збир површина троуглова добијених када се повуку све дијагонале из једног темена. Иако је и сложеност овог алгоритма линеарна, тај начин је компликованији и мало мање ефикасан него када се употреби формула пертле.

5.2 Пресеци хоризонталних и вертикалних дужи

Често се наилази на проблеме налажења пресека геометријских објеката. Некад је потребно пронаћи пресеке више објеката, а понекад само треба открити да ли је пресек непразан скуп. У наставку ћемо приказати један проблем налажења пресека, који илуструје важну технику за решавање геометријских проблема. Иста техника може се применити и на друге сличне проблеме.

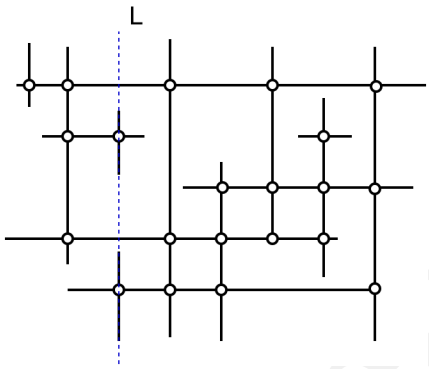
У претходном излагању фокус је био на томе да се одреди пресек дужи које се налазе у општем положају. Видели смо да тежина решења тог проблема лежи како у математичком апарату, тако у обради великог броја специјалних случајева. У наставку ћемо размотрити проблем у ком су дужи увек само хоризонталне и вертикалне, па је за две дужи веома једноставно испитати постојање пресека, међутим, тих дужи ће бити пуно и потребно је употребити геометријске особине (поредак координата) да би се смањио број испитивања пресека.

Проблем

За задати скуп од n хоризонталних и m вертикалних дужи пронаћи све њихове пресеке. Дужи су затворене (садрже и своје крајње тачке) и не постоје међусобни преседи хоризонталних ниши вертикалних дужи.

Овај проблем важан је, на пример, при пројектовању кола VLSI (интегрисаних кола са огромним бројем елемената). Коло може да садржи на хиљаде „жичица”, а пројек-

тант треба да буде сигуран да не постоје неочекивани пресеци. На проблем се такође наилази при елиминацији скривених линија када је потребно закључити које странице или делови страница су скривени самим објектом или неким другим објектом; тај проблем је обично компликованији, јер се не ради само о хоризонталним и вертикалним линијама. Пример улаза за овај проблем приказан је на слици 5.14.



Слика 5.14: Пресеци хоризонталних и вертикалних дужи.

Ако проблем решавамо додавањем једне по једне дужи (било хоризонталне, било вертикалне), онда је неопходно да се пронађу пресеци нове дужи са свим осталим дужима, па се добија алгоритам са $O(mn)$ налажења пресека дужи (што је заправо алгоритам грубе силе). У општем случају број пресека може да буде $O(mn)$, па алгоритам може да утроши време $O(mn)$ већ само за приказивање свих пресека. Међутим, број пресека може да буде много мањи од mn. Волели бисмо да конструишемо алгоритам који ради добро кад има мало пресека, а не превише лоше ако пресека има много. Дакле циљ нам је да проблем решимо коришћењем алгоритма чија сложеност зависи и од величине улаза и од величине излаза, тзв. алгоритма са *излазно зависном сложеношћу*.

Редослед обраде дужи може се одредити *бришућом правом* (енгл. sweeping line) која прелази („скенира”) раван слева удесно; дужи се разматрају оним редом којим на њих наилази покретна (бришућа) права. Значајни *догађаји* (енгл. events) су када права наиђе на неку карактеристичну тачку и алгоритам се извршава тако што се ти догађаји редом обрађују. Замислимо вертикалну праву која прелази раван слева удесно и обрадимо дужи у редоследу у којем права наилази на њих. Да бисмо имплементирали овај редослед, сортирамо све крајеве дужи према њиховим x-координатама. Две крајње тачке вертикалне дужи имају исте x-координате, па се региструје само једна x-координата. За хоризонталне дужи морају се користити оба краја. После сортирања крајева, дужи се разматрају једна по једна утврђеним редоследом.

Да ли је боље проверавати пресеке кад се обрађује хоризонтална или вертикална дуж?

Кад посматрамо било леви, било десни крај хоризонталне дужи, ми или још нисмо наишли на вертикалне дужи које је секу, или смо их већ обрадили. Дакле, боље је пресеке бројати приликом наиласка на вертикалну дуж. Претпоставимо да је покретна права тренутно преклопила вертикалну дуж L (видети слику 5.14).

Бришућа права (па и вертикалне дужи које она преклапа) се може сећи само са оним хоризонталним дужима чији се почетак налази лево, а крај десно од ње (пошто су дужи затворене, у разматрање треба узети и оне дужи чији се леви или десни крај поклапа са положајем бришуће праве). Дакле, бришућа права на позицији x се може сећи само са оним интервалима $[x_1, x_2]$ за које важи да је $x \in [x_1, x_2]$ тј. $x_1 \leq x$ и $x \leq x_2$. Стога ћемо ефикаснији алгоритам добити ако применимо одсецање и из разматрања елиминишемо све оне хоризонталне дужи које су се већ „завршиле" (за које је $x_2 < x$) и које још нису „почеле" (за које је $x < x_1$). За хоризонталне дужи које се још увек разматрају рећи ћемо да су *кандидати* и податке о њима ћемо чувати у некој погодној структури података. На слици 5.14 има четири кандидата за пресек са L.

Када се наиђе на вертикалну дуж L, претпоставићемо да знамо све кандидате које она сече. За налажење пресека са L, x-координате крајева кандидата нису од значаја, па је довољно чувати само њихове y-координате. Наиме, ми већ знамо да кандидати имају x-координате које „покривају" x-координату вертикалне дужи L, па је довољно проверити да ли су њихове y-координанте обухваћене опсегом y-координата дужи L. Зато ћемо одржавати структуру података која садржи само скуп y-координата кандидата. Одложићемо за тренутак анализу реализације ове структуре података.

Скуп кандидата (тј. њихових y-координата) можемо одржавати индуктивно: када наиђемо на леви крај хоризонталне дужи она се додаје у скуп кандидата, а када се наиђе на десни крај дужи, она се избацује из скупа кандидата.

На почетку је скуп кандидата празан и обрађујемо редом догађаје (у сортираном редоследу). Да би се исправно обрадили гранични случајеви (када се дужи додирују у једној тачки), претпоставићемо да ћемо догађаје са истом x-координатом поређати тако да прво иду леви крајеви хоризонталних дужи, затим вертикалне дужи и на крају десни крајеви хоризонталних дужи. На тај начин постижемо да су приликом обраде вертикалне дужи у скуп кандидата додате све дужи које почињу на њеној x-координати, а још нису уклоњене оне дужи које се завршавају на тој x-координати.

Постоје три могућности тј. три врсте догађаја које обрађујемо.

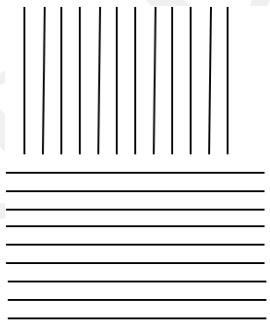
1. Наишли смо на леви крај хоризонталне дужи; дуж се тада додаје у скуп кандидата. Пошто десни крај дужи није достигнут, а дужи су затворене, она садржи текућу x-координату и заиста треба да буде део скупа кандидата.

2. Наишли смо на вертикалну дуж на координати x. Пресеци са овом вертикалном

дужи могу се пронаћи упоређивањем y-координата свих хоризонталних дужи из скупа кандидата са y-координатама крајева вертикалне дужи. У скупу кандидата се налазе све оне дужи које су започеле на координатама мањим или једнаким x, а нису се завршиле на координатама које су строго мање од x, па скуп кандидата заиста садржи тачно оне хоризонталне дужи које садрже координату x.

3. Наишли смо на десни крај хоризонталне дужи; дуж се тада просто елиминише из скупа кандидата. Као што је речено, пресеци се проналазе при разматрању вертикалних дужи, а у овом тренутку су обрађене све вертикалне дужи чија је x-координата мања или једнака текућој x-координати, па се ни један од пресека не губи елиминацијом ове хоризонталне дужи (она не може бити више кандидат за неки будући пресек).

Алгоритам је сада комплетан. Број упоређивања ће обично бити много мањи од mn, међутим, у најгорем случају овај алгоритам ипак захтева $O(mn)$ упоређивања, чак и кад је број пресека мали. Ако се, на пример, све хоризонталне дужи простиру слева удесно („целом ширином"), онда се мора проверити пресек сваке вертикалне дужи са свим хоризонталним дужима, што имплицира сложеност $O(mn)$. Овај најгори случај појављује се чак и ако ниједна вертикална дуж не сече ниједну хоризонталну дуж (видети пример на слици 5.15).



Слика 5.15: Случај када нема пресека дужи, а број поређења у оквиру основног алгоритма је $O(mn)$.

Да би се алгоритам усавршио, потребно је смањити број упоређивања y-координата вертикалне дужи са y-координатама хоризонталних дужи у скупу кандидата. Иако је полазни проблем дводимензионалан, налажење тих пресека је једнодимензионални проблем. Нека су y-координате вертикалне дужи која се тренутно разматра y_D и y_G, и нека су y-координате хоризонталних дужи из скупа кандидата $y_0, y_1, \ldots, y_{r-1}$. Ако претпоставимо да су хоризонталне дужи у скупу кандидата сортиране према y-координатама (односно низ $y_0, y_1, \ldots, y_{r-1}$ је растући), пресеци се могу пронаћи извођењем две би-

нарне претраге, једне за y_D, а друге за y_G. Нека је $y_i < y_D \leq y_{i+1} \leq y_j \leq y_G < y_{j+1}$. Вертикалну дуж секу хоризонталне дужи са координатама $y_{i+1}, y_{i+2}, \ldots, y_j$, и само оне, па је њихов број $j - i$. Ако је потребно да се наведу сви пресеци, тада се може извршити једна бинарна претрага за y_D, а затим пролазити y-координате, док се не дође до вредности y_{j+1} веће од y_G. Ако су бројеви сортирани, онда је време извршења пропорционално збиру времена тражења првог пресека и броја пронађених пресека. Наравно, не можемо да приуштимо себи сортирање хоризонталних дужи при сваком наиласку на вертикалну дуж, већ је потребно да дужи чувамо у некој погодној структури података.

Присетимо се још једном захтева. Потребна је структура података погодна за чување кандидата, која дозвољава уметање новог елемента, брисање елемента и ефикасно извршење бинарне претраге (тј. налажење првог елемента строго већег или већег или једнаког од дате вредности). На срећу, постоји више структура података које омогућују извршење уметања, брисања и тражења елемената сложености $O(\log n)$ по операцији (где је n број елемената у структури података), и линеарно претраживање за време пропорционално броју пронађених елемената — на пример, уређен скуп имплементиран помоћу уравнотеженог бинарног дрвета који је доступан у стандардној библиотеци већине савремених програмских језика (у језику C++ то је структура података `set`).

Имплементацију у језику C++ можемо направити на следећи начин.

```
// duz predstavljena koordinatama dve krajnje tacke
struct Duz {
  int x1, y1, x2, y2;

  Duz(int _x1, int _y1, int _x2, int _y2) {
    x1 = _x1; y1 = _y1;
    x2 = _x2; y2 = _y2;
  }

  // provera da li je duz horizontalna
  bool horizontalna() const {
    return y1 == y2;
  }
};

// ucitavamo podatke o n duzi (horizontalnih i vertikalnih)
int n;
cin >> n;
vector<Duz> duzi;
```

```
for (int i = 0; i < n; i++) {
  int x1, y1, x2, y2;
  cin >> x1 >> y1 >> x2 >> y2;
  duzi.emplace_back(x1, y1, x2, y2);
}

// brisuca prava obilazi duzi u neopadajucem redosledu x koordinata
// znacajni dogadjaji su kada brisuca prava naidje na levi kraj horizontalne
// duzi, desni kraj horizontalne duzi ili na vertikalnu duz
enum TipDogadjaja {LEVI_KRAJ, VERTIKALNA, DESNI_KRAJ};
// za svaki dogadaj pamtimo
struct Dogadjaj {
  int x;
  Duz duz;
  TipDogadjaja tip;

  Dogadjaj(int _x, const Duz& _duz, TipDogadjaja _tip)
    : x(_x), duz(_duz), tip(_tip) {
  }

  // dogadjaji se porede po x-koordinatama
  // dogadaje koji imaju istu x-koordinatu poredimo po tipu
  // (prvo idu levi krajevi, pa vertikalne duzi, pa desni krajevi)
  bool operator<(const Dogadjaj& d) {
    return x < d.x || (x == d.x && tip < d.tip);
  }
};

// popisujemo sve dogadjaje i sortiramo ih
vector<Dogadjaj> dogadjaji;
for (const Duz& duz : duzi) {
  if (duz.horizontalna()) {
    dogadjaji.emplace_back(duz.x1, duz, LEVI_KRAJ);
    dogadjaji.emplace_back(duz.x2, duz, DESNI_KRAJ);
  } else
    dogadjaji.emplace_back(duz.x1, duz, VERTIKALNA);
}
sort(begin(dogadjaji), end(dogadjaji));
```

```
// skup y koordinata horizontalnih duzi koje su aktivne
// (brisuca prava je presla njihov levi, ali ne i desni kraj)
set<int> yAktivnihHorizontalnih;

// obradjujemo sve dogadjaje
for (const Dogadjaj& dogadjaj : dogadjaji) {
  if (dogadjaj.tip == LEVI_KRAJ) {
    // brisuca prava je dosla do levog kraja horizontalne duzi, pa ona
    // postaje aktivna
    yAktivnihHorizontalnih.insert(dogadjaj.duz.y1);
  } else if (dogadjaj.tip == DESNI_KRAJ) {
    // brisuca prava je dosla do desnog kraja vertikalne duzi, pa ona
    // prestaje da bude aktivna
    yAktivnihHorizontalnih.erase(dogadjaj.duz.y1);
  } else {
    // brisuca prava je dosla do vertikalne duzi
    // ispisujemo sve preseke sa aktivnim horizontalnim duzima

    // najniza horizontalna duz koja je iznad donjeg temena
    // vertikalne duzi
    auto start = yAktivnihHorizontalnih.lower_bound(dogadjaj.duz.y1);

    // najniza horizontalna duz koja je strogo iznad gornjeg temena
    // vertikalne duzi
    auto end = yAktivnihHorizontalnih.upper_bound(dogadjaj.duz.y2);

    // sve duzi izmedju njih se seku
    for (auto it = start; it != end; it++)
      cout << dogadjaj.duz.x1 << " " << *it << endl;
  }
}
```

Као што је речено, Бентли-Отманов алгоритам представља уопштење овог алгоритма на произвољни скуп дужи. У том алгоритму, поред почетака и крајева дужи, значајан догађај је и када вертикална права наиђе на пресек дужи. У уређеном дрвету се чувају дужи сортиране по висини. Права може наићи само на пресек две суседне дужи и након што га прође, њихов међусобни редослед се мења. Пошто низ догађаја није статички (не може се у старту одредити, јер су догађаји и пресеци дужи који се одређују током рада алгоритма), догађаји се не чувају у сортираном низу, већ у реду са приоритетом. Детаљни опис и имплементација овог алгоритма неће бити дата у овом уџбенику.

