Алгоритми

Миодраг Живковић

Математички факултет, Београд
Е-маил аддресс: езивковм˘поинцаре.матф.бг.ац.ю

Предговор

Књига садржи уводни курс у конструкцију и анализу алгоритама. Намењена је студентима смера Рачунарство и информатика на Математичком факултету у Београду, за изборни предмет Примена рачунара. Књига такође може да послужи средњошколцима за припрему за такмичења из информатике. За разумевање и извођење анализе алгоритама потребно је основно предзнање из анализе и вероватноће. Наравно, књига ће сигурно користити и свима онима који се баве програмирањем.

Материјал се углавном заснива на књизи [**1**], одакле потиче и највећи део задатака. Редукован је део о графовима, а део о алгебарским алгоритмима проширен је појмовима из криптологије (према [**6**]). Коришћени су такође неки делови из књига [**2, 3, 4, 5, 7, 8, 9**]. За читаоца кога занима ова област, могу бити интересантне и књиге [**10, 11, 12, 13, 14, 15**].

Пошто је ово једна од првих књига о алгоритмима на нашем језику, терминологија се још није усталила. Приликом превода енглеских термина тежило се да превод буде близак оригиналу. Неки енглески термини који су практично прихваћени у нашем језику, нису превођени. На крају књиге су дати речници термина у оба смера. Тако се на једном месту може стећи увид у предложену терминологију.

Намена ове књиге је да читалац не само упозна ефикасне алгоритме за решавање неких конкретних проблема, него и да савлада "вештину" конструкције ефикасних алгоритама за решавање нових проблема. Уместо да се излажу готови, завршени алгоритми, приказује се њихов развој, полазећи од најједноставније варијанте (приступ преузет из [**1**]). Коначни облик алгоритама прилагођен је пре свега разумевању принципа на којима се они заснивају, а тек онда једноставности програмирања.

При конструкцији алгоритама највише се користи приступ заснован на доказивању тврђења математичком индукцијом. Иако су циљеви конструкције алгоритма и доказивања теореме различити, сличност ове две активности је неочекивано велика.

Ефикасни ненумерички алгоритми важни су у многим областима, као што су математика, статистика, молекуларна биологија, техника, ... Многи стручњаци сматрају бављење алгоритмима за досадан, неинвентиван посао (што он понекад и јесте). Али такав приступ проблему често као резултат даје тривијална и неефикасна решења — чак и тамо где постоје елегантна, ефикаснија

решења. Ова књига требало би да пружи аргументе да су алгоритми елегантна и важна дисциплина.

У првом делу књиге, у поглављима 1-4, дат је уводни материјал о математичкој индукцији, анализи алгоритама, структурама података, као и примери конструкције алгоритама помоћу математичке индукције. Наредних четири поглавља 5-8 баве се алгоритмима за решавање проблема у појединим областима: у вези са низовима и скуповима, графовима, из алгебре и геометрије. Поглавље 9 даје основне појмове из криптографије, области веома значајне кроз целу људску историју, у којој алгоритми имају велику примену. Редукцијама, односно међусобним свођењима алгоритама, посвећено је поглавље 10, а NP-комплетни проблеми анализирани су у поглављу 11. Последње поглавље садржи увод у паралелне алгоритме.

Прелиминарна верзија књиге је исправљена и допуњена задацима са решењима. Велики број грешака открили су студенти, на чему им се аутор захваљује. Аутор је унапред захвалан и за све будуће примедбе и сугестије на садржај књиге. Поруке се могу слати на адресу езивковм˜матф.бг.ац.ю. Тежи задаци означени су звездицом. У решавању задатака учествовали су и студенти, што је наставу чинило интересантнијом. Аутор је захвалан нарочито постдипломцима А. Самарџићу и М. Вугделији, који су решили већи број задатака.

За корисне примедбе и сугестије аутор посебну захвалност дугује рецензентима.

Миодраг Живковић

Садржај

Предговор иии

Поглавље 1. Математичка индукција 1
1.1. Увод 1
1.2. Два једноставна примера 2
1.3. Бројање области на које је подељена раван 2
1.4. Проблем са бојењем равни 3
1.5. Пример са сумирањем 4
1.6. Ојлерова формула 4
1.7. Грејови кодови 5
1.8. Налажење дисјунктних путева у графу 7
1.9. Неједнакост између аритметичке и геометријске средине 9
1.10. Инваријанта петље — доказ исправности алгоритма 10
1.11. Најчешће грешке 11
1.12. Резиме 12
Задаци 12

Глава 2. Анализа алгоритама 15
2.1. Увод 15
2.2. Асимптотска ознака O 16
2.3. Временска и просторна сложеност 18
2.4. Сумирање 18
2.5. Диференцне једначине 20
2.6. Резиме 27
Задаци 27

Глава 3. Структуре података 31
3.1. Елементарне структуре података 32
3.2. Стабла 34
3.3. Хип 36
3.4. Бинарно стабло претраге 38
3.5. АВЛ стабла 44
3.6. Хеш табеле 47
3.7. Проблем формирања унија 50
3.8. Графови 52
3.9. Резиме 53

Задаци 54

Глава 4. Конструкција алгоритама индукцијом 57
4.1. Увод 57
4.2. Израчунавање вредности полинома 57
4.3. Максимални индуковани подграф 59
4.4. Налажење бијекције 61
4.5. Проблем проналажења звезде 64
4.6. Пример примене декомпозиције: максимум скупа правоугаоника 66
4.7. Израчунавање фактора равнотеже бинарног стабла 69
4.8. Налажење максималног узастопног подниза 70
4.9. Појачавање индуктивне хипотезе 72
4.10. Динамичко програмирање, проблем ранца 72
4.11. Уобичајене грешке 75
4.12. Резиме 76
Задаци 77

Глава 5. Алгоритми за рад са низовима и скуповима 79
5.1. Увод 79
5.2. Бинарна претрага и варијације 79
5.3. Сортирање 85
5.4. Ранговске статистике 102
5.5. Компресија података 104
5.6. Тражење узорка у тексту 108
5.7. Упоређивање низова 115
5.8. Пробабилистички алгоритми 119
5.9. Резиме 122
Задаци 123

Глава 6. Графовски алгоритми 127
6.1. Увод 127
6.2. Ојлерови графови 129
6.3. Обиласци графова 130
6.4. Тополошко сортирање 142
6.5. Најкраћи путеви из задатог чвора 145
6.6. Минимално повезујуће стабло 152
6.7. Сви најкраћи путеви 156
6.8. Транзитивно затворење 158
6.9. Упаривање 160
6.10. Транспортне мреже 164
6.11. Хамилтонови циклуси 170
6.12. Резиме 172
Задаци 173

Глава 7. Геометријски алгоритми 181

7.1. Увод 181
7.2. Утврђивање да ли задата тачка припада многоуглу 182
7.3. Конструкција простог многоугла 185
7.4. Конвексни омотач 186
7.5. Најближи пар тачака 192
7.6. Пресеци хоризонталних и вертикалних дужи 196
7.7. Резиме 200
Задаци 201

Глава 8. Алгебарски алгоритми 205
8.1. Увод 205
8.2. Степеновање 206
8.3. Еуклидов алгоритам 207
8.4. Множење полинома 210
8.5. Множење матрица 211
8.6. Брза Фуријеова трансформација 214
8.7. Резиме 219
Задаци 220

Глава 9. Примене у криптографији 223
9.1. Увод 223
9.2. Класична криптографија 227
9.3. Случајна шифра 229
9.4. DES 230
9.5. RSA 240
9.6. Резиме 243
Задаци 243

Глава 10. Редукције 245
10.1. Увод 245
10.2. Примери редукција 246
10.3. Редукције на проблем линеарног програмирања 249
10.4. Примена редукција на налажење доњих граница 253
10.5. Уобичајене грешке 256
10.6. Резиме 258
Задаци 258

Глава 11. NP-комплетност 261
11.1. Увод 261
11.2. Редукције полиномијалне временске сложености 262
11.3. Недетерминизам и Кукова теорема 264
11.4. Примери доказа NP-комплетности 272
11.5. Технике за рад са NP-комплетним проблемима 282
11.6. Резиме 293
Задаци 293

Глава 12. Паралелни алгоритми 295
12.1. Увод 295
12.2. Модели паралелног израчунавања 296
12.3. Алгоритми за рачунаре са заједничком меморијом 298
12.4. Алгоритми за мреже рачунара 309
12.5. Систолички алгоритми 321
12.6. Резиме 323
Задаци 324

Глава 13. Упутства и решења задатака 327
13.1. Математичка индукција 327
13.2. Анализа алгоритама 332
13.3. Структуре података 336
13.4. Конструкција алгоритама индукцијом 340
13.5. Алгоритми за рад са низовима и скуповима 344
13.6. Графовски алгоритми 351
13.7. Геометријски алгоритми 362
13.8. Алгебарски алгоритми 368
13.9. Примене у криптографији 372
13.10. Редукције 374
13.11. NP-комплетност 377
13.12. Паралелни алгоритми 380

Српско-енглески и енглеско-српски терминолошки речник 385
Српско-енглески речник 385
Енглеско-српски речник 390

Литература 397

ПОГЛАВЉЕ 1

Математичка индукција

1.1. Увод

Математичка индукција игра значајну улогу при конструкцији многих алгоритама. Због тога се у овом уводном поглављу анализира неколико карактеристичних примера примене индукције.

Нека је $N = \{1, 2, \ldots\}$ скуп природних бројева. Принцип математичке индукције може се формулисати на следећи начин. Претпоставимо да треба доказати да је тврђење $T(n)$ тачно за сваки природан број $n \in N$. Уместо да се ово тврђење "напада" директно, довољно је доказати следећа два тврђења:

- $T(n)$ је тачно за $n = 1$, и
- за свако $n > 1$ важи: ако је тачно $T(n-1)$, онда је тачно и $T(n)$.

Ова чињеница је такозвани принцип индукције, и представља уствари аксиому, која дефинише скуп природних бројева.

У пракси се обично прво тврђење (база индукције) лако доказује. Доказ другог тврђења олакшан је претпоставком да је тачно $T(n-1)$, такозваном *индуктивном хипотезом.* Другим речима, довољно је тврђење свести на случај кад је n умањено за један. Илустроваћемо то једним једноставним примером.

Теорема 1.1. *За свака два броја $x, n \in N$ важи $x - 1 | x^n - 1$.*

Доказ. Доказ се изводи индукцијом по n. За $n = 1$ тврђење гласи $x-1|x-1$ и очигледно је тачно. Свођење случаја n на случај $n-1$ заснива се на изразу $x^n - 1 = (x^{n-1} - 1)x + (x - 1)$, јер он показује да из индуктивне хипотезе $x - 1|x^{n-1} - 1$ следи да је тачно $x - 1|x^n - 1$. □

Принцип математичке индукције често се користи у нешто промењеним облицима, који су непосредна последица основног.

Теорема 1.2. *Ако је тачно тврђење $P(1)$ и за свако $n > 1$ из претпоставке да је $P(k)$ тачно за свако $k < n$ следи тачност $P(n)$, онда је $P(n)$ тачно за свако $n \in N$.*

Ово је тзв. потпуна индукција, а добија се од основне варијанте применом на тврђење $T(n) = \bigwedge_{k=1}^{n} P(k)$.

Теорема 1.3 (Регресивна индукција)**.** *Нека је $a_1, a_2, \ldots$ растући низ природних бројева. Ако је тврђење $P(n)$ тачно за $n = a_k$, $k = 1, 2, \ldots$ и за свако*

$n \geq 2$ из претпоставке да је тачно $P(n)$ следи да је тачно $P(n-1)$, тада је $P(n)$ тачно за свако $n \in N$.

Први део претпоставке, да је $P(a_k)$ тачно за $k \geq 1$ доказује се индукцијом ($P(1)$, и ако $P(a_k)$ онда $P(a_{k+1})$, $k \geq 1$). Дакле, најпре се доказује да постоје произвољно велики бројеви n такви да је $P(n)$ тачно, а онда да је $P(n)$ тачно и за све "изостављене" бројеве. Ово је тзв. регресивна индукција.

1.2. Два једноставна примера

Потребно је израчунати суму $S(n) = 1+2+\ldots+n$. Одговор на ово питање даје следећа теорема.

Теорема 1.4. *Нека је $S(n) = \sum_{i=1}^{n} i$. За свако $n \in N$ је $S(n) = \frac{1}{2}n(n+1)$.*

Доказ. Како је $1 = \frac{1}{2} \cdot 1 \cdot 2$, тврђење је тачно за $n = 1$. Ако се претпостави да је оно тачно за неко n, онда је

$$S(n+1) = 1+2+\ldots+n+(n+1) = S(n)+n+1 = \frac{1}{2}n(n+1)+(n+1) = \frac{1}{2}(n+1)(n+2),$$

односно закључујемо да је тврђење тачно и за $n+1$. □

Теорема 1.5. *Ако је $n \in N$ и $1+x > 0$ онда је $(1+x)^n \geq 1+nx$.*

Доказ. За $n = 1$ тврђење је очигледно тачно. Ако је тврђење теореме тачно за неко n (индуктивна хипотеза), онда је

$$(1+x)^{n+1} = (1+x)(1+x)^n \geq (1+x)(1+nx) = 1+(n+1)x+nx^2 \geq 1+(n+1)x,$$

односно тврђење теореме је тачно и ако се у њему n замени са $n+1$. □

1.3. Бројање области на које је подељена раван

За неколико правих у равни каже се да су у општем положају ако никоје две нису паралелне, а никоје три се не секу у истој тачки. Треба одредити број области на које раван дели $n \geq 1$ правих у општем положају (претпоставка да су праве у општем положају уведена је да поједностави проблем, јер тада нема потребе за анализом разних могућих специјалних случајева).

Непосредно се види да једна, две, односно три праве у општем положају деле раван редом на две, четири, односно седам области. Наслућује се да укључивање n-те праве повећава за n број области на које је раван подељена. Ако се претпостави да је то тачно, онда је број области на које n правих у општем положају дели раван $2+2+3+4+\ldots+n = \frac{1}{2}n(n+1)+1$. Дакле, остаје да се докаже

Хипотеза 1.6. *Додавање n-те праве у равни (при чему су свих n правих у општем положају) повећава број области на које је раван подељена за n.*

Доказ. Доказаћемо ово тврђење индукцијом по n. Тврђење је очигледно тачно за $n = 1, 2, 3$. Претпоставимо да је оно тачно за неко n; нека је дато $n+1$ правих у равни у општем положају, и нека је p једна од тих правих. Због тога што су праве у општем положају, права p са произвољном облашћу, од оних на коју раван деле осталих n правих, нема заједничких тачака, или је сече. Права p сече свих осталих n правих, тј. сече $n + 1$ област, што значи да се њеним додавањем број области повећава за $n + 1$. □

Примедба. Овде је индукција примењена два пута: повећање броја области додавањем n-те праве, а затим укупан број области на које раван дели n правих. Овакав поступак, понекад и са више нивоа, често се примењује.

1.4. Проблем са бојењем равни

Произвољних n правих у равни, дели раван на области; за праве се не претпоставља да су у општем положају. Кажемо да је раван обојена са две боје (на пример црном и белом) ако је свака област обојена једном од те две боје, а сваке две суседне области (оне које имају део праве или целу праву као границу) су обојене различитим бојама.