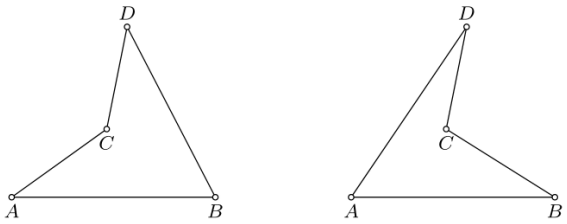
5.3 Конструкција простог многоугла

Скуп тачака у равни дефинише већи број различитих многоуглова, зависно од изабраног редоследа тачака. Размотрићемо сада проналажење простог многоугла са задатим скупом темена.

Проблем

Дато је n различитих тачака у равни, таквих да нису све колинеарне. Повезати их простим многоуглом.

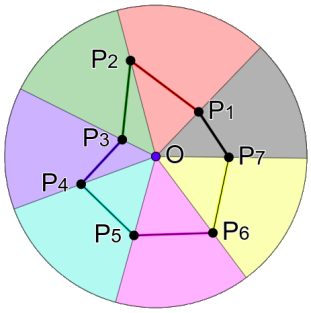
Постоји више метода за конструкцију траженог простог многоугла; уосталом, јасно је да у општем случају проблем нема једнозначно решење (слика 5.16).



Слика 5.16: Пример два различита проста многоугла са истим теменима.

Природна идеја је да се тачке обилазе некако у круг. Нека је дат неки круг са центром у тачки O, чија унутрашњост садржи све тачке. Површина тог круга се може „пребрисати" (прегледати) ротирајућом полуправом којој је почетак центар круга O и тачке се могу повезати у многоугао у редоследу у којем полуправа на њих наилази.

Очекујемо да ћемо спајањем тачака оним редом којим полуправа наилази на њих добити прост многоугао. Покушајмо да то докажемо. Претпоставимо за тренутак да ротирајућа полуправа у сваком тренутку садржи највише једну тачку. Означимо тачке, уређене у складу са редоследом наиласка полуправе на њих, са $P_1, P_2, \ldots, P_n$ (прва тачка бира се произвољно). Полуправе које спајају тачку O са овим тачкама деле круг на n међусобно дисјунктних кружних исечака. Темена сваке дужи P_iP_{i+1} припадају ивицама тих исечака. Ако су сви исечци конвексни (ако им је централни угао мањи од π), тада је то што темена дужи P_iP_{i+1} припадају исечку довољно да би могло да се гарантује да цела дуж припада том исечку. Дакле, ако би сви исечци били конвексни, добијени многоугао би морао да буде прост (што је и случај на слици 5.17). Наиме, исечци су сигурно међусобно дисјунктни, па ако им дужи целе припадају, и оне су међусобно дисјунктне (осим што се суседне додирују крајњим тачкама).

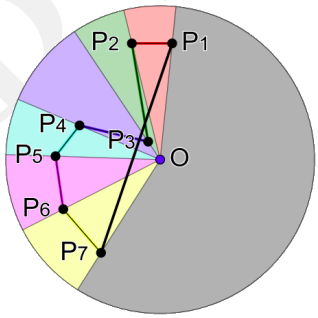


Слика 5.17: Прост многоугао у кругу.

Приметимо, међутим, да угао између полуправих кроз неке две узастопне тачке P_i и P_{i+1} може да буде већи од π. Тада исечак који садржи дуж P_iP_{i+1} садржи више од пола круга и није конвексна фигура, а дуж P_iP_{i+1} не припада том исечку, већ пролази кроз друге исечке круга, па може да сече друге странице многоугла.

Пример 5.3.1

У примеру илустрованом на слици 5.18 дуж P_1P_7 пролази кроз друге исечке круга и сече странице P_2P_3 и P_3P_4.



Слика 5.18: Лош избор центра круга који доводи до тога да конструисани многоугао није прост.

Да би се уочени проблем решио, круг је потребно одабрати тако да не постоји права која пролази кроз центар тако да се све тачке налазе са исте стране те праве. Једна природна идеја је да се тачка бира тако буде „на средини" датог скупа тачака. Могу се, на пример, фиксирати произвољне три тачке из скупа, а за центар круга изабрати нека тачка O унутар њима одређеног троугла (на пример тежиште, које се лако израчунава).

Овакав избор гарантује да ни један од добијених сектора круга неће имати угао већи од π. Заиста, полуправе из тежишта до темена троугла деле пун угао на три конвексна угла, а полуправе ка свим осталим тачкама полазног скупа само могу да те конвексне углове поделе на мање делове који такође морају бити конвексни.

Тачке сортирамо према положају у кругу са центром O. Прецизније, тачке P_i се сортирају на основу величине *угла* између полуправе из O која се простире у правцу позитивног дела x-осе и полуправе OP_i. Ако две или више тачака имају исти угао, оне се даље сортирају растуће (или опадајуће) према растојању од тачке O. На крају, тачке повезујемо у складу са добијеним уређењем, по две узастопне. Основна компонента временске сложености овог алгоритма потиче од сортирања тачака, те је сложеност алгоритма $O(n \log n)$.

Угао ϕ који права OP_i захвата са x осом једнак је вредности atan2$(P_{iy} - O_y, P_{ix} - O_x)$. Приметимо и да кад две тачке имају исти нагиб није неопходно рачунати њихова растојања од тачке O – довољно је израчунати квадрате растојања и нема потребе за израчунавањем квадратних коренова.

Претходни алгоритам се може имплементирати на следећи начин:

```
double ugao(double x0, double y0, double x, double y) {
  // ako su tacke t0=(x0, y0) i t=(x, y) racunamo
  // ugao koji vektor t0t zaklapa sa negativnim smerom x ose
  return atan2(y - y0, x - x0);
}

bool kolinearne(const Tacka& t1, const Tacka& t2, const Tacka& t3) {
  return abs(ugao(t1.x, t1.y, t2.x, t2.y) - ugao(t1.x, t1.y, t3.x, t3.y)) < EPS;
}

void prostMnogougao(vector<Tacka>& tacke) {
  // pronalazimo centar kruga kao teziste neke 3 nekolinearne tacke
  int i = 2;
  while (kolinearne(tacke[0], tacke[1], tacke[i]))
    i++;
  double x0 = (tacke[0].x + tacke[1].x + tacke[i].x) / 3.0;
  double y0 = (tacke[0].y + tacke[1].y + tacke[i].y) / 3.0;

  // sortiramo tacke na osnovu ugla
  sort(begin(tacke), end(tacke),
       [x0, y0](const Tacka& t1, const Tacka& t2) {
```

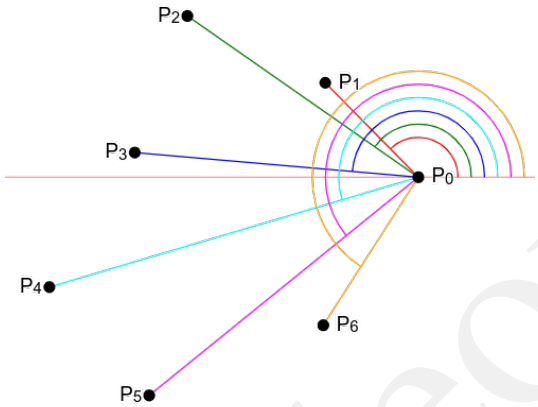
```
        // ugao koji vektor t0t1 zaklapa sa negativnim smerom x ose
        double ugao1 = ugao(x0, y0, t1.x, t1.y);
        // ugao koji vektor t0t2 zaklapa sa negativnim smerom x ose
        double ugao2 = ugao(x0, y0, t2.x, t2.y);

        if (ugao1 < ugao2 - EPS)
          return true;

        if (ugao2 < ugao1 - EPS)
          return false;

        // razlika uglova je u intervalu [-EPS, EPS],
        // pa tacke smatramo kolinearnim
        return kvadratRastojanja(x0, y0, t1.x, t1.y) <
               kvadratRastojanja(x0, y0, t2.x, t2.y);
      });
}
```

Уместо тежишта троугла одређеног са неке три неколинеарне тачке из скупа, за центар круга O може се узети и једна *екстремна тачка* тачка из скупа – на пример, тачка са највећом x-координатом (и са најмањом y-координатом, ако има више тачака са највећом x-координатом)[5]. Овакве тачке се често користе приликом решавања геометријских проблема. Екстремну тачку повезујемо са свим осталим тачкама датог скупа и тачке сортирамо растуће у односу на угао који полуправа од тачке O кроз ту тачку захвата са полуправом из темена O која је паралелна са x-осом (а тај је угао исти као угао који полуправа од тачке O кроз ту тачку захвата са позитивним смером x-осе, јер су то углови са паралелним крацима). Тачке означене у овом редоследу су приказане на слици 5.19.

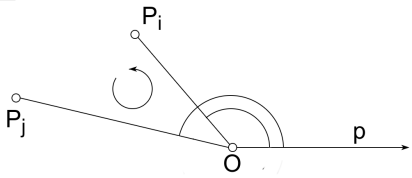
Пошто све тачке леже лево од тачке O, угао између полуправих кроз две узастопне тачке не може бити већи од π, те до дегенерисаног случаја о коме је било речи не може доћи. Ако две или више тачака захватају исти угао, оне се даље сортирају према растојању од тачке O и то на следећи начин: уколико првих неколико тачака захватају исти угао, њих сортирамо у растућем редоследу растојања од тачке O, уколико је последњих две или више тачака колинеарно са тачком O њих сортирамо у опадајућем редоследу растојања од тачке O, док је за остале тачке које су колинеарне са тачком O свеједно да ли ћемо их сортирати растуће или опадајуће.

Приметимо и то да је уместо рачунања углова могуће разматрати оријентацију тачака. Наиме, очигледно важи следећа лема (илустрована на слици 5.20).

[5] Наравно, пошто је проблем симетричан и дуж хоризонталне и дуж вертикалне осе, било која друга комбинација услова минималности и максималности координата је у реду.



Слика 5.19: Екстремна тачка $O = P_0$ и тачке сортиране према углу који права P_0P_i захвата са позитивним смером x-осе.



Слика 5.20: Илустрација леме 5.3.1.

Лема 5.3.1 **[Поредак углова и оријентација]**

Нека су P_i, P_j и O три неколинеарне тачке и нека се P_i и P_j налазе лево од O (тј. нека је x-координата тачке O већа или једнака од x-координата тачака P_i и P_j). Нека је p полуправа са теменом у O која се простире дуж позитивног смера x осе. Угао између полуправе p и полуправе OP_i је строго мањи од угла између полуправе p и полуправе OP_j ако и само ако је тројка P_jOP_i позитивно оријентисана (исто као и OP_iP_j).

Дакле, за потребе поређења тачака P_i и P_j према углу који полуправе OP_i и OP_j захватају са хоризонталном полуправом p кроз тачку O (тј. са позитивним смером x-осе) довољно је израчунати оријентацију тачака тројке OP_iP_j. Ако је она позитивна, онда OP_i заклапа мањи угао од OP_j, ако је негативна, онда OP_j заклапа мањи угао од OP_i, а ако је вредност оријентације нула, тада су те три тачке колинеарне, углови су једнаки и потребно је упоредити их по другим критеријумима (у нашем случају на основу растојања од O).

Претходни алгоритам се може имплементирати на следећи начин:

```
void prostMnogougao(vector<Tacka>& tacke) {
  // trazimo tacku sa maksimalnom x koordinatom,
  // u slucaju da ima vise tacaka sa maksimalnom x koordinatom
  // biramo onu sa najmanjom y koordinatom
  auto max = max_element(begin(tacke), end(tacke),
                         [](const Tacka& t1, const Tacka& t2) {
                           return t1.x < t2.x ||
                                  (t1.x == t2.x && t1.y > t2.y);
                         });
  // dovodimo je na početak niza - ona predstavlja centar kruga
  swap(*begin(tacke), *max);
  const Tacka& t0 = tacke[0];

  // sortiramo ostatak niza (tačke sortiramo na osnovu ugla koji
  // zaklapaju u odnosu vertikalnu polupravu koja polazi naviše iz
  // centra kruga), a kolinearne na osnovu rastojanja od centra kruga
  sort(next(begin(tacke)), end(tacke),
       [&t0](const Tacka& t1, const Tacka& t2) {
         Orijentacija o = orijentacija(t0, t1, t2);
         if (o == KOLINEARNE)
           return kvadratRastojanja(t0, t1) <= kvadratRastojanja(t0, t2);
```

```
        return o == POZITIVNA;
      });

  // obrcemo redosled tacaka na poslednjoj pravoj
  auto it = prev(end(tacke));
  while (orijentacija(*prev(it), *it, t0) == KOLINEARNE)
    it = prev(it);
  reverse(it, end(tacke));
}
```

Сложеност свих описаних алгоритама је $O(n \log n)$ и потиче од сортирања.

Иако оба приступа дају коректан резултат, визуелно допадљивији резултати се добијају првим приступом, тј. када се центар круга налази што ближе средишту многоугла.

5.4 Припадност тачке унутрашњости многоугла

У многим применама је потребно да се испита да ли тачка са датим координатама припада унутрашњости многоугла који ограничава неку област. На пример, у апликацијама са графичким корисничким интерфејсом или рачунарским играма је потребно одредити да ли се показивач миша налази унутар неког објекта, који је представљен многоуглом. Још једна примена је у склопу геокодирања. Задатак геокодирања јесте процес који за задато место или адресу враћа његове географске координате. Једнако важан задатак је и њему инверзан, којим се за дате координате одређује место – град, држава или нека област којој тачка са тим координатама припада.

Решење овог проблема се разликује у односу на то какав је многоугао који се анализира (да ли је прост, да ли је конвексан и слично).

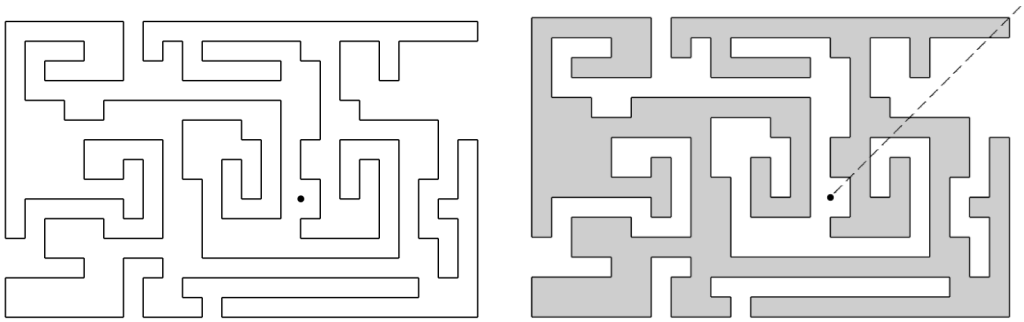
5.4.1 Испитивање припадности тачке унутрашњости простог многоугла

У поглављу 5.1.6.2 размотрили смо на који начин можемо утврдити да ли се тачка налази унутар датог троугла. Размотримо како се ово може уопштити на произвољни прост многоугао.