Теорема 1.7. *Раван подељена са произвољних n правих може се обојити са две боје.*

Доказ. Тврђење теореме доказује се индукцијом по броју правих n. За $n = 1$ тврђење је очигледно тачно (раван је подељена на две области, једна од њих боји се црно, друга бело). Претпоставимо да се раван, издељена са произвољних $n - 1$ правих, може обојити са две боје. Додавање n-те праве на произвољан начин дели неке области на два суседна дела обојена истом бојом. Међутим, ако се свим областима са једне стране n-те праве боја промени у супротну, добија се исправно бојење равни. Заиста, границе између суседних области могу се поделити на три врсте: границе са једне, односно друге стране n-те праве, и границе на n-тој правој. Границе све три врсте одвајају области обојене различитим бојама. Дакле, полазећи од индуктивне хипотезе да се може обојити раван подељена се $n - 1$ правих, показано је како се добија бојење равни подељене са n произвољних правих. □

Примедба. Овај пример је илустрација максималног, еластичног коришћења индуктивне хипотезе: од бојења области у једној полуравни направљено је друго исправно бојење заменом беле и црне боје.

1.5. Пример са сумирањем

Претпоставимо да су сви непарни бројеви исписани у овиру шеме у облику троугла, тако да у i-тој врсти има i бројева, $i = 1, 2, \ldots$:

i								сума
1				1				1
2			3		5			8
3		7		9		11		27
4	13		15		17		19	64
				...				

Потребно је израчунати суму бројева у i-тој врсти.

Израчунавши суме у првих неколико врста, запажамо да је сума једнака кубу редног броја врсте. Да се ово тврђење докаже индукцијом, довољно је доказати да је разлика збирова у $(i+1)$-ој и i-тој врсти једнака $(i+1)^3 - i^3 = 3i^2 + 3i + 1$. Даље, ако са a_i означимо последњи број у i-тој врсти, разлика збирова у $(i+1)$-ој и i-тој врсти једнака је $i \cdot 2i + a_{i+1}$ јер је сваки од првих i бројева у $(i+1)$-ој врсти већи од одговарајућег у i-тој врсти за $2i$ (пошто у i-тој врсти има i узастопних непарних бројева, разлика првог у $(i+1)$-ој врсти и првог у i-тој врсти је $2i$). Дакле, да би се доказало друго тврђење, довољно је доказати да је $2i^2 + a_{i+1} = 3i^2 + 3i + 1$, односно $a_{i+1} = i^2 + 3i + 1 = (i+1)^2 + (i+1) - 1$, или $a_i = i^2 + i - 1$. Ово последње тврђење ћемо наравно доказати индукцијом. За $i = 1$ је $a_1 = 1 = 1^2 + 1 - 1$, тј. тврђење је тачно. Из претпоставке да је за неко i $a_i = i^2 + i - 1$ следи (јер у $(i+1)$-ој врсти има $i+1$ узастопних непарних бројева) да је $a_{i+1} = a_i + 2(i+1) = i^2 + 3i - 1 = (i+1)^2 + (i+1) - 1$, чиме је доказано последње помоћно тврђење, а самим тим и полазно тврђење.

У следећем примеру видећемо да се у доказу индуктивна хипотеза мора користити еластично, а не онако како би на први поглед изгледало да треба учинити.

Теорема 1.8. *За сваки природан број* n *је* $\frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^n} < 1$.

Доказ. За $n = 1$ је ово очигледно тачно. Неједнакост

$$(\frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^n}) + \frac{1}{2^{n+1}} < 1$$

не следи из претходне, јер ако је збир у загради мањи од 1, он после повећавања за $\frac{1}{2^{n+1}}$ не мора да *остане* мањи од 1. Проблеми се избегавају ако се индуктивна хипотеза примени на последњих n сабирака у овој неједнакости:

$$\frac{1}{2} + (\frac{1}{4} + \ldots + \frac{1}{2^n} + \frac{1}{2^{n+1}}) = \frac{1}{2} + \frac{1}{2}(\frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^n}) < \frac{1}{2} + \frac{1}{2} \cdot 1 = 1.$$

□

1.6. Ојлерова формула

Сада ћемо доказати тврђење познато као Ојлерова формула (L. Euler , 1707-1783.). Посматрајмо повезану планарну мапу са V чворова (темена), E

грана (ивица) и F број области на које је раван подељена ивицама; једна од области је и спољашња, неограничена област. Интуитивно, мапа је повезана ако се састоји од једног дела. Специјално, повезана планарна мапа са једном облашћу ($F = 1$) зове се **стабло** На слици 1 приказани су примери мапе (са $V = 4$ чвора, $E = 5$ грана и $F = 3$ области), односно стабла.

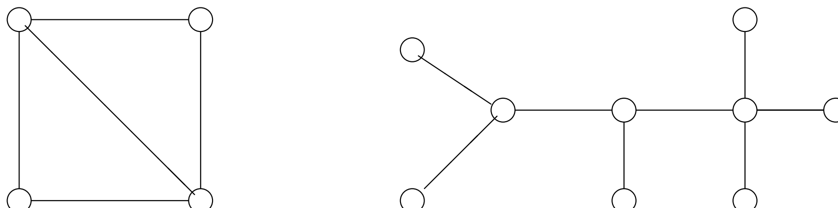


Рис. 1. Пример планарне мапе, односно стабла.

Теорема 1.9 (Ојлерова формула)**.** *За повезану планарну мапу важи једнакост* $V + F = E + 2$.

Доказательство. Доказ ће бити изведен двоструком индукцијом: најпре по броју области F, а у оквиру тога по броју чворова V. Посматрајмо најпре случај $F = 1$. Тада мапа (у овом случају стабло) не садржи ни једну затворену област, јер би у противном због постојања спољашње бесконачне области било $F \geq 2$. Треба доказати да тада важи $V = E + 1$. Ово тврђење доказаћемо индукцијом по броју чворова $V = n$. Ако је $n = 1$ онда нема грана и тврђење је тачно. Нека је тврђење тачно за $V = n$. Посматрајмо произвољно стабло са $n+1$ чворова. То стабло мора да има бар један чвор v са само једним суседним чвором, јер би у противном мапа садржала затворену област. Мапа, која се од ове добија уклањањем чвора v и гране e која га повезује са суседним чвором, повезана је и нема затворених области, а број чворова у њој је n. За ту мапу важи индуктивна хипотеза: $(V - 1) = (E - 1) + 1$, па је $V = E + 1$, чиме је завршен доказ теореме за случај $F = 1$.

Претпоставимо да је тврђење теореме тачно за планарне мапе са n области, $V + n = E + 2$. Произвољна мапа са $n + 1$ области има бар једну област која је суседна са спољашњом облашћу. Избацивањем из мапе неке од грана које одвајају ту област од спољашње, добија се мапа која је и даље повезана и планарна, при чему су број области F и грана E смањене у односу на полазне вредности за по један. На основу индуктивне хипотезе за нову мапу важи $V + (F - 1) = (E - 1) + 2$, из чега непосредно следи да за полазну мапу важи тврђење теореме. □

1.7. Грејови кодови

Грејов код дужине $n > 1$ је низ k-торки бита ($k \geq 1$) $s_1, s_2, \ldots, s_n$ таквих да се у том низу свака два узастопна члана, као и s_1, s_n, разликују на тачно

једној позицији. На пример, $00, 01, 11, 10$ је Грејов код дужине 4. Због ове своје особине Грејови кодови имају широку примену. Лако је видети да не постоје Грејови кодови непарне дужине. Заиста, s_1 и s_i се за $i > 1$ разликују на парном, односно непарном броју позиција ако је i непарно, односно парно. Због тога се s_1 и s_n за непарно n разликују на парном, (дакле различитом од један) броју позиција.

Теорема 1.10. *За сваки паран број n постоји Грејов код дужине n.*

Доказательство. Грејов код дужине два је, на пример, низ $0, 1$. Ако постоји Грејов код $s_1, s_2, \ldots, s_n$ дужине $n = 2k$ за неко $k \in N$ (односно s_i, s_{i+1}, $i = 1, 2, \ldots, n-1$, као и s_n, s_1, разликују се на тачно једној позицији), онда се непосредно проверава да је $0s_1, 1s_1, 1s_2, 0s_2, 0s_3, \ldots, 0s_n$ Грејов код дужине $n+2$, видети слику 2. Овде су са $0s_i$ односно $1s_i$ означени блокови бита добијени од s_i додавањем нуле, односно јединице са леве стране. □

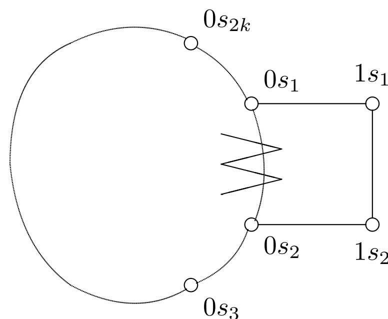


Рис. 2. Продужавање Грејовог кода за два

Описана конструкција Грејових кодова парне дужине има важан недостатак: за код дужине $2k$ користе се k-торке бита, тј. дужине блокова бита повећавају се за један сваки пут кад се дужина кода повећа за два. С друге стране, ако не би постојао захтев о односу суседних чланова низа, низ дужине n могао би се кодирати k-торкама бита, где је $k = \lceil \log_2 n \rceil$; са $\lceil x \rceil$ означен најмањи цео број већи или једнак од реалног броја x. Нерационалност описане конструкције може се исправити тако што би се повећавањем дужине блокова за један конструисао два пута дужи, а не Грејов код дужи за два:

$$0s_1, 0s_2, \ldots, 0s_n, 1s_n, 1s_{n-1}, \ldots, 1s_1, \tag{1.1}$$

видети слику 3.

Назовимо низ блокова бита $s_1, s_2, \ldots, s_n$ — код кога се суседни блокови бита s_i, s_{i+1} за свако $i = 1, 2, \ldots, n-1$ разликују на тачно једној позицији — затвореним, односно отвореним Грејовим кодом ако јесте, односно није испуњен услов да се блокови s_1, s_n разликују на тачно једној позицији. Доказ следеће теореме даје конструкцију Грејовог кода са минималном дужином блокова бита.

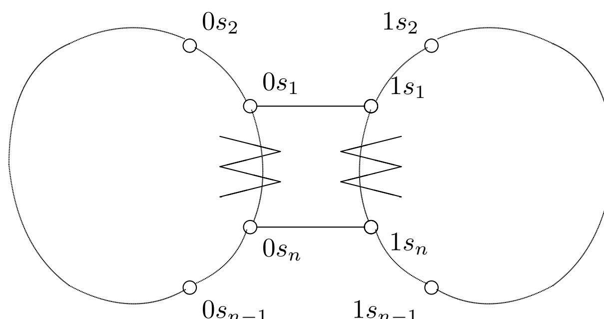


Рис. 3. Удвостручавање дужине Грејовог кода

Теорема 1.11. *За свако $n > 1$ постоји Грејов код дужине n са $\lceil \log_2 n \rceil$ бита, и то затворен ако је n парно, односно отворен ако је n непарно.*

Доказательство. Очигледно је да постоје Грејови кодови дужине један и два са блоковима дужине један. Претпоставимо да постоје отворени, односно затворени Грејови кодови свих дужина $k < n$, са величином блока $\lfloor \log_2 k \rfloor$. Ако је n парно, онда се применом конструкције (1.1) (слика 3) на код дужине $n/2$ са $\lceil \log_2(n/2) \rceil$ бита (који по индуктивној хипотези постоји), добија код дужине n са $1 + \lceil \log_2(n/2) \rceil = \lceil \log_2 n \rceil$ бита.

Нека је сад $n = 2m + 1$ непарно. Према индуктивној хипотези постоји Грејов код дужине m са $\lceil \log_2 m \rceil$ бита. Од њега се на исти начин добија код дужине $2m$ са $d = 1 + \lceil \log_2 m \rceil = \lceil \log_2(2m) \rceil$ бита. Ако $2m$ није степен двојке, онда постоје неискоришћени блокови (тј. блокови који не припадају коду) од d бита. Бар један од њих (нека је то код s') суседан је неком од блокова $1s_i$ (односно разликује се од њега на тачно једној позицији). Због тога се може (видети слику 4) конструисати отворени код дужине

$$d = \lceil \log_2(2m) \rceil = \lceil \log_2(2m+1) \rceil = \lceil \log_2 n \rceil.$$

У противном, ако $2k$ јесте степен двојке, онда су сви кодови дужине d искоришћени, па се могу најпре све кодне речи продужити дописивањем нуле спреда. Тако се појављују неискоришћени кодови (они који почињу јединицом), и случај је сведен на претходни, при чему је број бита у кодним речима

$$\lceil \log_2(2m) \rceil + 1 = \log_2(2m) + 1 = \lceil \log_2(2m+1) \rceil = \lceil \log_2 n \rceil.$$

□

1.8. Налажење дисјунктних путева у графу

Нека је $G = (V, E)$ *неусмерени* (граф за кога код грана као парова чворова није битан редослед) *повезан* граф. *Степен* чвора је број грана које су му суседне. Ови појмови објашњени су у уводном одељку поглавља 6 о графовима, а користе се у овом интересантном примеру. Нека је O скуп чворова непарног

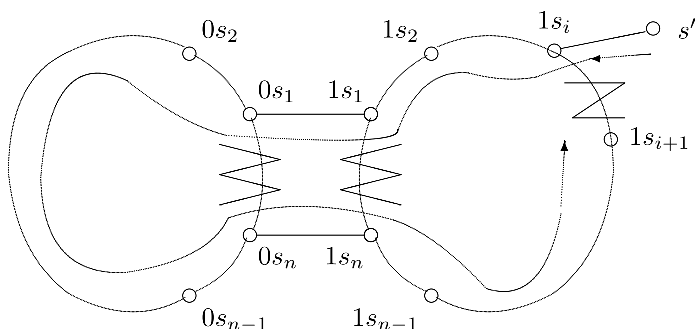


Рис. 4. Конструкција отвореног Грејовог кода

степена; лако је видети да је број таквих чворова увек паран. Заиста, сабирање степена чворова је исто што и бројање грана, при чему се свака грана броји тачно два пута. Пошто је дакле збир степена свих чворова, па и чворова из скупа O, паран број, број чворова у O мора бити паран. За скуп O важи следеће тврђење.

Теорема 1.12. *Чворови из скупа O чворова непарног степена у неусмереном повезаном графу $G = (V, E)$ могу се поделити на парове, тако да се могу наћи дисјунктни (без заједничких грана) путеви који повезују први и други чвор у сваком пару.*

Доказательство. Да би се ово тврђење доказало индукцијом, можемо да претпоставимо (индуктивна хипотеза) да је оно тачно за графове са мање од m грана. Ако у скупу O има више од два чвора, изаберу се међу њима произвољна два. Пошто је граф повезан, у њему постоји пут (низ грана) који повезује два изабрана чвора. Уклањањем ових грана из графа добија се граф са мање грана. На овај граф се међутим не може применити индуктивна хипотеза. Проблем је у томе што је могуће да се вађењем ових грана добије неповезан граф (видети пример на слици 5).