Проблем

Задат је прост многоугао $P_1 \ldots P_n$ и тачка Q, која не лежи на некој од његових страница. Установити да ли је тачка Q унутар тог многоугла.

Приметимо да смо, једноставности ради, изоставили питање провере да ли се тачка налази на страницама простог многоугла, па није релевантно да ли се ради о отвореном или затвореном многоуглу.



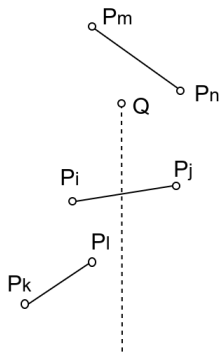
Слика 5.21: Утврђивање припадности тачке унутрашњости простог многоугла. На слици лево делује прилично нејасно да ли је тачка унутра или напољу. Тек када се унутрашњост многоугла обоји, постаје јасно да је тачка ван многоугла, што се може утврдити и бројањем пресека полуправе са страницама тог многоугла.

Проблем на први поглед изгледа једноставно, али ако се разматрају сложени неконвексни многоуглови, као онај приказан на слици 5.21, види се да то није увек случај. Интуитивни приступ је покушати некако „изаћи напоље", полазећи од задате тачке Q. Посматрајмо произвољну полуправу са теменом Q. Види се да је довољно пребројати пресеке са страницама многоугла. У примеру на слици 5.21, идући на североисток[6] од дате тачке (пратећи испрекидану линију), наилазимо на шест пресека са многоуглом. Пошто нас последњи пресек пре изласка изводи из унутрашњости многоугла, а претпоследњи нас враћа у његову унутрашњост, и тако даље, тачка Q је у спољашњости многоугла. Дакле, можемо закључити да тачка Q припада унутрашњости многоугла ако и само ако је број пресека полуправе из Q са страницама многоугла непаран.

Размотримо како се овај алгоритам може имплементирати ако је многоугао задат низом координата темена, што је уобичајени сценарио. Када се проблем решава визуелно, лако је наћи добар пут (полуправу са мало пресека) и није потребно разматрати странице многоугла које су далеко од тог пута. У случају многоугла датог низом координата, то није једноставно и највећи део времена троши се на испитивање постојања пресека полуправе и страница многоугла, јер нема једноставног начина да се избегне обрада свих страница. Испитивање пресека дужи (странице многоугла) и полуправе се може упростити ако је полуправа која садржи тачку Q паралелна једној од оса — на пример y-оси и рецимо усмерена надоле (као на слици 5.22). Означимо ту полуправу са q.

Наиме, пресек полуправе q са страницом P_iP_{i+1} многоугла постоји ако се x-координата тачке Q налази између вредности x-координата темена P_i и P_{i+1} и ако је y-координата

[6] Ради једноставнијег објашњења алгоритма, претпостављамо да се у Декартовом координатом систему зна шта је горе, доле, десно, лево, тј. где су север, југ, исток и запад.



Слика 5.22: Различити положаји страница многоугла и полуправе са теменом у тачки Q усмерене надоле: полуправа са почетком у тачки Q сече страницу P_iP_j, а не сече странице P_kP_l и страницу P_mP_n (иако се у сва три случаја праве секу, у случају странице P_kP_l та пресечна тачка је ван сегмента, а у случају странице P_mP_n та тачка је ван полуправе).

пресечне тачке полуправе q и праве P_iP_{i+1} мања од y-координате тачке Q. Претпоставимо, једноставности ради, да је $x_{P_i} < x_{P_{i+1}}$ (ако није, можемо разменити те две тачке). Пресечна тачка праве $x = x_Q$ и праве P_iP_{i+1} чија је једначина $y - y_{P_i} = \frac{y_{P_{i+1}} - y_{P_i}}{x_{P_{i+1}} - x_{P_i}}(x - x_{P_i})$ је тачка чија је x-координата x_Q, а y-координата је једнака:

$$y = y_{P_i} + \frac{y_{P_{i+1}} - y_{P_i}}{x_{P_{i+1}} - x_{P_i}}(x_Q - x_{P_i}).$$

Дакле, потребно је да важи $y \leq y_Q$, што се, када се израз трансформише да би се избегло дељење (именилац разломка је позитиван), своди на:

$$y_{P_i}(x_{P_{i+1}} - x_{P_i}) + (y_{P_{i+1}} - y_{P_i})(x_Q - x_{P_i}) \leq y_Q(x_{P_{i+1}} - x_{P_i})$$

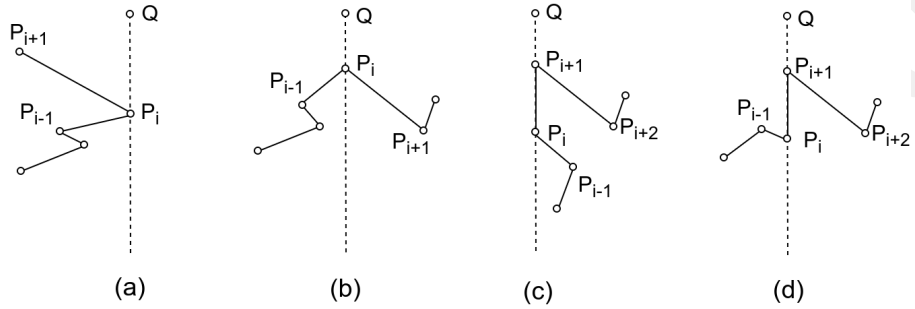
тј.

$$(y_{P_i} - y_Q)(x_{P_{i+1}} - x_{P_i}) \leq (y_{P_{i+1}} - y_{P_i})(x_{P_i} - x_Q)$$

До овог израза се може доћи и ако се провери оријентација тројке тачака $P_iP_{i+1}Q$ (та оријентација је позитивна ако и само ако се тачка Q налази изнад дужи P_iP_{i+1}, што је потребно да би пресек могао да постоји).

Као што је већ поменуто, обично постоје неки специјални случајеви које треба посебно размотрити. Циљ је утврдити да ли тачка Q припада унутрашњости многоугла P на

основу броја пресека полуправе q са страницама многоугла P. Размотримо пресек полуправе q и странице P_iP_{i+1}. Први специјални случај је када полуправа q садржи једну крајњу тачку те странице (тј. теме P_i или P_{i+1} многоугла), али не садржи ту целу страницу. Без губитка на општости претпоставимо да q садржи теме P_i. На слици 5.23 (a) приказан је случај кад тај пресек не треба бројати, а на слици 5.23 (b) случај кад тај пресек треба бројати.



Слика 5.23: Специјални случајеви кад вертикална полуправа q са почетком у тачки Q садржи неко теме или неку страницу многоугла.

Постоји више начина за утврђивање да ли неки овакав пресек треба бројати или не. Наведимо два.

- Ако су два темена многоугла суседна темену P_i са исте стране полуправе q (тј. праве која је садржи), пресек се не рачуна (оба суседна темена на слици 5.23 (a) су са леве стране полуправе q). У противном, ако су темена суседна темену P_i са различитих страна ове праве, пресек се броји (на 5.23 (b) једно суседно теме је са леве, а друго са десне полуправе q).
- С обзиром на то да разматрамо праву која је паралелна са y-осом, за сваку страницу многоугла требало би рачунати пресеке са левим теменом, а не са десним (темена неке странице многоугла класификујемо као лево и десно у односу на вредности x-координате темена). На тај начин нећемо рачунати пресек налик оном на слици 5.23 (a) јер је за обе странице многоугла суседне темену P_i пресек кроз десно теме (и слично два пута бисмо рачунали пресек са теменом које је леви екстремум и опет се не би променила парност бројача). За пресек приказан на слици 5.23 (b) убрајамо један пресек, јер је за једну од страница које се сустичу у том темену то лево теме, а за другу десно.

Други специјалан случај се јавља када полуправа q садржи страницу P_iP_{i+1}. На слици 5.23 (c) приказан је случај кад тај пресек не треба бројати, а на слици 5.23 (d) случај кад тај пресек треба бројати.

- Један начин да се одреди да ли пресек треба или не треба бројати је да се провери да ли су темена суседна страници P_iP_{i+1} (темена P_{i-1} и P_{i+2}) са исте или са разних страна полуправе q.

- У приступу у ком се за сваку страницу броје само пресеци са левим теменом, а не и са десним, пресеке са вертикалним дужима треба просто занемарити. Заиста, на слици 5.23 (c) би се тада рачунала два пресека са левим теменима дужи P_iP_{i-1} и $P_{i+1}P_{i+2}$, док би се на слици (d) рачунао само један пресек са левим теменом дужи $P_{i+1}P_{i+2}$.

У наставку ћемо размотрити имплементацију алгоритма за испитивање да ли је тачка у унутрашњости многоугла која не укључује обраду специјалних случајева.

```
// utvrdjuje da li se tacka Q nalazi u datom prostom mnogouglu P
// (on je zadat nizom svojih temena, kojih ima bar 3)
bool tackaUMnogouglu(const vector<Tacka>& P, const Tacka& Q) {
  // broj temena
  int n = P.size();
  // razmatramo preseke stranica sa vertikalnom pravom iz Q
  // tacka je inicijalno van mnogougla
  bool uMnogouglu = false;
  // prolazimo skupom svih stranica mnogougla
  for (int i = 0; i < n; i++) {
    // naredno teme u poretku temena mnogougla
    int j = (i + 1) % n;
    // obezbedjujemo da prva tacka ima manju x koordinatu
    Tacka Pi = P[i], Pj = P[j];
    if (Pi.x > Pj.x) swap(Pi, Pj);
    // ako se x koordinata tacke Q nalazi izmedju vrednosti
    // x koordinata krajnjih tacaka stranice PiPj i
    // y koordinata presecne tacka prave x = x_Q i prave PiPj
    // je manja od y koordinate tacke Q
    if (Pi.x < Q.x && Q.x < Pj.x &&
        (Pi.y-Q.y)*(Pj.x-Pi.x) <= (Pj.y-Pi.y)*(Pi.x-Q.x))
      // registrujemo jos jedan presek
      uMnogouglu = !uMnogouglu;
  }
  return uMnogouglu;
}
```

Нека је n број страница многоугла. Кроз основну петљу алгоритма пролази се n пута.

У сваком проласку кроз петљу рачуна се пресек две праве и извршавају се још неке операције за константно време. Дакле, укупно време извршавања алгоритма износи $O(n)$. Ако је познат гранични правоугаоник (енгл. bounding box) који садржи многоугао, тада се провера може убрзати тако што алгоритам се покреће само за оне тачке које припадају том правоугаонику (на тај начин понекад веома ефикасно можемо елиминисати велики број тачака). Алгоритам се може убрзати и тако што се избегне анализа неких страница многоугла, што се може учинити смештањем страница у специјалне структуре података које омогућавају брзу просторну претрагу, попут, на пример, R-дрвета (енгл. R-trees), чије изучавање препуштамо читаоцу.

5.4.2 Испитивање припадности тачке унутрашњости конвексног многоугла

Провера припадности тачке унутрашњости многоугла може бити много ефикаснија ако је многоугао конвексан.

Проблем

Испитати да ли тачка Q припада унутрашњости конвексног многоугла $P_0P_1 \dots P_{n-1}$.

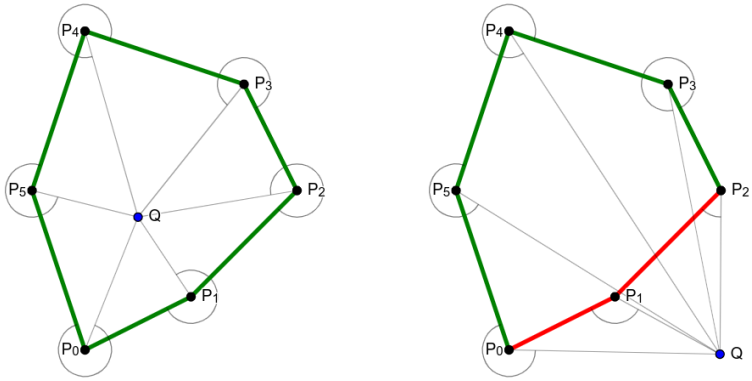
За почетак претпоставимо да тачка Q не припада страницама датог конвексног многоугла $P_0P_1 \dots P_{n-1}$.

5.4.2.1 Алгоритам заснован на оријентацији

Једно могуће решење је уопштење алгоритма за испитивање да ли тачка припада унутрашњости датог троугла. Оно није ефикасније него опште решење за прост многоугао, али је једноставније од њега. Једноставности ради, претпоставимо да су темена конвексног многоугла дата у математички позитивном смеру (смеру супротном од смера казаљке на часовнику). Тачка Q припада унутрашњости многоугла ако и само ако се налази с леве стране сваке усмерене странице P_iP_{i+1}, $0 \leq i \leq n-1$, $P_n = P_0$ многоугла, односно ако за свако i тројка тачака $P_iP_{i+1}Q$ има позитивну оријентацију. У примеру приказаном на слици 5.24 овај услов важи за тачку Q на слици лево, али не и за тачку Q на слици десно. Сложеност овог алгоритма износи $O(n)$.

Ако није позната оријентација темена конвексног многоугла (већ само њихов редослед) како би се утврдило да ли тачка Q припада унутрашњости многоугла могуће је проверити да ли свака тројка $P_iP_{i+1}Q$ има исту оријентацију (била она позитивна или негативна).

Ако је оријентација неке тројке $P_iP_{i+1}Q$ једнака нули, то значи да тачка Q припада правој P_iP_{i+1}, па пошто смо претпоставили да Q не припада страници P_iP_{i+1}, она сигурно припада спољашњости многоугла.



Слика 5.24: На слици лево тачка Q припада унутрашњости конвексног многоугла јер су све тројке тачака $P_iP_{i+1}Q$ позитивно оријентисане. На слици десно тачка Q не припада унутрашњости датог многоугла (тројке тачака P_0P_1Q и P_1P_2Q имају негативну оријентацију).

Можемо размотрити и општији проблем, у ком није унапред познато да тачка не припада страницама многоугла. Тада, ако је оријентација неке тројке $P_iP_{i+1}Q$ једнака нули, експлицитно треба проверити да ли тачка Q припада дужи P_iP_{i+1} (што се може урадити провером пројекција на осе, како је описано у поглављу 5.1.4.1) и резултат израчунати у складу са тим да ли разматрамо отворен или затворен многоугао (тј. да ли поред унутрашњости допуштамо и припадност тачке страницама многоугла).

5.4.2.2 Алгоритам заснован на бинарној претрази

Идеја на којој се заснива ефикаснији алгоритам се ослања на бинарну претрагу. Једноставније је имплементирати проверу да ли тачка припада затвореном многоуглу.