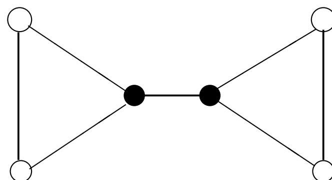


Рис. 5. Пример графа који уклањањем гране постаје неповезан.

Решење проблема на који смо наишли је на први поглед необично. Састоји се у појачању тврђења теореме (*појачавање индуктивне хипотезе*): покушаћемо да докажемо да тврђење важи не само за повезане, него и за произвољне

неусмерене графове. Изложени доказ треба променити само у делу који се односи на избор два чвора непарног степена: њих треба бирати тако да припадају једној истој *компоненти повезаности* (повезаном делу, "острву") графа (што је могуће, јер је број чворова непарног степена у свакој компоненти повезаности паран број), што обезбеђује постојање пута у графу између та два чвора. Граф који се добија удаљавањем грана пута, без обзира на евентуално повећање броја компоненти повезаности, има мањи број грана од m, па за њега важи (појачана) индуктивна хипотеза, тј. преостали чворови из O могу се поделити на парове, који се могу повезати дисјунктним путевима. □

Идеја која је овде примењена, појачање индуктивне хипотезе, често се користи. Заснована је на привидном парадоксу да се (врло често) јаче тврђење лакше доказује.

1.9. Неједнакост између аритметичке и геометријске средине

Као пример тврђења које се може доказати применом регресивне индукције искористићемо неједнакост између аритметичке и геометријске средине.

Теорема 1.13. *Нека су $x_1, x_2, \ldots, x_n$ позитивни реални бројеви. Тада је*

$$\sqrt[n]{x_1 x_2 \ldots x_n} \leq \frac{x_1 + x_2 + \cdots + x_n}{n} \tag{1.2}$$

Доказательство. Тврђење ћемо доказати применом регресивне индукције. Најпре ћемо доказати да оно важи за бројеве облика $n = 2^k$, дакле за произвољно велике бројеве, а затим да из претпоставке да оно важи за неко n следи да оно важи за $n-1$.

У првом делу доказа база индукције ($n = 2^1$) је неједнакост

$$\sqrt{x_1 + x_2} \leq \frac{1}{2}(x_1 + x_2),$$

која је последица очигледне неједнакости $(x_1 - x_2)^2 \geq 0$ (одакле се добија $x_1^2 + 2x_1x_2 + x_2^2 \geq 4x_1x_2$, односно $\frac{1}{4}(x_1 + x_2)^2 \geq x_1x_2$). Из претпоставке да (1.2) важи за $n = 2^k$ следи да за $2n = 2^{k+1}$ бројева $x_1, x_2, \ldots, x_{2n}$ важи

$$\sqrt[2n]{\prod_{i=1}^{2n} x_i} = \sqrt{\sqrt[n]{\prod_{i=1}^{n} x_i} \sqrt[n]{\prod_{i=n+1}^{2n} x_i}} \leq \sqrt{\Big(\frac{1}{n}\sum_{i=1}^{n} x_i\Big)\Big(\frac{1}{n}\sum_{i=n+1}^{2n} x_i\Big)} \leq$$

$$\leq \frac{1}{2}\Big(\frac{1}{n}\sum_{i=1}^{n} x_i + \frac{1}{n}\sum_{i=n+1}^{2n} x_i\Big) = \frac{1}{2n}\sum_{i=1}^{2n} x_i,$$

односно тврђење (1.2) важи за двоструко већи број бројева x_i. Тиме је завршен први део доказа.

Сада ћемо показати да ако једнакост (1.2) важи за произвољних n бројева, онда она важи и за произвољних $n-1$ бројева. Да би се у (1.2) смањио број променљивих, једну од њих треба везати за остале, односно изразити је

преко њих. Нека је дакле $x_n = \frac{1}{n-1}\sum_{i=1}^{n-1} x_i$, тј. x_n бирамо тако да буде једнако аритметичкој средини остале $n-1$ променљиве. Тада десна страна неједнакости (1.2) постаје једнака x_n:

$$\frac{1}{n}\sum_{i=1}^{n} x_i = \frac{n-1}{n}\Big(\frac{1}{n-1}\sum_{i=1}^{n-1} x_i\Big) + \frac{1}{n}x_n = \frac{n-1}{n}x_n + \frac{1}{n}x_n = x_n,$$

па (1.2) прелази у $x_n \prod_{i=1}^{n-1} x_i \le x_n^n$, односно

$$\sqrt[n-1]{\prod_{i=1}^{n-1} x_i} \le x_n = \frac{1}{n-1}\sum_{i=1}^{n-1} x_i.$$

Према томе, завршетком другог дела доказа доказано је да неједнакост (1.2) између аритметичке и геометријске средине важи за свако $n \in N$. □

1.10. Инваријанта петље — доказ исправности алгоритма

Математичка индукција често се користи за доказивање коректности алгоритма, посебно у случају кад се алгоритам састоји од вишеструког извршавања исте групе наредби. Идеја је да се покаже да је неко тврђење (*инваријанта петље*) тачно пре уласка у петљу и после сваког проласка кроз петљу, а да из његове тачности на крају извршења алгоритма следи коректност алгоритма.

Овај метод илустроваћемо доказом коректности алгоритма који за задати природни број n израчунава његове бинарне цифре, компоненте вектора b, тј. врши његову конверзију у бинарни облик (слика 6). Алгоритам се састоји од петље у којој су три наредбе: повећавање k (индекса за вектор b), израчунавање остатка и целог дела количника при дељењу помоћне променљиве t са два.

```
Алгоритам Bin_cifre(n);
Улаз: n (природан број)
Излаз: b (низ бинарних цифара броја n)
begin
    t := n; {помоћна променљива t са почетном вредношћу n}
    k := 0;
    while t > 0 do
        k := k + 1;
        b[k] := t mod 2;
        t := t div 2;
end
```

Рис. 6. Алгоритам за израчунавање бинарних цифара броја.

Теорема 1.14. *По завршетку алгоритма Bin_cifre у вектору b налазе се бинарне цифре броја n.*

Доказательство. Доказ исправности алгоритма биће спроведен помоћу инваријанте петље. Размотримо следећу индуктивну хипотезу.

Индуктивна хипотеза. Нека је m број чије су бинарне цифре бити добијени у k првих пролазака кроз петљу, при чему је $b[1]$ најнижи бит. Тада је

$$t \cdot 2^k + m = n.$$

За $k = 0$ је $t = n$ и $m = 0$, па је ово тврђење тачно. Нека је индуктивна хипотеза тачна за неко $k \geq 0$. У наредном проласку кроз петљу добијају се нове вредности $b[k+1] = t \bmod 2$, $t' = t \operatorname{div} 2$, и $m' = b[k+1]2^k + m = (t \bmod 2)2^k + m$, па је

$$\begin{aligned} t'2^{k+1} + m' &= (t \operatorname{div} 2)2^{k+1} + (t \bmod 2)2^k + m = \\ &= 2^k\big(2 \cdot (t \operatorname{div} 2) + (t \bmod 2)\big) + m = 2^k \cdot t + m = n, \end{aligned}$$

због индуктивне хипотезе. Тиме је доказано да израз $t2^k + m$ не мења вредност од проласка до проласка кроз петљу, односно да је он инваријанта петље. Како је после последњег проласка кроз петљу $t = 0$, у том тренутку ће бити $m = n$, чиме је исправност алгоритма доказана. □

1.11. Најчешће грешке

Приликом извођења доказа индукцијом честа је грешка да се као очигледна чињеница искористи тврђење које је блиско или еквивалентно доказиваном. Тиме се фактички појачава индуктивна хипотеза, али се не доказује да из тачности појачане хипотезе за n следи њена тачност за $n + 1$.