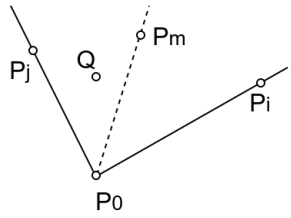
Лема 5.4.1

Нека се тачке Q и P_m налазе унутар угла $\angle P_iP_0P_j$ (слика 5.25), при чему тројка тачака $P_0P_iP_j$ има позитивну оријентацију. Тада важи:

- *ако су тачке P_i и Q са исте стране праве P_0P_m, тј. ако су тројке $P_0P_mP_i$ и P_0P_mQ исто оријентисане, тада тачка Q припада углу $\angle P_iP_0P_m$ (тј. његовој унутрашњости). Овај услов своди се на то да је тројка тачака P_0P_mQ негативно оријентисана.*
- *ако су тачке P_j и Q са исте стране праве P_0P_m (као на слици 5.25), тј. ако су тројке $P_0P_mP_j$ и P_0P_mQ исто оријентисане, тада тачка Q припада углу $\angle P_mP_0P_j$ (тј. његовој унутрашњости). Овај услов своди се на то да је тројка*

тачака P_0P_mQ позитивно оријентисана (што је, наравно, услов супротан оном у претходном случају).

- *ако тачка Q баш припада правој P_0P_m, усвајамо договор да тачка Q припада углу $\angle P_iP_0P_m$.*



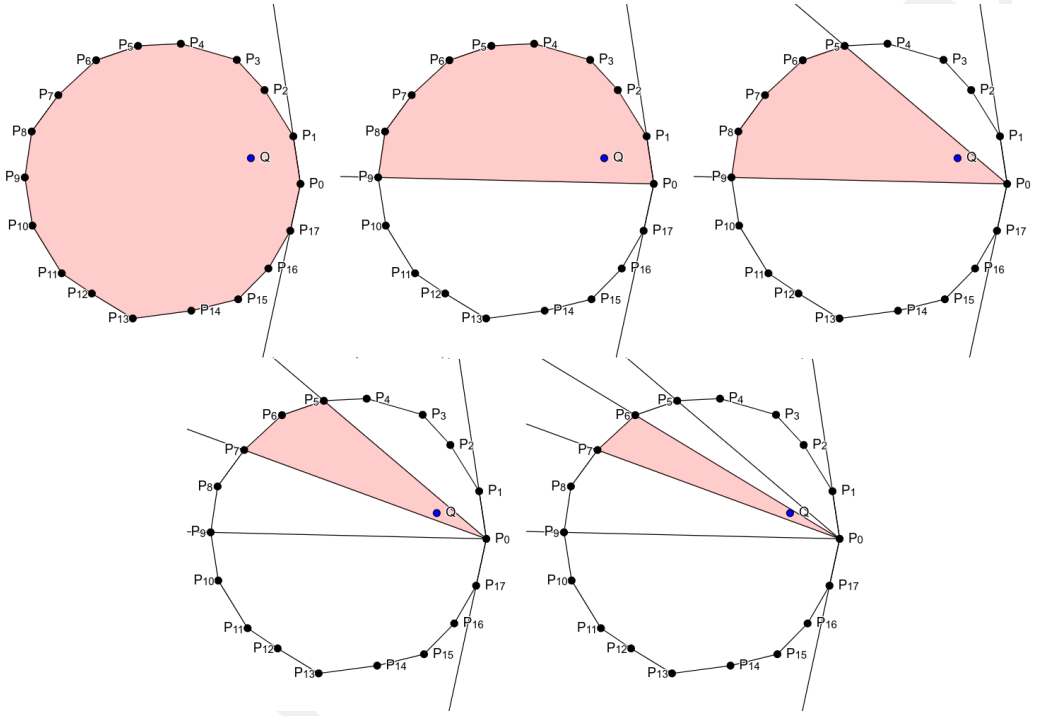
Слика 5.25: Свођење проблема испитивања припадности тачке неком углу на припадност неком његовом делу.

Искористићемо претходну лему да бинарном претрагом пронађемо вредност i такву да тачка Q припада затвореном углу $P_iP_0P_{i+1}$. Одржаваћемо текући угао $\angle P_iP_0P_j$ коме припада тачка Q, који иницијализујемо на $\angle P_1P_0P_{n-1}$. У сваком кораку за тачку P_m бирамо средишњу тачку – тачку са индексом $m = \lfloor (i+j)/2 \rfloor$ и ажурирамо угао коме припада тачка Q према претходној леми. Заустављамо се када тачке P_i и P_j постану суседна темена многоугла, тј. када је $j = i+1$. После највише $O(\log n)$ корака проналази се угао $\angle P_iP_0P_{i+1}$ коме тачка Q мора припадати ако припада затвореном многоуглу. Тада знамо да тачка Q припада затвореном многоуглу ако и само ако припада затвореном троуглу $\triangle P_0P_iP_{i+1}$. Сложеност је, дакле $O(\log n)$. На слици 5.26 приказан је пример поступка половљења угла коме припада тачка Q.

На почетку је могуће проверити да ли тачка припада углу $\angle P_1P_0P_{n-1}$ и ако не припада одмах се може закључити да тачка не припада ни многоуглу. Међутим, та провера није неопходна, јер ако тачка не припада том углу бинарном претрагом ће се доћи или до троугла $\triangle P_0P_1P_2$ или $\triangle P_0P_{n-2}P_{n-1}$ и тада ће се закључити да му тачка не припада, па не припада ни затвореном многоуглу.

Овај алгоритам се може веома једноставно имплементирати.

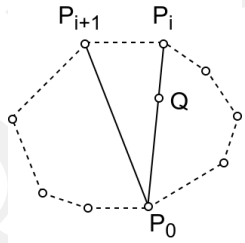
```
// niz sadrzi redom tacke mnogougla P, funkcija proverava da li se tacka Q
// nalazi unutar preseka zatvorenog mnogougla P i ugla P[l]P[0]P[d]
bool tackaUZatvorenomKonveksnomMnogouglu(const vector<Tacka>& P,
                                         const Tacka& Q, int i, int j) {
  if (j - i == 1)
    return tackaUZatvorenomTrouglu(Q, P[0], P[i], P[j]);
```



Слика 5.26: Испитивање припадности тачке Q конвексном многоуглу методом половљења угла. На крају је потребно испитати да ли тачка Q припада троуглу $P_0P_6P_7$.

```
  int m = i + (j - i) / 2;
  if (orijentacija(P[0], P[m], Q) == POZITIVNA)
    return tackaUZatvorenomKonveksnomMnogouglu(P, Q, m, j);
  else
    return tackaUZatvorenomKonveksnomMnogouglu(P, Q, i, m);
}

// provera da li dati zatvoreni konveksni mnogougao P sadrzi tacku Q
bool tackaUZatvorenomKonveksnomMnogouglu(const vector<Tacka>& P,
                                         const Tacka& Q) {
  int n = P.size();
  return tackaUZatvorenomKonveksnomMnogouglu(P, Q, 1, n-1);
}
```

Ако разматрамо отворени многоугао, у првој фази на исти начин проналазимо угао $P_iP_0P_{i+1}$ коме тачка Q припада. Тачка Q припада отвореном многоуглу ако и само ако припада унутрашњости троугла $P_iP_0P_{i+1}$ или припада дужи P_0P_i (када је $i \neq 1$) или припада дужи P_0P_{i+1} (када је $i \neq n-2$).

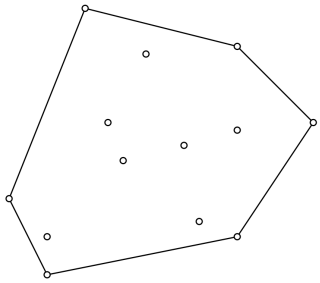


Слика 5.27: Иако се испитује припадност отвореном многоуглу, када се бинарном претрагом открије угао $P_iP_0P_{i+1}$ коме припада тачка Q, потребно је проверити да ли тачка припада троуглу $P_iP_0P_{i+1}$ који је полуотворен – мора се укључити и провера припадности страницама P_0P_i и P_0P_{i+1}, осим у случају када су оне странице многоугла.

5.5 Конструкција конвексног омотача

Конвексни омотач (енгл. convex hull) коначног скупа тачака дефинише се као најмањи конвексни многоугао такав да све тачке тог скупа припадају унутрашњости или страницама тог многоугла (слика 5.28). Дефинисаћемо и да је за једну тачку конвексни омотач та тачка, а за пар тачака дуж која их спаја. Јасно је из дефиниције да је конвексни омотач три неколинеарне тачке троугао коме су те тачке темена. Забранићемо да конвексни омотач садржи колинеарне тачке и тада за сваки скуп тачака постоји једин-

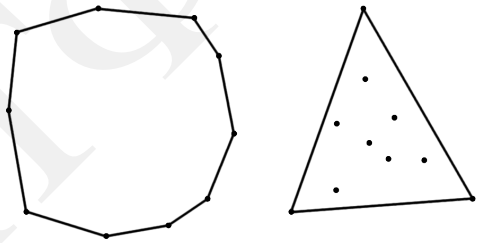
ствен конвексни омотач. Конвексни омотач се представља на исти начин као било који многоугао — низом својих узастопних темена.



Слика 5.28: Пример скупа тачака и његовог конвексног омотача.

Као што смо већ видели, обрада конвексних многоуглова једноставнија је од обраде произвољних многоуглова. На пример, провера припадности тачке унутрашњости произвољног простог n-тоугла је сложености $O(n)$, док је испитивање припадности тачке унутрашњости конвексног n-тоугла сложености $O(\log n)$.

Из услова да је конвексни омотач најмањи конвексни многоугао са датим својствима, следи да су темена конвексног омотача неке од тачака из задатог скупа. Казаћемо да тачка *припада* омотачу ако је теме омотача. Ако занемаримо тривијалне случајеве када полазни скуп не садржи бар три неколинеарне тачке, конвексни омотач може се састојати од најмање три, а највише n тачака (слика 5.29).



Слика 5.29: Конвексни омотач који садржи све тачке скупа (лево) и конвексни омотач скупа који садржи само три тачке (десно).

Конвексни омотачи имају широку примену (на пример, у праћењу ширења епидемија, моделовању глатких кривих, смањењу броја боја на слици, планирању кретања робота, као градивни елемент у другим алгоритмима, као што је рецимо тражење дијаметра скупа тачака), па су због тога развијени многобројни алгоритми за њихову конструкцију. Размотримо проблем навигације робота у равни од тачке A до тачке B. Најједностав-

нији начин да робот реализује ово кретање је праволинијски, дуж дужи AB. Међутим, на овој праволинијској путањи се може наћи нека од препрека. Стога се свака од препрека може узорковати тачком у равни, а онда направити конвексни омотач свих ових тачака. За путању која не сече овај омотач се гарантује да неће бити судара робота са неком од препрека.

У наставку ћемо размотрити неколико различитих решења наредног проблема.

Проблем

Конструисати конвексни омотач задатих n тачака у равни.

5.5.1 Директни индуктивни приступ

Као и обично, покушаћемо најпре са директним индуктивним приступом. Конвексни омотач за једну тачку је једночлан и то може бити база индукције. Претпоставимо следећу индуктивну хипотезу.

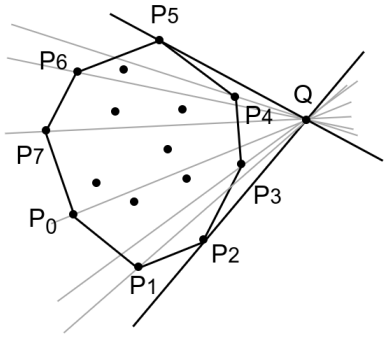
Индуктивна хипотеза. Умемо да конструишемо конвексни омотач скупа који садржи мање од n тачака.

Покушајмо да конструишемо конвексни омотач скупа од n тачака. Поставља се питање на који начин n-та тачка може да промени конвексни омотач првих $n-1$ тачака. Постоје два могућа случаја:

- нова тачка је у претходном конвексном омотачу, па он остаје непромењен,
- нова тачка је ван претходног конвексног омотача, па се омотач „шири” да обухвати и нову тачку (слика 5.30).

Приликом додавања сваке нове тачке потребно је, дакле, решити два потпроблема: утврђивање да ли је нова тачка унутар омотача и проширивање омотача новом тачком.

Ствар се може упростити погодним избором n-те тачке, тако да не морамо експлицитно вршити проверу да ли она припада претходном омотачу. Звучи примамљиво покушати да увек бирамо тачку унутар омотача, јер тада не морамо да мењамо омотач да би се проширила индуктивна хипотеза; то, међутим, није увек могуће јер у неким случајевима све тачке полазног скупа припадају конвексном омотачу (5.29, лево). Сасвим супротна могућност је да се у сваком кораку бира тачка за коју знамо да не припада унутрашњости текућег омотача. То ће бити задовољено ако се у сваком кораку бира нека екстремна тачка новог многоугла, при чему нам тај избор, као што ћемо видети, додатно упрошћава поступак ажурирања конвексног омотача. Подсетимо се, екстремне тачке су се показале успешне при решавању проблема конструкције простог многоугла. Екстремна тачка која се у том проблему користила је она са максималном x-координатом (при чему се ако има више таквих тачака бира она са минималном y-координатом). Да би се



Слика 5.30: Проширивање конвексног омотача новом тачком у случају када нова тачка не припада претходном омотачу.

у сваком кораку користила екстремна тачка, потребно је тачке обрађивати у неопадајућем редоследу x-координата (при чему тачке са истом x-координатом треба обрађивати у нерастућем редоследу y-координата), што се најлакше постиже ако се у старту тачке сортирају на тај начин. Тада заправо користимо претходну индуктивну хипотезу, али тако да је увек примењујемо на скуп претходних тачака у сортираном редоследу. Нека је текућа тачка којом се проширује конвексни омотач конструисан од почетних $n-1$ тачака тачка Q, која је екстремна у текућем скупу од n тачака. Тачка Q мора бити теме новог конвексног омотача. Питање је како променити конвексни омотач осталих $n-1$ тачака тако да нови омотач обухвати тачку Q. Потребно је најпре пронаћи темена старог омотача која су у унутрашњости новог омотача (P_3 и P_4 у примеру на слици 5.30) и уклонити их, а затим је потребно уметнути ново теме Q између два постојећа (P_2 и P_5 на слици 5.30).

Одређивање тачака текућег омотача које треба да буду замењене тачком Q може се извршити коришћењем правих ослонца. *Права ослонца* (енгл. supporting plane) конвексног многоугла је права која садржи бар једну тачку многоугла и све остале тачке многоугла се налазе са исте стране те праве. У нашем случају су битне праве ослонца које садрже тачку Q. На пример, на слици 5.30 праве QP_2 и QP_5 су две праве ослонца и полазног и проширеног конвексног многоугла. Обично само једно теме многоугла повезивањем са Q одређује праву ослонца (постоји и специјални случај кад више темена многоугла лежи на прави ослонца, који ћемо на тренутак занемарити). Многоугао лежи између две праве ослонца, што указује на то на који начин треба извести модификацију омотача. Праве ослонца захватају минимални и максимални угао са хоризонталном полуправом p која се из тачке Q простире у позитивном смеру x осе, међу свим правама које садрже тачку Q и неко теме многоугла. Да бисмо одредили та два темена, потребно је да размотримо полуправе из тачке Q ка свим теменима, да израчунамо углове са p, и међу

тим угловима изаберемо минималан (чиме добијамо тачку P_i) и максималан (чиме добијамо тачку P_j). Као што је показано лемом 5.3.1, уместо експлицитног рачунања углова, могуће их је само упоређивати на основу оријентације (угао који полуправа QP_m захвата са полуправом p мањи је од угла који полуправа QP_n захвата са p ако и само ако је тројка QP_mP_n позитивно оријентисана). У специјалном случају се може десити да су неке тачке P_m и P_n колинеарне са тачком Q и тада узимамо ону од њих која се налази даље од Q (јер претпостављамо да је конвексни омотач затворен многоугао и да пошто ближа тачка припада страници, припада и омотачу).