Важно је да се при преласку са случаја n на $n + 1$ доказ изведе полазећи од произвољног случаја $n+1$. Као пример могуће грешке, размотримо следећи "доказ" Ојлерове формуле $V + F = E + 2$ индукцијом по F. Полазећи од повезане планарне мапе са $F = n$ која има бар једну унутрашњу област ($n \geq 2$), у тој области се на две ивичне гране изаберу два нова чвора и повежу граном. У новој мапи F, V и E су повећани редом за један, два и три, па дакле и за њу важи Ојлерова формула. Проблем са овим "доказом" је у томе што се описаном конструкцијом не могу добити све могуће мапе за које је $F = n + 1$: могући су и случајеви да се једна унутрашња област подели на две граном која повезује један нови и један постојећи чвор, односно граном која повезује два постојећа чвора. Претходно описаном конструкцијом се уствари могу добити само графови којима сви новододати чворови имају степен три.

Размотримо још један пример "доказа" индукцијом.

Теорема 1.15 (нетачна)**.** *Дато је n правих у равни тако да никоје две међу њима нису паралелне. Доказати да свих n правих имају заједничку тачку.*

Доказательство. (погрешан) За $n = 1$ тврђење је тачно. Шта више, оно је тачно и за $n = 2$. Нека оно важи за неко n и нека је дато произвољних $n + 1$ правих у равни таквих да никоје две међу њима нису паралелне. Тада према

индуктивној хипотези првих n правих имају заједничку тачку. Исто важи и за последњих n правих. Ове две тачке се морају поклапати, што значи да свих $n+1$ правих имају заједничку тачку. □

Грешка у овом доказу прикривена је у чињеници да прелаз са n на $n+1$ не важи за $n = 2$: од три разматране праве a, b и c пресечна тачка првих двеју, a и b, не мора да се поклапа са пресечном тачком последњих двеју, b и c.

1.12. Резиме

Математичка индукција је моћан метод за доказивање теорема. Најпре се формулише индуктивна хипотеза и бира се променљива по којој ће доказ индукцијом бити изведен. Понекад се индуктивна хипотеза може формулисати на више начина — неки воде краћем, а неки дужем доказу; понекад се уводи потпуно нова променљива, потребна само за извођење доказа. Доказ се понекад изводи индукцијом на више нивоа, од којих сваки води све ближе циљу. Појачавање индуктивне хипотезе је чест поступак, којим се доказује тврђење општије од задатог.

Доказ индукцијом састоји се од два дела — базе и корака индукције (редукције, свођења). Базу понекад није лако доказати, али је то ипак ређи случај. Због тога се овај део доказа понекад грешком игнорише. Срж доказа је у редукцији: најчешће се тврђење са параметром n своди на аналогно тврђење са параметром $n-1$. Друга могућа варијанта је потпуна индукција, кад се случај n своди на неколико других за вредности $k < n$. Такође се користи и поступак свођења са $2n$ на n, а затим са n на $n+1$, тзв. регресивна индукција. У свим варијантама је важно да се о доказиваном тврђењу не претпостави ништа више од индуктивне хипотезе.

Задаци

У неким задацима о графовима користе се појмови из уводног одељка поглавља 6.

1.1. Доказати да је за произвољне природне бројеве n, x, y, такве да је $x \neq y$, $x^n - y^n$ дељиво са $x - y$.

1.2. Одредити израз за суму $1 \cdot 2 + 2 \cdot 3 + \cdots + n \cdot (n+1)$ и доказати тачност тог израза индукцијом.

1.3. Доказати да је $1^2 - 2^2 + 3^2 - 4^2 + \cdots + (-1)^{k-1}k^2 = (-1)^{k-1}k(k+1)/2$.

1.4. Нека је A подскуп скупа $\{1, 2, \ldots, 2n\}$ који има $n+1$ елеменат. Доказати да постоје бројеви $a, b \in A$ такви да $a|b$.

1.5. Нека су a, b и n природни бројеви. Доказати да је $2^{n-1}(a^n + b^n) \geq (a+b)^n$.

1.6. Одредити збир елемената i-те врсте *Паскаловог троугла* (видети слику 7) и доказати тачност тог тврђења индукцијом. На ивицама троугла су јединице, а у унутрашњости је сваки број једнак збиру два броја изнад њега.

1.7. Доказати да за свако $n > 1$ важи неједнакост $\sum_{i=n+1}^{2n} \frac{1}{i} > \frac{13}{24}$

1.8. * Доказати да за свако $n > 1$ из једнакости $\sum_{i=1}^{n} 1/i = k/m$ (при чему су k и m су узајамно прости бројеви), следи да је k непаран, а m паран природан број.

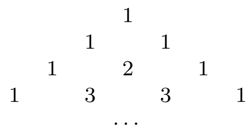


Рис. 7. Уз задатак 1.6.

1.9. Посматрајмо низ $1, 2, 3, 4, 5, 10, 20, 40, \ldots$, који почиње као аритметичка прогресија, а после првих пет чланова постаје геометријска прогресија. Доказати да се сваки природан број може представити у облику збира различитих бројева из овог низа.

1.10. Дато је $n \geq 3$ правих у равни у општем положају. Доказати да је бар једна од области које оне формирају — троугао.

1.11. Дато је n тачака у равни, таквих да за сваке три међу њима постоји круг полупречника 1 који их садржи. Доказати да постоји круг полупречника 1 који садржи свих n тачака.

1.12. Доказати да области на које раван дели n кружница са по једном повученом тетивом (слика 8) могу бити обојене са три боје, тако да су суседне области увек обојене различитим бојама.

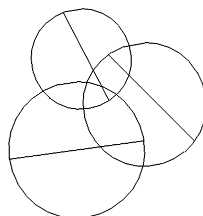


Рис. 8. Уз задатак 1.12.

1.13. Дата је планарна мапа чији су сви чворови парног степена (односно сви чворови су повезани ивицом са парним бројем других чворова). Доказати да се области на које дели раван ова мапа могу обојити са две боје тако да су суседне области увек обојене различитом бојом.

1.14. Нека су $d_1, d_2, \ldots, d_n$ природни бројеви, $n \geq 2$. Доказати да ако је $\sum_{i=1}^{n} d_i = 2n-2$ онда постоји стабло са n чворова чији су степени бројеви $d_1, d_2, \ldots, d_n$.

1.15. Нека је дато n тачака на кружници и нека су сваке две од њих спојене дужи, при чему се никоје три дужи не секу у истој тачки. На колико области ове дужи деле круг?

1.16. Нека је $G = (V, E)$ неусмерено стабло и нека је $f : V \to V$ пресликавање чворова тог стабла које задовољава следећи услов: крајеве сваке гране стабла пресликава или у један исти чвор или у два чвора који су такође крајеви гране стабла ("контракција"). Показати да у G постоји "непокретни чвор" (такав да га f пресликава у самог себе) или "непокретна грана" (грана коју f "обрће", односно f пресликава један крај гране у други и обрнуто).

1.17. Дирихлеов принцип гласи: ако је у n урни убачена $n + 1$ куглица, онда нека урна садржи више од једне куглице. Доказати овај принцип индукцијом.

1.18. **Комплетно бинарно стабло** (КБС) се индуктивно дефинише на следећи начин. КБС висине 0 састоји се од само једног чвора, корена. КБС висине $h+1$ састоји се од два КБС висине h, чији су корени синови истог чвора — који је нови корен. Нека је T КБС висине h. **Висина** чвора у T је h умањено за растојање чвора од корена; тако нпр. корен има висину h, а листови висину 0. Доказати да је сума висина свих чворова у T једнака $2^{h+1} - h - 2$.