После проналажења два екстремна темена P_i и P_j модификовани омотач добијамо тако што онај низ темена између P_i и P_j (без P_i и P_j) чије су тачке са исте стране праве P_iP_j као и тачка Q заменимо тачком Q (ако је тај низ празан, само се умеће нова тачка Q).

Пример 5.5.1

У примеру приказаном на слици 5.30 темена P_3 и P_4 се налазе се са исте стране праве P_5P_2 као и тачка Q, па се низ темена старог омотача $[P_3, P_4]$ замењује новим теменом Q. Тачке P_0 и P_1 су са супротне стране праве P_5P_2 од тачке Q, па оне остају део конвексног омотача.

Да би се реализовао претходни алгоритам, није неопходно експлицитно испитивати који од два низа тачака између P_i и P_j лежи са исте стране праве P_iP_j као и Q. Наиме, ако сва темена тренутног конвексног омотача чувамо у редоследу позитивне оријентације, пошто је тачка Q екстремна и налази се десно од праве P_iP_j, тачке између P_j и P_i ће бити са исте стране праве P_iP_j као и Q, док ће тачке између P_i и P_j бити са супротне стране. Дакле, увек је потребно низ тачака од P_{j+1} до P_{i-1} заменити тачком Q (при чему се увећање и умањење индекса рачуна циклично, тј. по модулу n). Ако су тачке P_j и P_i суседне, тај низ може бити и празан.

У наставку је приказана C++ имплементација овог алгоритма.

```
vector<Tacka> konveksniOmotac(vector<Tacka>& tacke) {
  // sortiramo tacke po x koordinati neopadajuce,
  // ako postoje dve tacke sa istom vrednoscu x koordinate
  // manjom smatramo onu sa vecom y koordinatom
  sort(begin(tacke), end(tacke),
       [](const Tacka &p1, const Tacka &p2) {
         return (p1.x < p2.x) || (p1.x == p2.x && p1.y > p2.y);
       });

  // u konveksni omotac dodajemo prvu tacku
```

```
  vector<Tacka> omotac;
  omotac.push_back(tacke[0]);

  // dodajemo jednu po jednu tacku u omotac u redosledu rastucih x koordinata
  for (int k = 1; k < tacke.size(); k++) {
    // trazimo gornju pravu oslonca
    // to je prava P_kP_i tako da za svako ii vazi
    // da trojka tacaka {P_k, P_i, P_ii} ima pozitivnu orijentaciju ili su
    // tacke kolinearne, ali je P_ii dalje od tacke P_k nego P_i
    int i = 0;
    for (int ii = 0; ii < omotac.size(); ii++) {
      int o = orijentacija(tacke[k], omotac[i], omotac[ii]);
      if (o == NEGATIVNA ||
          (o == KOLINEARNE && kvadratRastojanja(tacke[k], omotac[i]) <
                              kvadratRastojanja(tacke[k], omotac[ii])))
        i = ii;
    }

    // trazimo donju pravu oslonca
    // to je prava P_kP_j tako da za svako jj vazi
    // da trojka tacaka {P_k, P_jj, P_j} ima pozitivnu orijentaciju ili su
    // tacke kolinearne, ali je P_jj dalje od tacke P_k nego P_j
    int j = 0;
    for (int jj = 0; jj < omotac.size(); jj++) {
      int o = orijentacija(tacke[k], omotac[jj], omotac[j]);
      if (o == NEGATIVNA ||
          (o == KOLINEARNE && kvadratRastojanja(tacke[k], omotac[j]) <
                              kvadratRastojanja(tacke[k], omotac[jj])))
        j = jj;
    }

    zameni(omotac, (j + 1) % omotac.size(), i, tacke[k]);
  }
  return omotac;
}

// podniz niza a_i,...,a_{j-1} menjamo elementom x
// ako je i > j, brisu se elementi do kraja, pa zatim sa pocetka niza
void zameni(vector<Tacka>& a, int i, int j, const Tacka& x) {
```

```
    if (i < j)
      // ako je i < j, onda su tacke koje brisemo susedne
      a.erase(a.begin() + i, a.begin() + j);
    else if (i > j) {
      // ako je i > j, onda elemente brisemo iz dva dela
      a.erase(a.begin() + i, a.end());
      a.erase(a.begin(), a.begin() + j);
    }
    // ako je i = j, onda se ne brise nista

    // ubacujemo novi broj x
    a.insert(a.begin() + i, x);
}
```

Приметимо да се у функцији сортира дати низ тачака, што значи да аргумент функције није константан. Ако бисмо инсистирали да се низ не сме мењати током израчунавања омотача (што је корисна пракса), морали бисмо направити сортирану копију низа унутар функције, чиме би се добио мало неефикаснији алгоритам. Одређивање правих ослонца се суштински спроводи алгоритмом одређивања позиције максималног тј. минималног елемента, па га је могуће имплементирати и библиотечким функцијама `max_element` или `min_element`.

Размотримо сложеност овог алгоритма. Прво се сортира низ темена, за шта је потребно време $O(n \log n)$. За сваку нову тачку која се додаје скупу треба израчунати углове које граде праве одређене тачком Q и сваким од темена претходног омотача и x-оса, пронаћи међу њима минимални и максимални угао, избацити нека темена из претходног омотача и додати ново теме. Дакле, сложеност додавања k-те тачке у текући скуп тачака је $O(k)$, а укупно разматрамо n тачака па се сложеност овог дела алгоритма може описати рекурентном једначином $T(n) = T(n-1) + O(n)$ чије је решење $O(n^2)$. Пошто је време сортирања $O(n \log n)$ асимптотски мање од времена $O(n^2)$ потребног за остале операције, оно не утиче на укупну сложеност $O(n^2)$. Нагласимо и да избор структуре података за чување тренутног низа темена конвексног многоугла (листе или вектора) не утиче на асимптотску сложеност (сложеност функције `zameni` би могла бити снижена када би се користила листа, али би асимптотска сложеност остала квадратна).

5.5.2 Увијање поклона

На који начин се описани алгоритам може побољшати? Кад проширујемо многоугао теме по теме, доста времена трошимо на формирање конвексних омотача од тачака за које ће се касније можда показати да су унутрашње за коначни конвексни омотач. Може ли се то избећи? Уместо да правимо конвексне омотаче подскупова датог скупа

тачака, можемо да посматрамо комплетан скуп тачака и да директно правимо његов конвексни омотач. Може се, као и у претходном алгоритму, кренути од екстремне тачке. У принципу, свеједно је која се екстремна тачка користи. Да бисмо илустровали мало другачији приступ од досадашњих решења, можемо, на пример, кренути од тачке са најмањом вредношћу x координате, а ако их има више оне међу њима која има најмању y координату. Као што смо већ поменули, екстремна тачка увек припада конвексном омотачу. Идеја је да се у сваком кораку у омотач дода наредно теме које је део коначног конвексног омотача (при чему темена обилазимо, на пример, у негативном математичком смеру). Како налазимо наредну страницу конвексног омотача? Једно теме те странице биће последња тачка до сада одређеног дела конвексног омотача. Да бисмо одредили друго теме странице, анализираћемо све тачке и тражити ону која даје дуж која са претходном дужи гради што већи угао. Приметимо да приликом одређивања прве странице многоугла не постоји претходна дуж, те у том случају тражимо дуж која гради максимални угао са нпр. негативним смером y осе.

Нека је P_t текућа тачка у конвексном омотачу. Обрађиваћемо преостале тачке и сушински примењивати алгоритам одређивања максимума. За почетну вредност ћемо узети произвољну тачку P_l различиту од P_t. Затим ћемо редом обрађивати све остале тачке. Аналогно леми 5.3.1, уместо рачунања углова можемо употребити оријентацију.

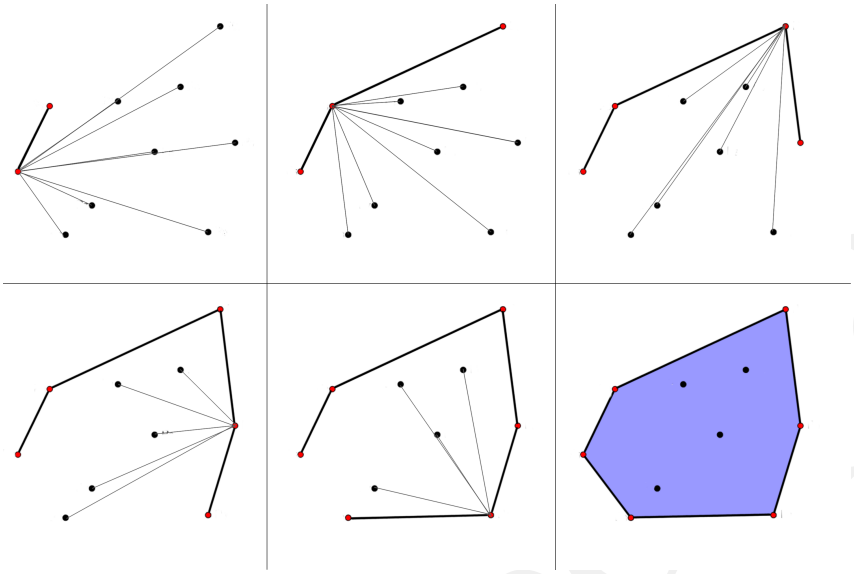
- Ако је оријентација тројке тачака $P_tP_lP_i$ негативна, тада је угао дужи P_tP_l већи од угла P_tP_i и P_l остаје кандидат за теме дужи са максималним углом.
- Ако је оријентација тројке тачака $P_tP_lP_i$ позитивна, тада је угао дужи P_tP_i већи од угла P_tP_l и P_i постаје нови кандидат за теме дужи са максималним углом.
- Ако је оријентација тројке тачака $P_tP_lP_i$ једнака нули, три тачке су колинеарне. Пошто сматрамо да је конвексни омотач затворен многоугао, желимо да узмемо ону тачку која је даља од P_t, тако да се текући кандидат мења ако је P_l између P_t и P_i.

Поступак се завршава када се установи да је тачка која одређује највећи угао са претходном тачком једнака почетној тачки. Кораци у извршавању овог алгоритма за један скуп тачака су приказани на слици 5.31.

Овај алгоритам се из разумљивих разлога зове *увијање поклона* (енгл. gift wrapping algorithm). Полази се од једног темена „поклона”, и онда се он увија у конвексни омотач проналажењем темена по темена омотача. Познат је и под називом *Џарвисов*[7] *марш* (енгл. Jarvis March algorithm), по научнику који га је описао у раду 1973. године.

Алгоритам увијања поклона је директна последица примене следеће индуктивне хипотезе (по k):

[7] Рејмонд Џарвис (енгл. R.A.Jarvis), аустралијски информатичар.



Слика 5.31: Кораци у извршавању алгоритма увијања поклона.

Индуктивна хипотеза. За задати скуп од n тачака у равни, умемо да пронађемо конвексни пут дужине $k < n$ који је део коначног конвексног омотача датог скупа тачака.

Код ове хипотезе нагласак је на проширивању *пута*, а не омотача. Уместо да проналазимо конвексне омотаче подскупова, ми проналазимо део коначног конвексног омотача.

Алгоритам увијања поклона је могуће имплементирати на следећи начин:

```
vector<Tacka> konveksniOmotac(vector<Tacka>& tacke) {
  // odredjujemo redni broj ekstremne tacke (one sa najmanjom x koordinatom)
  auto cmp = [] (const Tacka& t1, const Tacka& t2) {
    return t1.x < t2.x || (t1.x == t2.x && t1.y < t2.y);
  };
  int prva = distance(begin(tacke),
                      min_element(begin(tacke), end(tacke), cmp));

  // niz tacaka koje cine omotac
  vector<Tacka> omotac;
  // krecemo od ekstremne tacke
  int tekuca = prva;
```

```
  do {
    omotac.push_back(tacke[tekuca]);
    // krecemo obradu od prve tacke u nizu, osim ako je prva tacka
    // bas ekstremna; u tom slucaju krecemo od druge tacke
    int naredna = tekuca == 0 ? 1 : 0;
    // odredjujemo narednu tacku u omotacu, zahtevajuci da se sve
    // ostale tacke nalaze sa desne strane prave odredjene tekucom i
    // tom narednom tackom
    for (size_t i = 0; i < tacke.size(); i++) {
      // razmatramo sve tacke osim onih kojima je odredjena poslednja duz
      if (i == tekuca || i == naredna)
        continue;
      Orijentacija o = orijentacija(tacke[tekuca], tacke[naredna], tacke[i]);
      if (o == POZITIVNA ||
          (o == KOLINEARNE &&
           izmedju(tacke[tekuca], tacke[naredna], tacke[i])))
        naredna = i;
    }
    tekuca = naredna;
  } while (tekuca != prva);
  return omotac;
}
```

Ако је m број темена коначног конвексног омотача, временска сложеност увијања поклона је $O(mn)$. Дакле, овај алгоритам спада у групу алгоритама чија сложеност зависи не само од димензије улаза, већ и од димензије излаза, тј. алгоритама са излазно зависном сложеношћу. У случају када је конвексни омотач скупа тачака троугао (слика 5.29 десно), алгоритам увијања поклона има сложеност $O(n)$, а у случају када све тачке (или у општем случају $O(n)$ тачака) припадају конвексном омотачу (слика 5.29 лево) сложеност алгоритма је $O(n^2)$.

5.5.3 Грејемов алгоритам

У наставку ћемо размотрити ефикасније алгоритме за налажење конвексног омотача.

Кренимо од *Грејемовог*[8] *алгоритма*. Започиње се сортирањем тачака према угловима, слично као при конструкцији простог многоугла. Нека је P_0 екстремна тачка, рецимо она са највећом x-координатом (и са најмањом y-координатом, ако има више тачака са највећом x-координатом). За сваку тачку P_i из датог скупа тачака израчунавамо угао између полуправе P_0P_i и хоризонталне полуправе из P_0 која се простире у правцу

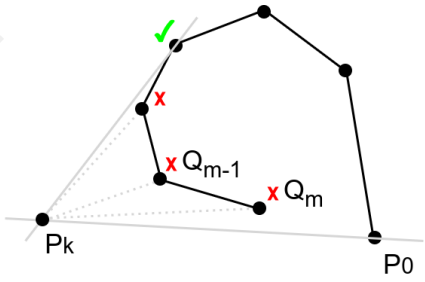
[8] Роналд Грејем (енгл. Ronald Graham), (1935-2020), амерички математичар.

позитивног смера x-осе, и сортирамо тачке према величини ових углова (видети пример на слици 5.19). Тачке које захватају исти угао сортирамо на основу растојања од P_0, опадајуће (тако да се прво обрађују даље тачке). Алтернативно, могуће је и да избацимо све колинеарне тачке осим оне која је најдаља од P_0.