1.19. Доказати да чланови Фибоначијевог низа ($F_1 = F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$ за $n \geq 3$) задовољавају једнакост $F_n^2 + F_{n+1}^2 = F_{2n+1}$ за $n \geq 1$.

1.20. * Нека је K_n комплетан неусмерени граф са n чворова (граф у коме су свака два чвора повезана граном), и нека је n паран број. Доказати да се гране K_n могу изделити на тачно $n/2$ повезујућих стабала (повезујуће стабло графа је његов подграф – стабло, који садржи све чворове графа).

1.21. * У неусмереном графу $G = (V, E)$ **упаривање** је скуп грана у коме никоје две гране немају заједнички чвор. **Савршено упаривање** је такво упаривање које "покрива" све чворове графа. Конструисати пример фамилије графова G_n, са $2n$ чворова и n^2 грана, $n = 1, 2, \ldots$ тако да G_n има тачно једно савршено упаривање.

1.22. Нека су $a_1, a_2, \ldots, a_n$ позитивни реални бројеви такви да је њихов производ 1. Доказати, не користећи неједнакост између аритметичке и геометријске средине, да је

$$(1 + a_1)(1 + a_2) \ldots (1 + a_n) \geq 2^n.$$

1.23. Конструисати алгоритам за конверзију бинарног броја у декадни. Улаз је низ бита b дужине k, а излаз је број n. Доказати коректност алгоритма помоћу инваријанте петље.

Глава 2

Анализа алгоритама

2.1. Увод

Циљ анализе алгоритма је да се предвиди његово понашање, посебно брзина извршавања, не реализујући га на (неком конкретном) рачунару. Идеја је дакле да се процени брзина рада без реализације алгоритма, и то тако да процена важи за сваки рачунар. Уосталом, напор да се алгоритам испита на сваком рачунару је преамбициозан, између осталог и због тога што би требало узети у обзир све могуће реализације "подалгоритама" (нпр. многи алгоритми садрже сортирање, које се може извршити различитим поступцима).

Тачно понашање алгоритма је немогуће предвидети, сем у најједноставнијим случајевима. На понашање утиче много фактора, па се у обзир узимају само главне карактеристике, а занемарују се детаљи везани за тачну реализацију. Према томе — анализа алгоритма мора да буде *приближна*. Ипак се на тај начин добијају значајне информације о алгоритму, које нпр. омогућују упоређивање различитих алгоритама за решавање истог проблема.

Логичан приступ приликом анализе брзине извршавања алгоритма је да се занемарују константни фактори, јер се брзине извршавања алгоритма на различитим рачунарима разликују се приближно за константни фактор. Интересантно је оценити понашање брзине извршавања алгоритма кад величина улаза тежи бесконачности. Ако је, на пример, улаз у алгоритам вектор дужине n, а алгоритам се састоји од $100n$ корака, онда се каже да је сложеност алгоритма n; или за сложеност $2n^2 + 50$ корака каже се да је n^2. За други од ова два алгоритма каже се да је спорији, иако за $n = 5$ важи $100n > 2n^2 + 50$; неједнакост $2n^2 + 50 > 100n$ тачна је нпр. за свако $n > 100$. На срећу, обично су константе у изразима за сложеност алгоритама мале. Због тога, иако асимптотски приступ оцењивању сложености понекад може да доведе до забуне, у пракси се добро показује и довољан је за прву апроксимацију ефикасности.

Циљ анализе, одређивање времена извршавања алгоритма за одређени улаз — практично је неостварљив (сем за најједноставније алгоритме), јер је тешко узети у обзир све могуће улазе. Због тога се за сваки могући улаз најпре дефинише његова величина (димензија) n, после чега се у анализи користи само овај податак. Величину улаза нећемо овде строго дефинисати; обично је то мера величине меморијског простора потребног за смештање описа улаза кад се употреби било које разумно кодирање нпр. битима. Непостојање опште

дефиниције величине проблема неће овде изазивати потешкоће, јер се обично упоређују различити алгоритми за решавање истог проблема.

Дакле, анализа алгоритма треба да као резултат да израз за утрошено време у зависности од n. Међутим, поставља се питање — који међу улазима величине n изабрати као репрезентативан? Често се за ову сврху бира најгори случај. Најбољи улаз је често тривијалан. Даље, средњи (просечан) улаз само на први поглед изгледа као добар избор. Међутим, анализа је тада компликована, а потешкоћу представља и дефинисање расподеле вероватноћа на скупу свих улаза величине n, и то тако да она одговара ситуацијама које се појављују у пракси. Због тога је корисно анализу вршити за најгори случај. Добијени резултати најчешће су блиски просечним, односно експерименталним резултатима; или, ако је чак најгори случај у погледу сложености битно другачији од "просечног", алгоритам који добро ради у најгорем случају обично добро ради и у просеку. Изнесени разлози оправдавају праксу да се најчешће врши анализа најгорег случаја.

Анализа асимптотског понашања сложености алгоритма, и то у најгорем случају међу улазима одређене величине — то је дакле *апроксимација* времена рада одређеног алгоритма на одређеном улазу, која ипак најчешће добро карактерише особине алгоритма.

2.2. Асимптотска ознака O

Дефиниција 2.1. Нека су f и g позитивне функције од аргумента n из скупа $\mathbf{N}$ природних бројева. Каже се да је $g(n) = O(f(n))$ ако постоје позитивне константе c и N, такве да за свако $n > N$ важи $g(n) \leq cf(n)$.

Ознака $O(f(n))$ се уствари односи на класу функција, а једнакост $g(n) = O(f(n))$ је уобичајена ознака за инклузију $g(n) \in O(f(n))$. Јасно је да је функција f само нека врста горње границе за функцију g. На пример, поред једнакости $5n^2 + 15 = O(n^2)$ (јер је $5n^2 + 15 \leq 6n^2$ за $n \geq 4$) важи и једнакост $5n^2 + 15 = O(n^3)$ (јер је $5n^2 + 15 \leq n^3$ за $n \geq 6$). Ова нотација омогућује игнорисање мултипликативних константи: уместо $O(5n + 4)$ може се писати $O(n)$. Слично, у изразу $O(\log n)$ основа логаритма није битна, јер се логаритми за различите основе разликују за мултипликативну константу:

$$\log_a n = \log_a(b^{\log_b n}) = \log_b n \cdot \log_a b.$$

Специјално, $O(1)$ је ознака за класу ограничених функција.

Познато је да свака експоненцијална функција са основом > 1 расте брже од сваког полинома. Дакле, ако је $f(n)$ монотоно растућа функција која није ограничена, $a > 1$ и $c > 0$, онда је

$$f(n)^c = O\big(a^{f(n)}\big). \tag{2.1}$$

Специјално, за $f(n) = n$ добија се $n^c = O(a^n)$, а за $f(n) = \log_a n$ добија се једнакост $(\log_a n)^c = O\big(a^{\log_a n}\big) = O(n)$, тј. произвољан степен логаритамске функције расте спорије од линеарне функције.