Тачке, дакле, обилазимо редоследом којим се појављују у (простом) многоуглу, и приликом обиласка покушавамо да идентификујемо темена конвексног омотача. Као и код увијања поклона памтимо пут састављен од тачака које су обрађене током обиласка. Прецизније, то је конвексни пут који одређује конвексни многоугао (добијен повезивањем прве и последње тачке пута), који садржи све до сада прегледане тачке. Због тога, је у тренутку кад су све тачке прегледане, конвексни омотач скупа тачака конструисан. Основна разлика између овог алгоритма и алгоритма увијања поклона је у чињеници да текући конвексни пут не мора да буде део коначног конвексног омотача. Он одређује конвексни омотач до сада прегледаних тачака. Дакле текући конвексни пут може да садржи тачке које не припадају коначном конвексном омотачу; те тачке биће елиминисане касније. На пример, пут од P_0 до Q_m на слици 5.32 је конвексан, али тачке Q_m и Q_{m-1} очигледно не припадају конвексном омотачу комплетног скупа тачака. Ово разматрање сугерише алгоритам заснован на следећој индуктивној хипотези.

Индуктивна хипотеза. Ако је дато n тачака у равни, уређених према алгоритму за конструкцију простог многоугла, онда умемо да конструишемо конвексни пут преко неких од првих k тачака, такав да је њему одговарајући конвексни многоугао конвексни омотач првих k тачака.

Случај $k = 1$ је тривијалан (омотач садржи само тачку P_0). Означимо конвексни пут добијен (индуктивно) за првих k тачака са $P = Q_0, Q_1, \ldots, Q_m$ (при чему је $\{Q_0, \ldots Q_m\} \subseteq \{P_0, \ldots, P_{k-1}\}$). Проширимо индуктивну хипотезу на $k+1$ тачака. Посматрајмо угао између правих $Q_{m-1}Q_m$ и Q_mP_k (видети слику 5.32).



Слика 5.32: Илустрација основног корака у Грејемовом алгоритму.

Ако је тај угао мањи од π (при чему се угао мери из унутрашњости многоугла), онда се тачка P_k може додати постојећем путу (нови пут је због тога такође конвексан), чиме

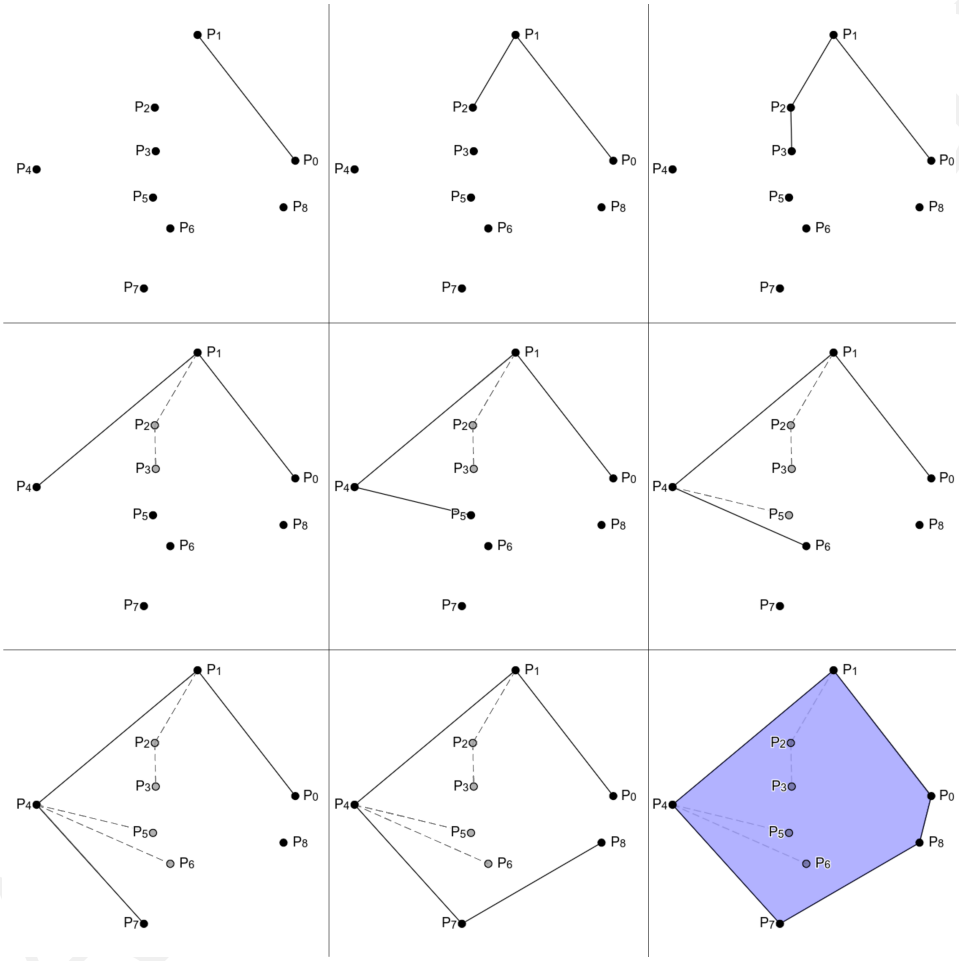
је корак индукције завршен. У противном, тврдимо да тачка Q_m лежи у многоуглу добијеном заменом тачке Q_m у путу P тачком P_k, и повезивањем тачке P_k са тачком P_0. Ово је тачно јер су тачке уређене према одговарајућим угловима, па се права P_0P_k налази „лево" од првих k тачака. Због тога Q_m јесте унутар горе дефинисаног многоугла, може се избацити из пута P, а тачка P_k се може додати текућем путу. Да ли је тиме завршена обрада $(k+1)$-ве тачке? Не мора бити. Након избацивања тачке Q_m добијени пут још не мора да буде конвексан. Заиста, слика 5.32 показује да постоје случајеви кад треба елиминисати још тачака. Наиме, и тачка Q_{m-1} може да буде унутар многоугла дефинисаног модификованим путем. Морамо да (уназад) наставимо са проверама последње две странице пута, све док угао између њих не постане мањи од π: ово се увек мора десити пре него што се исцрпу све тачке, јер ће у најгорем случају барем тачке Q_0, Q_1 и P_k чинити конвексни пут (због претходног уређења тачака). Пут је тада конвексан, а хипотеза је проширена на $k+1$ тачака.

Уместо да рачунамо угао између правих $Q_{m-1}Q_m$ и Q_mP_k, може се разматрати оријентација тројке $Q_{m-1}Q_mP_k$: уколико је оријентација овог троугла позитивна, онда је угао $Q_{m-1}Q_mP_k$ мањи од π, а ако је негативна, тај угао је већи од π. Ако је оријентација једнака нули, тачке $P_kQ_mQ_{m-1}$ су колинеарне. На основу сортирања тачака знамо да тачка Q_m лежи између Q_{m-1} и P_k, па се и у том случају Q_m може заменити тачком P_k.

На слици 5.33 приказани су кораци приликом извршавања Грејемовог алгоритма на једном примеру.

Грејемов алгоритам је могуће имплементирати на следећи начин:

```
vector<Tacka> konveksniOmotac(vector<Tacka>& tacke) {
  // trazimo ekstremnu tacku sa maksimalnom x koordinatom,
  // u slucaju da ima vise tacaka sa maksimalnom x koordinatom
  // biramo onu sa najmanjom y koordinatom
  auto max = max_element(begin(tacke), end(tacke),
                         [](const Tacka& t1, const Tacka& t2) {
                           return t1.x < t2.x ||
                                  (t1.x == t2.x && t1.y > t2.y);
                         });
  // dovodimo je na početak niza
  swap(*begin(tacke), *max);
  const Tacka& t0 = tacke[0];

  // sortiramo ostatak niza (tačke sortiramo na osnovu ugla koji
  // zaklapaju u odnosu horizontalnu polupravu koja polazi nadesno
```



Слика 5.33: Кораци приликом извршавања Грејемовог алгоритма. Приметимо како се тачке P_2 и P_3 прво укључују у омотач, а када се наиђе на тачку P_4 онда се искључују из њега. Слично се касније догађа са тачком P_5, а затим и тачком P_6.

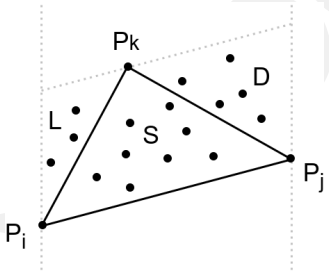
```
  // iz ekstremne tacke), a kolinearne na osnovu rastojanja te tacke
  sort(next(begin(tacke)), end(tacke),
       [t0](const Tacka& t1, const Tacka& t2) {
         Orijentacija o = orijentacija(t0, t1, t2);
         if (o == KOLINEARNE)
           return kvadratRastojanja(t0, t1) <= kvadratRastojanja(t0, t2);
         return o == POZITIVNA;
       });

  // obradjujemo tacke u sortiranom redosledu izbacujuci prethodne sve
  // dok se tekuca tacka sa dve poslednje ne cini levi zaokret
  vector<Tacka> omotac;
  omotac.push_back(tacke[0]);
  omotac.push_back(tacke[1]);
  for (size_t i = 2; i < tacke.size(); i++) {
    while (omotac.size() >= 2 &&
           orijentacija(omotac[omotac.size()-2],
                        omotac[omotac.size()-1],
                        tacke[i]) != POZITIVNA)
      omotac.pop_back();
    omotac.push_back(tacke[i]);
  }
  return omotac;
}
```

Главни део сложености Грејемовог алгоритма потиче од почетног сортирања. Остатак алгоритма извршава се за време $O(n)$. Свака тачка скупа разматра се тачно једном у индуктивном кораку као P_k. У том тренутку тачка се увек додаје конвексном путу. Иста тачка биће разматрана и касније (можда чак и више него једном) да би се проверила њена припадност конвексном путу. Број тачака на које се примењује овакав повратни тест може бити велики, али се све оне, сем две (текућа тачка и тачка за коју се испоставља да даље припада конвексном путу) елиминишу, а тачка може бити елиминисана само једном! Према томе, троши се највише константно време за елиминацију сваке тачке и константно време за њено додавање, те је укупна сложеност ове фазе $O(n)$. Због почетног сортирања тачака време извршавања комплетног алгоритма износи $O(n \log n)$.

5.5.4 Брзи алгоритам за тражење конвексног омотача

Покушајмо да исти проблем решимо техником декомпозиције, налик алгоритму брзог сортирања (енгл. QuickSort). Одредимо тачке из датог скупа са минималном и мак-

сималном x координатом (ако има више таквих онда бирамо ону са минималном или максималном y координатом): нека су то тачке P_i и P_j. Обе ове тачке су екстремне и као такве сигурно припадају конвексном омотачу. Дуж P_iP_j дели скуп тачака на два подскупа: сваки од подскупова чине тачке које се налазе са исте стране те дужи. Размотримо један од ова два подскупа: у њему тражимо тачку на максималном растојању од дужи P_iP_j: нека је то тачка P_k (слика 5.34). И она сигурно припада конвексном омотачу. Тачке из скупа које припадају затвореном троуглу $P_iP_jP_k$ не могу бити део конвексног омотача и не морају се даље разматрати. Претходно разматрање сада можемо применити на две нове дужи које формирају троугао: P_iP_k и P_kP_j: тражимо најудаљенију тачку од дужи P_iP_k са оне стране праве P_iP_k са које није P_j и, слично, најудаљенију тачку од дужи P_kP_j са оне стране праве P_kP_j са које није тачка P_i. Са поступком настављамо све док на располагању има још тачака. Тачке које су током извршавања алгоритма биране као најудаљеније припадају конвексном омотачу. На овај начин дошли смо до тзв. *брзог алгоритма за одређивање конвексног омотача* (енгл. QuickHull algorithm).



Слика 5.34: Одређивање конвексног полуомотача. Тачка P_i има минималну x-координату, P_j максималну x-координату, а тачка P_k је најдаља од дужи P_iP_j. Тачке у затвореном троуглу (у области S) нису део конвексног омотача и елиминишу се. Рекурзивно се обрађује дуж P_iP_k и тачке из скупа L и дуж P_kP_j и тачке из скупа D.

Напоменимо да приликом тражења најудаљеније тачке P_k од дужи P_iP_j, није неопходно рачунати тачно растојање по формули:

$$d = \frac{|\overrightarrow{P_kP_i} \times \overrightarrow{P_kP_j}|}{|\overrightarrow{P_iP_j}|}$$

Наиме, ми тражимо најудаљенију тачку од праве кроз фиксиране две тачке, те ће именилац овог разломка увек бити исти. Стога се мера растојања овде може рачунати по формули $d = |\overrightarrow{P_kP_i} \times \overrightarrow{P_kP_j}|$.

У наставку је приказана C++ имплементација овог алгоритма. Током рекурзивних позива одржава се скуп темена конвексног омотача. На крају је потребно одредити и редослед тих темена, тако да узастопна темена јединственог конвексног многоугла буду једно иза другог. То се може постићи тако што се нека екстремна тачка T_0, у овом случају она са максималном x-координатом, стави на почетак, а затим се остале тачке T_i сортирају по углу који полуправа T_0T_i заклапа са хоризонталном полуправом која почиње у T_0 и простире се у правцу позитивног дела x-осе. Заиста, у траженом конвексном многоуглу тачке су сортиране на тај начин, па се овим сортирањем добија баш тај конвексан многоугао (који је јединствен). Пошто је основни алгоритам такав да је могуће да су у изграђеном омотачу нека три темена колинеарна, након формирања омотача вршимо његово упрошћавање задржавајући само крајња темена у низу колинеарних темена.

```
// funkcija koja racuna konveksni omotac skupa tacaka
vector<Tacka> konveksniOmotac(vector<Tacka>& tacke) {
  // funkcija poredjenja koja je potrebna za bibliotecke funkcije
  // min_element i max_element
  auto porediKoordinate =
    [](const Tacka& A, const Tacka& B) {
      return (A.x < B.x) || (A.x == B.x && A.y > B.y);
    };

  // trazimo tacke sa najmanjom i najvecom x koordinatom
  int min = distance(tacke.begin(),
                     min_element(tacke.begin(), tacke.end(),
                                 porediKoordinate));
  int max = distance(tacke.begin(),
                     max_element(tacke.begin(), tacke.end(),
                                 porediKoordinate));

  // skup temena omotaca
  set<Tacka> temena;

  // rekurzivno trazimo konveksne omotace tacaka sa jedne strane
  // duzi odredjene tackama tacka[min] i tacka[max]
  vector<Tacka> iznad =
    izdvojOrijentaciju(tacke, tacke[min], tacke[max], POZITIVNA);
  konveksniPoluomotac(iznad, tacke[min], tacke[max], temena);
  // i sa druge strane
```

```
  vector<Tacka> ispod =
    izdvojOrijentaciju(tacke, tacke[min], tacke[max], NEGATIVNA);
  konveksniPoluomotac(ispod, tacke[min], tacke[max], temena);

  // prebacujemo tacke iz skupa u vektor i sortiramo ga po uglovima
  vector<Tacka> omotac(temena.begin(), temena.end());
  const Tacka& maxTacka = tacke[max];
  auto porediUgao =
    [maxTacka](const Tacka& A, const Tacka& B) {
      if (A == maxTacka) return true;
      if (B == maxTacka) return false;
      return orijentacija(maxTacka, A, B) == POZITIVNA;
    };
  sort(begin(omotac), end(omotac), porediUgao);

  // eliminisemo susedne kolinearne tacke
  vector<Tacka> omotacBezKolinearnih;
  for (int i = 0; i < omotac.size(); i++) {
    int prethodna = i == 0 ? omotac.size() - 1 : i-1;
    int sledeca = i == omotac.size() - 1 ? 0 : i + 1;
    if (orijentacija(omotac[prethodna], omotac[i], omotac[sledeca]) !=
        KOLINEARNE)
      omotacBezKolinearnih.push_back(omotac[i]);
  }

  return omotacBezKolinearnih;
}

void konveksniPoluomotac(const vector<Tacka>& tacke,
                         const Tacka& P, const Tacka& Q,
                         set<Tacka>& temena) {
  // dodajemo tacke P i Q u omotac
  temena.insert(P);
  temena.insert(Q);

  // ako ne postoji tacka sa date strane prave PQ zavrsavamo postupak
  if (tacke.empty())
    return;
```

```
  // trazimo tacku sa date strane duzi PQ
  // koja je na najvecem rastojanju od prave PQ

  // pozicija najdalje tacke i njeno rastojanje od prave PQ
  int iMax = -1; long long dMax = 0;
  for (int i = 0; i < tacke.size(); i++) {
    long long di = rastojanjeOdPrave(P, Q, tacke[i]);
    if (di > dMax) {
      iMax = i;
      dMax = di;
    }
  }
  // najdalja tacka
  const Tacka& M = tacke[iMax];

  // rekurzivno pozivamo za dva podskupa dobijena tackom `tacke[iMax]`

  // analiziramo tacke koje su na suprotnoj strani prave MP u odnosu na Q
  vector<Tacka> suprotnoOdQ =
    izdvojOrijentaciju(tacke, M, P, -orijentacija(M, P, Q));
  konveksniPoluomotac(suprotnoOdQ, M, P, temena);

  // analiziramo tacke koje su na suprotnoj strani prave MQ u odnosu na P
  vector<Tacka> suprotnoOdP =
    izdvojOrijentaciju(tacke, M, Q, -orijentacija(M, Q, P));
  konveksniPoluomotac(suprotnoOdP, M, Q, temena);
}

// funkcija koja vraca vrednost koja je proporcionalna rastojanju
// od tacke A do prave odredjene tackama P i Q
long long rastojanjeOdPrave(Tacka P, Tacka Q, Tacka A) {
  return abs((long long)(A.y - P.y) * (long long)(Q.x - P.x) -
             (long long)(Q.y - P.y) * (long long)(A.x - P.x));
}

vector<Tacka> izdvojOrijentaciju(const vector<Tacka>& tacke,
                                 const Tacka& P, const Tacka& Q,
                                 int o) {
  vector<Tacka> rezultat;
```

```
  for (const Tacka& T : tacke)
    if (orijentacija(P, Q, T) == 0)
      rezultat.push_back(T);
  return rezultat;
}
```

Сложеност овог алгоритма може се описати рекурентном једначином облика $T(n) = T(k)+T(n-k-1)+O(n)$, где је са k означена величина једног, а са $n-k-1$ величина другог проблема на који се врши подела (налик алгоритму QuickSort). У најбољем случају проблем се увек дели на два потпроблема исте димензије па добијамо рекурентну једначину $T(n) = 2T(n/2) + O(n)$, чије је решење $T(n) = O(n \log n)$. У најгорем случају добијамо рекурентну једначину облика $T(n) = T(n-1) + O(n)$ чије решење задовољава једначину $T(n) = O(n^2)$. Ово одговара сложености алгоритма QuickSort и могуће је доказати да је просечна сложеност овог алгоритма једнака $O(n \log n)$.

Задатак: Најдаље тачке

Дате су координате n телекомуникационих антена које међусобно комуницирају. Брзина комуникације зависи од растојања између антена. Написати програм који одређује највеће растојање између датих антена.

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 50\,000$), а затим координате n тачака на којима су антене (координате су цели бројеви из интервала $[-10^5, 10^5]$).

Опис излаза

На стандардни излаз исписати највеће растојање између две унете тачке (реалан број заокружен на 5 децимала).

Пример

Улаз	*Излаз*	*Објашњење*
7 1 5 -1 4 2 3 3 5 3 3 -1 1 0 3	5.65685	Највеће је растојање између тачака са координатама $(-1, 1)$ и $(3, 5)$ и то растојање износи $4\sqrt{2}$.

Решење

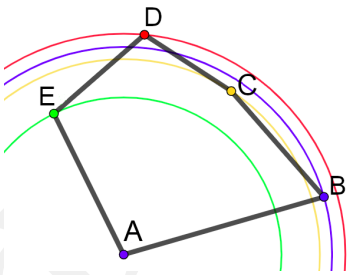
Највеће растојање између две тачке неког скупа се назива *дијаметар скупа тачака*.

Решење грубом силом подразумева израчунавање растојања између свих парова тачака и сложености је $O(n^2)$.

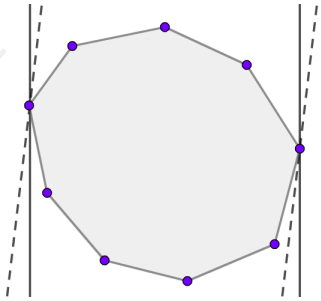
Ефикасније решење започињемо опаском да се две најдаље тачке морају налазити на конвексном омотачу скупа тачака. Дијаметар скупа тачака једнак је дијаметру његовог конвексног омотача. Зато у првој фази алгоритма конструишемо конвексни омотач датог скупа тачака (што Грејемовим алгоритмом можемо урадити у времену $O(n \log n)$).

Након тога можемо проверавати све парове тачака на конвексном омотачу, што у многим случајевима доводи до убрзања, међутим, сложеност најгорег случаја не би била поправљена, јер свих n тачака полазног скупа може лежати на конвексном омотачу. Стога ћемо анализу тачака на конвексном омотачу вршити пажљивије.

За свако теме конвексног омотача ћемо одредити њему најдаље теме на омотачу. Иако можда на први поглед делује да растојања темена од фиксираног темена на конвексном омотачу имају неко својство монотоности док редом обилазимо темена многоугла, то није случај (као што је илустровано на слици 5.35).



Слика 5.35: Растојање од темена A до темена B, C, D, E нема својство монотоности.

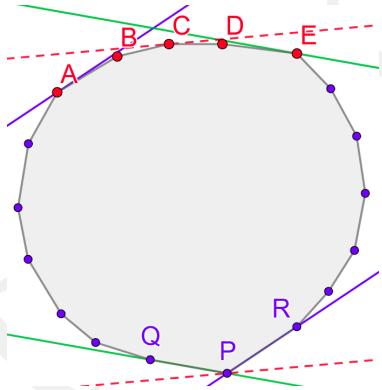


Слика 5.36: Два антиподална темена и два пара паралелних правих које показују да су та темена антиподална.

Кроз пар међусобно најдаљих темена се могу повући две паралелне праве тако да се цео

многоугао налази између те две праве (слика 5.36) — за таква два темена кажемо да су *антиподална*. Те паралелне праве не морају бити јединствене. Теме најдаље од датог темена му је сигурно антиподално, међутим, могуће је да постоји више темена која су антиподална датом темену. Алгоритам ћемо организовати тако што ћемо за свако теме обилазити њему антиподална темена и међу њима тражити оно најдаље од датог.

Како одредити сва темена која су антиподална датом темену P? Права која доказује антиподалност тј. права која пролази кроз P са чије се једне стране налази цео полигон мора лежати између правих које спајају P са њему суседним теменима полигона (на слици 5.37, то су праве PQ и PR) – праву кроз P можемо ротирати све док не достигне једну од те две праве. Сваку од тако добијених правих можемо паралелно померати преко многоугла, све док не достигне теме многоугла које је од ње најдаље и то теме ће сигурно бити аниподално темену P (на слици 5.37, антиподална темена темену P су темена A, B, C, D и E).



Слика 5.37: Одређивање антиподалних тачака тачки P. Тачка A је најдаља тачка од праве PR, а тачка E је најдаља тачка од праве QP. Оне су очигледно антиподалне тачки P, а, антиподалне су јој и све тачке између њих (тачке B, C и D)

Проблем се, дакле, своди на то да се за сваку страницу многоугла пронађе теме које је најдаље од ње. Када се странице обилазе редом (на пример, у позитивном математичком смеру) и најдаља темена од тих страница се померају редом, у истом смеру. Дакле, за разлику од одређивања најдаљег темена од сваког темена многоугла, најдаље теме од сваке странице многоугла има својство монотоности, па се одређивање тих темена може засновати на алгоритму два показивача (један ће се кретати редом кроз странице, а други редом кроз њима најдаља темена). Крећемо од произвољне странице многоугла и грубом силом одређујемо теме које је најдаље од ње. У сваком кораку се померамо на следећу страницу, а затим померамо најдаље теме све док растојање од те следеће странице расте. Растојање можемо лако израчунати као количник површине троугла који

чине та страница и текуће теме и дужине те странице, а пошто се страница не мења, показивач можемо померати све док површина троугла расте. Током тог померања пролазимо кроз све антиподалне тачке почетном темену текуће странице, па рачунајући њихово растојање од тог темена можемо одредити најдаље теме од тог темена, па самим тим и две најдаље тачке почетног скупа.

Сложеност одређивања најдаљег темена од полазне странице захтева један обилазак темена многоугла и сложености је $O(n)$. Сваки од показивача се помера у истом смеру и обилази сва темена многоугла по једном, па је укупна сложеност друге фазе алгоритма $O(n)$. Укупном сложеношћу, дакле, доминира одређивање конвексног омотача почетног скупа, које је сложености $O(n \log n)$.

```
double najdaljiParTacaka(const vector<Tacka>& tacke) {
  // odredjujemo konveksni omotac i broj njegovih tacaka
  vector<Tacka> omotac = konveksniOmotac(tacke);
  int m = omotac.size();

  // specijalni slucajevi kada omotac ima manje od 3 tacke
  if (m == 1) return 0.0;
  if (m == 2) return rastojanje(omotac[0], omotac[1]);

  // odredjujemo najdalju tacku od duzi Qm-1Q0
  int najdaljaTacka = 2;
  for (int i = 3; i < omotac.size(); i++)
    if (povrsina(omotac[m-1], omotac[0], omotac[i]) >
        povrsina(omotac[m-1], omotac[0], omotac[najdaljaTacka]))
      najdaljaTacka = i;

  // prva antipodalna tacka od tacke Q0
  double maxRastojanje = rastojanje(omotac[0], omotac[najdaljaTacka]);
  // obradjujemo ostale tacke Qi
  for (int i = 0; i < m; i++) {
    // do sada je obradjena prva antipodalna tacka tacke Qi
    // sada redom obradjujemo njene ostale antipodalne tacke
    // njih nalazimo trazeci najdalju tacku od duzi QiQi+1
    while (povrsina(omotac[i], omotac[(i+1)%m], omotac[najdaljaTacka]) <=
           povrsina(omotac[i], omotac[(i+1)%m], omotac[(najdaljaTacka+1)%m])) {
      // za svaku antipodalnu tacku od Qi odrequjemo rastojanje od Qi
      maxRastojanje = max(maxRastojanje,
```

```
                        rastojanje(omotac[i], omotac[najdaljaTacka]));
      // i sledeca tacka je antipodalna tacki Qi
      najdaljaTacka = (najdaljaTacka + 1) % n;
    }
  }
  return maxRastojanje;
}
```

Задатак: Пресек конвексних многоуглова

Написати програм који одређује пресек два конвексна многоугла.

Опис улаза

Са стандардног улаза се учитава број $m(3 \leq m \leq 10^5)$ који представља број темена првог многоугла P, затим се учитава низ x-координата темена тог многоугла, а затим низ y-координата тих темена. Темена су задата редом, у позитивном математичком смеру, а координате су им реални бројеви задати са највише 5 децимала.

Након тога се у истом формату уносе и подаци за други многоугао Q: број темена $n(3 \leq n \leq 10^5)$, низ x-координата и низ y-координата.

Опис излаза

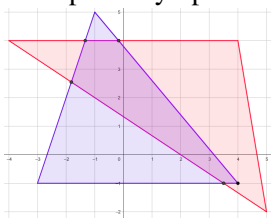
На стандардни излаз исписати координате конвексног многоугла који је пресек многоуглова P и Q. Исписати његов број темена, а затим и координате темена, редом, у позитивном математичком смеру (кренувши од произвољног темена). Координате свих темена исписати заокружене на 5 децимала.

Пример 1

Улаз	*Излаз*
3	5
-3 -1 4	-1.81818 2.54545
-1 5 -1	-1.33333 4.00000
3	-0.16667 4.00000
-4 5 4	4.00000 -1.00000
4 -2 4	3.50000 -1.00000

Објашњење

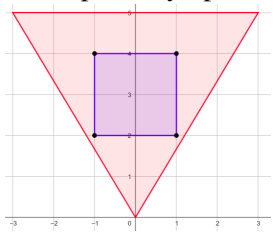
Многоуглови и њихов пресек су приказани на слици.



Пример 2

Улаз	*Излаз*	*Објашњење*

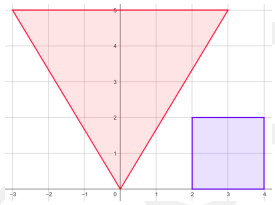
`3`
`-3 0 3`
`5 0 5`
`4`
`-1 1 1 -1`
`2 2 4 4`
	`4`
	`-1.00000 2.00000`
	`1.00000 2.00000`
	`1.00000 4.00000`
	`-1.00000 4.00000`
		Многоуглови и њихов пресек су приказани на слици.



Пример 3

Улаз	*Излаз*	*Објашњење*

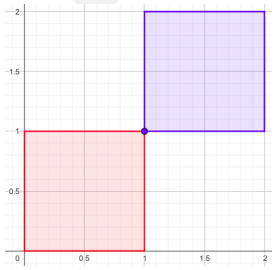
`3`
`-3 0 3`
`5 0 5`
`4`
`2 4 2 4`
`0 0 2 2`
	`0`	Многоуглови су приказани на слици. Они се не секу



Пример 4

Улаз	*Излаз*	*Објашњење*

`4`
`0 1 1 0`
`0 0 1 1`
`4`
`1 2 2 1`
`1 1 2 2`
	`1`
	`1.00000`
	`1.00000`
		Многоуглови су приказани на слици. Њихов пресек је једна тачка



Решење

Директно решење

Једно могуће директно решење се може засновати на различитим алгоритмима које смо до сада описали. Пресек два конвексна многоугла је конвексни многоугао чији се скуп темена састоји од:

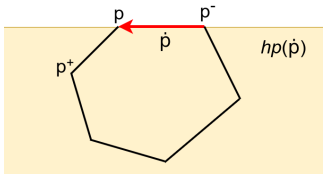
- темена првог многоугла која припадају унутрашњости другог (затвореног) многоугла,
- темена другог многоугла која припадају унутрашњости првог (затвореног) многоугла,
- пресечних тачака страница многоуглова.

Када се одреде сва темена у ова три скупа, потребно их је уредити, што се може урадити, на пример, сортирањем на основу угла који полуправа која спаја тежиште резултујућег многоугла са тренутним теменом заклапа са позитивним смером x-осе (пошто су сва темена позната, тежиште се лако израчунава израчунавањем аритметичке средине њихових координата).

За сваку тачку многоуглова можемо покренути проверу припадности другом многоуглу, чија је сложеност, у случају конвексних многоуглова, логаритамска. Стога је за одређивање прва два скупа темена потребно време $O(m \log n + n \log m)$. Да би се одредили пресеци страница, можемо алгоритам за одређивање пресека две дужи применити на све парове страница два многоугла, за шта је потребно време $O(mn)$. Број парова дужи које се испитују је могуће смањити тако што се испитују само дужи чија су темена таква да једно припада, а друго не припада многоуглу. Ипак, у наставку ћемо видети да постоји и алгоритам линеарне сложености $O(m + n)$ за одређивање пресека два многоугла, па овај алгоритам нећемо даље разрађивати.

Ефикасан алгоритам

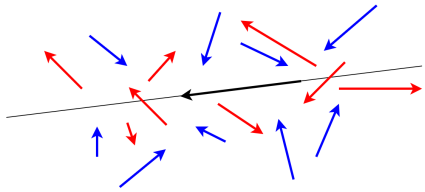
Један ефикасан алгоритам за одређивање пресека два конвексна многоугла су описали О Рурк, Чин, Олсон и Надор. Алгоритам је заснован на техници два показивача. Показивач p обилази многоугао P у позитивном математичком смеру, а показивач q многоугао Q такође у позитивном смеру. Са p^-, p и p^+ биће обележене редом претходна, текућа и следећа тачка многоугла P. Положајем показивача p одређена је текућа страница многоугла, тј. усмерена дуж p^-p која представља ту текућу страницу, а која је увек усмерена у позитивном смеру тј. тако да јој се многоугао налази са леве стране (слика 5.38). Врх те дужи је тачка p, дно јој је тачка p^-, док ћемо саму дуж обележити са $\dot{p}$. Са $hp(\dot{p})$ ћемо обележити полураван чија је ивица права која садржи $\dot{p}$, и која садржи многоугао P (тј. налази се лево од $\dot{p}$). Произвољна тачка m припада полуравни $hp(\dot{p})$ ако и само ако је $\overrightarrow{p^-p} \times_{2d} \overrightarrow{p^-m} \geq 0$. Аналогне ознаке користимо и за q.



Слика 5.38: Показивач p и основне ознаке.

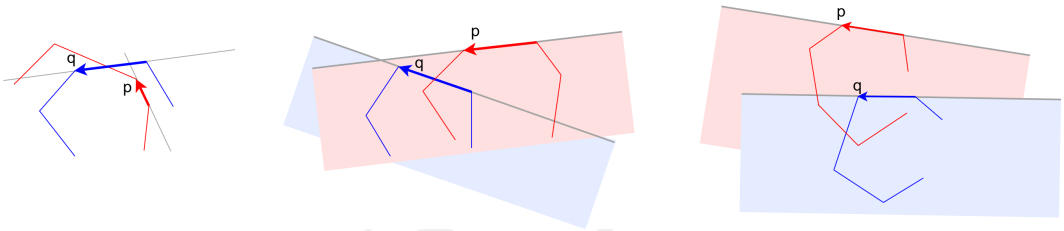
Интуитивно је јасно када усмерена дуж указује ка правој, а када указује од ње (слика 5.39). Ово се може и прецизирати. Вектор $\dot{p}$ указује према правој на којој лежи $\dot{q}$ ако и само ако је $p \in hp(\dot{q})$ и $\dot{q} \times_{2d} \dot{p} < 0$ или $p \notin hp(\dot{q})$ и $\dot{q} \times_{2d} \dot{p} \geq 0$ (ако је та права хоризонтална, а дуж q усмерена на лево, као на слици, тада ово значи да је дуж

$\dot{p}$ усмерена на горе и врх јој је у полуравни испод праве или је усмерена на доле и врх јој је у полуравни изнад праве).



Слика 5.39: Плаве дужи указују ка црној правој, а црвене не.

Показивачи се померају тако да се гарантује да ће се током њиховог померања пронаћи сви пресеци страница. У сваком кораку алгоритма се проверава да ли се дужи $\dot{p}$ и $\dot{q}$ секу и ако се секу, пресечна тачка се додаје резултујућем многоуглу. Након тога се помера један од два показивача. Размотримо разне могућности за усмерене дужи $\dot{p}$ и $\dot{q}$.



Слика 5.40: Разни случајеви који могу наступити приликом померања показивача.

1. Могуће је да дуж $\dot{p}$ указује ка правој на којој лежи $\dot{q}$, а да дуж $\dot{q}$ не указује ка правој на којој лежи $\dot{p}$ (слика 5.40, лево). У том случају померамо показивач p, у циљу да у неком тренутку дуж $\dot{p}$ пресече праву на којој лежи $\dot{q}$ (ивицу полуравни $hp(\dot{q})$) и дође до полуравни $hp(\dot{q})$ у којој лежи многоугао Q, па и сви пресеци које треба пронаћи. Случај када $\dot{q}$ указује ка правој на којој лежи $\dot{p}$, а $\dot{p}$ не указује ка правој на којој лежи $\dot{q}$ се обрађује аналогно.

2. Могуће је да обе дужи $\dot{p}$ и $\dot{q}$ указују ка правама на којој лежи она друга (слика 5.40, средина). У том случају, због конвексности мора да важи да тачно једна од тачака p или q припада полуравни којој је граница права на којој лежи друга дуж и у којој се налази њој одговарајући многоугао. У примеру на слици тачка p не припада полуравни $hp(\dot{q})$, одређеном правом на којој је дуж $\dot{q}$ и у којој се налази многоугао Q, док тачка q припада полуравни $hp(\dot{p})$ одређеном правом на којој је дуж $\dot{p}$ и у којој се налази многоугао Q. Да би дуж $\dot{p}$ секла неку страницу многоугла Q, потребно је да тачка p припада унутрашњости многоугла Q, па самим тим и полуравни $hp(\dot{q})$. Аналогно важи и за $\dot{q}$ и P. Зато се у овом случају помера

онај показивач који одговара тачки која не припада унутрашњости одговарајуће полуравни.

3. Могуће је и да ниједна дуж не указује ка правој на којој лежи она друга (слика 5.40, десно). И у том случају због конвексности мора да важи да тачно једна од тачака p тј. q припада унутрашњости многоугла Q тј. P и поново је потребно померити ону која не припада унутрашњости одговарајуће полуравни.

Дакле, када год тачно једна од две усмерене дужи указује ка правој на којој лежи она друга, њу померамо, док у осталим случајевима померамо врх оне дужи који не лежи у унутрашњости полуравни са леве стране праве одређене другом дужи.

Алгоритам се може зауставити када се цео пресечни многоугао комплетира тј. када се последњи пронађени пресек дужи поклопи са првим пронађеним пресеком (пресеци ће бити одређени редом, у позитивном смеру). Међутим, ако пресеци страница не постоје, показивачи ће се непрестано померати. Доказује се да се након највише $2(|P| + |Q|)$ корака проналазе сви пресеци страница, тако да се алгоритам може зауставити и ако се након $2(|P| + |Q|)$ не пронађе ниједна пресечна тачка. У том случају, ако p припада унутрашњости Q, пресек је цео многоугао P, ако q припада унутрашњости P, пресек је цео многоугао Q, док се у супротном многоуглови не секу.

У претходном опису је претпостављено да не постоје дегенерисани случајеви, тј. да се темена многоуглова не поклапају, да ниједно теме једног не припада страници другог многоугла нити да се странице многоуглова поклапају. И ови случајеви се могу исправно обрадити ако се алгоритам мало прилагоди. Прво, за две колинеарне дужи (када је $\dot{p} \times \dot{q} = 0$) се увек сматра да се не секу. Друго, критеријум заустављања треба прилагодити. Наиме, када страница садржи теме, може се догодити да се исти пресек нађе и након померања показивача. Зато се може десити да се алгоритам заустави јер ће други пронађени пресек бити једнак првом. Стога се алгоритам може зауставити када је текући пресек једнак првом пронађеном, али под условом да тај први пресек није пронађен у претходном кораку алгоритма.

```
double uPoluravni(const Tacka& A, const Tacka& B, const Tacka& M) {
  Vektor AB = vektor(A, B);
  Vektor AM = vektor(A, M);
  return vektorski_proizvod(AB, AM) >= 0;
}

vector<Tacka> presekMnouglova(const vector<Tacka>& P, const vector<Tacka>& Q) {
  int m = P.size(), n = Q.size();
  vector<Tacka> presek;
```

```
// dva pokazivaca
int pm = 0, p = 1;
int qm = 0, q = 1;

// ili P[p] pripada poluravni hp(Q[qm]Q[q])
// ili Q[q] pripada poluravni hp(P[pm]P[p])
// ovom promenljivom registrujemo za koju od tacaka p ili q je to vazilo
// prilikom pronalaska prethodnog preseka
char unutra = ' ';
// presecna tacka nadjena u prethodnom koraku (mozda i nije postojao presek)
Tacka prethodniPresek;
bool postojiPrethodniPresek = false;
for (int i = 0; i <= 2*(m + n); i++) {
  Tacka M;
  // proveravamo da li se duzi seku
  if (presekDuzi(P[pm], P[p], Q[qm], Q[q], M)) {
    if (presek.size() > 0) {
      // vratili smo se na pocetnu tacku i zatvorili ceo presecni mnogougao
      // presek nije pronadjen u prethodnom koraku
      if (M == presek[0] && (!postojiPrethodniPresek || M != prethodniPresek))
        return presek;
    }
    // nasli smo novu presecnu tacku i dodajemo je u presecni mnogougao
    if (presek.empty() || M != presek.back())
      presek.push_back(M);

    // pripremamo se za naredni korak tako sto pamtimo tekuci presek
    prethodniPresek = M;
    postojiPrethodniPresek = true;

    // pamtimo da li p pripada poluravni hp(q.) ili q pripada poluravni hp(p.)
    if (uPoluravni(Q[qm], Q[q], P[p]))
      unutra = 'P';
    else
      unutra = 'Q';
  } else
    // pripremamo se za naredni korak tako sto belezimo da nije bilo preseka
    postojiPrethodniPresek = false;
```

```
// funkcija kojom se pomera pokazivac p
auto pomeriP =
  [&](){
    // ako je poslednji presek P[p] pripadao poluravni hp(Q[qm]Q[q]),
    // onda je tekuce P[p] deo presecnog mnogougla
    if (unutra == 'P' && (presek.empty() || P[p] != presek.back()))
      presek.push_back(P[p]);
    // pomeramo se na sledecu tacku mnogougla P
    pm = p;
    p = (p + 1) % m;
  };

// funkcija kojom se pomera pokazivac q
auto pomeriQ =
  [&](){
    // ako je poslednji presek Q[q] pripadao poluravni hp(P[pm]P[p]),
    // onda je tekuce Q[q] deo presecnog mnogougla
    if (unutra == 'Q' && (presek.empty() || Q[q] != presek.back()))
      presek.push_back(Q[q]);
    // pomeramo se na sledecu tacku mnogougla Q
    qm = q;
    q = (q + 1) % m;
  };

Vektor pdot = vektor(P[pm], P[p]);
Vektor qdot = vektor(Q[qm], Q[q]);
if (vektorski_proizvod(qdot, pdot) >= 0) {
  if (uPoluravni(Q[qm], Q[q], P[p])) {
    // pdot nije usmeren ka qdot i P[p] pripada poluravni hp(qdot)
    // q treba pomeriti:
    //   i ako qdot jeste usmeren ka pdot (jer je jedini usmeren)
    //   i ako qdot nije usmeren ka pdot  (jer se pomera tacka
    //         koja ne pripada poluravni, a to je Q[q])
    pomeriQ();
  } else {
    // pdot je usmeren ka qdot i P[p] ne pripada poluravni hp(qdot)
    // p treba pomeriti:
    //   i ako qdot nije usmeren ka pdot (jer je pdot jedini usmeren)
```

```
        //   i ako qdot jeste usmeren ka pdot (jer se pomera tacka
        //         koja ne pripada poluravni, a to je P[p])
        pomeriP();
      }
    } else {
      if (uPoluravni(P[pm], P[p], Q[q])) {
        // qdot nije usmeren ka pdot i Q[q] pripada poluravni hp(pdot)
        // p treba pomeriti
        //   i ako pdot jeste usmeren ka qdot (jer je pdot jedini usmeren)
        //   i ako pdot nije usmeren ka qdot (jer se pomera tacka
        //         koja ne pripada poluravni, a to je P[p])
        pomeriP();
      }
      else {
        // qdot jeste usmeren ka pdot i Q[q] ne pripada poluravni hp(pdot)
        // q treba pomeriti
        //   i ako pdot nije usmeren ka qdot (jer je qdot jedini usmeren)
        //   i ako pdot jeste usmeren ka qdot (jer se pomera tacka
        //         koja ne pripada poluravni, a to je Q[q])
        pomeriQ();
      }
    }
  }

  // ili je presek prazan, ili je P sadrzan u Q ili je Q sadrzan u P

  if (uMnogouglu(Q[0], P))
    // Q je sadrzan u P
    presek = Q;
  else if (uMnogouglu(P[0], Q))
    // P je sadrzan u Q
    presek = P;

  return presek;
}
```

Литература

1. Миодраг Живковић, *Алгоритми*, Математички факултет, Београд, 2000.
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*, 3rd Edition, MIT Press, 2009.
3. Robert Sedgewick, Kevin Wayne, *Algorithms*, Addison-Wesley Professional; 4th edition, 2011.
4. Udi Manber, *Introduction to algorithms, a creative approach*, Addison-Wesley, Reading, 1989.
5. Jeff Erickson, *Algorithms*, available at https://jeffe.cs.illinois.edu/teaching/algorithms/#book 2019.
6. Antti Laaksonen, *Guide to Competitive Programming, Learning and Improving Algorithms Through Contests*, Springer, 2017.
7. Data Structures and Algorithms, OpenDSA hypertext project, https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/#
8. Algorithms for Competitive Programming, Editors: Jakob Kogler, Oleksandr Kulkov, Rodion Gorkovenko, https://cp-algorithms.com/
9. Драган Урошевић, Алгоритми и структуре података, Рачунарски факултет, Београд.

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.421.2(075.8)

МАРИНКОВИЋ, Весна, 1982-
Конструкција и анализа алгоритама / Весна Маринковић, Филип Марић ; [илустрације аутори]. - Београд : Математички факултет, 2024 (Београд : Скрипта интернационал). - 590 стр. : граф. прикази, табеле ; 24 cm

Тираж 150. - Библиографија: стр. [591].

ISBN 978-86-7589-195-6

1. Марић, Филип, 1978- [autor] [ilustrator]

a) Алгоритми b) Програмирање

COBISS.SR-ID 153335817