Лако се показује да се O-изрази могу сабирати и множити:

$$O(f(n)) + O(g(n)) \quad = \quad O(f(n) + g(n)),$$
$$O(f(n))O(g(n)) \quad = \quad O(f(n)g(n)).$$

На пример, друга од ових једнакости може се исказати на следећи начин: ако је нека функција $h(n)$ једнака производу произвољне функције $r(n)$ из класе $O(f(n))$ и произвољне функције $s(n)$ из класе $O(g(n))$, онда $h(n) = r(n)s(n)$ припада класи $O(f(n)g(n)))$. Заиста, ако је $r(n) = O(f(n))$ и $s(n) = O(g(n))$ онда постоје позитивни N_1, N_2, c_1 и c_2 такви да за $n > N_1$ важи $r(n) < c_1 f(n)$ и за $n > N_2$ важи $s(n) < c_2 g(n)$. Међутим, за $N = \max\{N_1, N_2\}$ и $c = \max\{c_1, c_2, c_1c_2\}$ важи $r(n) + s(n) < c\big(f(n) + g(n)\big)$ и $r(n)s(n) < cf(n)g(n)$, тј. $r(n) + s(n) = O\big(f(n) + g(n)\big)$ и $r(n)s(n) = O\big(f(n)g(n)\big)$. Међутим, O-изрази одговарају релацији $\leq$, па се не могу одузимати и делити. Тако нпр. из $f(n) = O(r(n))$ и $g(n) = O(s(n))$ не следи да је $f(n) - g(n) = O\big(r(n) - s(n)\big)$ (видети задатке 2.13 и 2.14).

Следећи пример илуструје зашто треба сконцентрисати пажњу на асимптотско понашање. Нека је $f(n)$ укупан број операција при извршавању неког алгоритма на улазу величине n, при чему је $f(n)$ нека од функција $\log_2 n$, n, $n \log_2 n$, $n^{1.5}$, n^2, n^3, 1.1^n. Претпоставимо да се алгоритам извршава на рачунару који извршава m операција у секунди, $m \in \{10^3, 10^4, 10^5, 10^6\}$. У табели 1 приказана су времена извршавања алгоритма на улазу фиксиране величине $n = 1000$, на рачунарима различите брзине. Види се какав је утицај константних фактора: код алгоритама који се могу извршити за прихватљиво време утицај замене рачунара бржим је приметан. Међутим, код алгоритма у последњој колони, са експоненцијалном функцијом $f(n)$, замена рачунара бржим не чини проблем решивим.

$f(n)$ / m	$\log_2 n$	n	$n \log_2 n$	$n^{1.5}$	n^2	n^3	1.1^n
10^3	0.01	1.0	10.0	32.0	1000	10^6	10^{39}
10^4	0.001	0.1	1.	3.2	100	10^5	10^{38}
10^5	0.0001	0.01	0.1	0.32	10	10^4	10^{37}
10^6	0.00001	0.001	0.01	0.032	1	10^3	10^{36}

Таблица 1. Трајање извршавања $f(n)$ операција на рачунару који извршава m операција у секунди, за $n = 1000$.

Други проблем у вези са сложеношћу алгоритма је питање доње границе за потребан број рачунских операција. Док се горња граница односи на конкретан алгоритам, доња граница сложености односи се на произвољан алгоритам из неке одређене класе. Због тога оцена доње границе захтева посебне поступке анализе. За функцију $g(n)$ каже се да је асимптотска доња граница функције $T(n)$ и пише се $T(n) = \Omega(g(n))$, ако постоје позитивни c и N, такви да за свако

$n > N$ важи $T(n) > cg(n)$. Тако је на пример $n^2 = \Omega(n^2 - 100)$, и $n = \Omega(n^{0.9})$. Види се да симбол Ω одговара релацији $\geq$. Ако за две функције $f(n)$ и $g(n)$ истовремено важи и $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$, онда оне имају исте асимптотске брзине раста, што се означава са $f(n) = \Theta(g(n))$. Тако је на пример $5n \log_2 n - 10 = \Theta(n \log n)$, при чему је у последњем изразу основа логаритма небитна.

На крају, чињеница да је $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ означава се са $f(n) = o(g(n))$. На пример, очигледно је $\frac{n}{\log_2 n} = o(n)$, а једнакост $\frac{n}{10} = o(n)$ није тачна. Однос између степене и експоненцијалне функције (2.1) може се прецизирати: ако је $f(n)$ монотоно растућа функција која није ограничена, $a > 1$ и $c > 0$, онда је

$$f(n)^c = o\big(a^{f(n)}\big).$$

2.3. Временска и просторна сложеност

Оцена (временске) сложености алгоритма састоји се у бројању рачунских корака које треба извршити. Међутим, термин рачунска операција може да подразумева различите операције, на пример сабирање и множење, чије извршавање траје различито време. Различите операције се могу посебно бројати, али је то обично компликовано. Поред тога, време извршавања зависи и од конкретног рачунара, изабраног програмског језика, односно преводиоца. Зато се обично у оквиру алгоритма издваја неки основни корак, онај који се најчешће понавља. Тако, ако се ради о сортирању, основни корак је упоређивање. Ако је број упоређивања $O(f(n))$, а број осталих операција је пропорционалан броју упоређивања, онда је $O(f(n))$ граница временске сложености алгоритма.

Под просторном сложеношћу алгоритма подразумева се величина меморије потребне за за извршавање алгоритма, при чему се простор за смештање улазних података најчешће не рачуна То омогућује упоређивање различитих алгоритама за решавање истог проблема. Као и код временске сложености, и за просторну сложеност се најчешће тражи њено асимптотско понашање у најгорем случају, за велике величине проблема. Просторна сложеност $O(n)$ значи да је за извршавање алгоритма потребна меморија пропорционална оној за смештање улазних података. Ако је пак просторна сложеност алгоритма $O(1)$, онда то значи да је потребан меморијски простор за његово извршавање ограничен константом, без обзира на величину улаза.

Бројање основних корака у алгоритму најчешће није једноставан посао. Размотрићемо неколико типичних ситуација на које се наилази.

2.4. Сумирање

Ако се алгоритам састоји од делова који се извршавају један за другим, онда је његова сложеност једнака збиру сложености делова. Међутим, ово сабирање није увек једноставно. На пример, ако је основни део алгоритма петља у којој индекс i узима вредности $1, 2, \ldots, n$, а извршавање i-тог проласка кроз петљу троши $f(i)$ корака, онда је временска сложеност алгоритма $\sum_{i=1}^{n} f(i)$. Размотрићемо неке примере сумирања.

Пример 2.1. Ако је $f(i) = i$, односно у i-том проласку кроз петљу извршава се i корака, онда је укупан број корака $S(n) = \sum_{i=1}^{n} i = \frac{1}{2}n(n+1)$, што смо видели у поглављу 1 (теорема 1.4). Ако ово упоредимо са ситуацијом у којој се у сваком проласку кроз петљу извршава n корака, односно укупно n^2 корака, видимо да је у првом случају сложеност приближно два пута мања.

Пример 2.2 (Степене суме)**.** Ако са $S_k(n)$ означимо општију суму k-тих степена првих n природних бројева, $\sum_{i=1}^{n} i^k$, за израчунавање сума $S_k(n)$ може се искористити рекурентна релација

$$S_k(n) = -\frac{1}{k+1}\sum_{j=0}^{k-1}\binom{k+1}{j}S_j(n) + \frac{1}{k+1}\big((n+1)^{k+1} - 1\big). \tag{2.2}$$

Ова једнакост добија се сумирањем очигледне једнакости

$$i^k = -\frac{1}{k+1}\sum_{j=0}^{k-1}\binom{k+1}{j}i^j + \frac{1}{k+1}\big((i+1)^{k+1} - i^{k+1}\big), \quad i = 1, 2, \dots, n.$$

Полазећи од $S_0(n) = n$, $S_1(n) = n(n+1)/2$, за $k = 2$ се добија

$$\begin{aligned} S_2(n) = \sum_{i=1}^{n} i^2 &= -\frac{1}{3}\big(S_0(n) + 3S_1(n)\big) + \frac{1}{3}\big((n+1)^3 - 1\big) = \\ &= \frac{1}{3}\left(n^3 + 3n^2 + 3n - n - 3\frac{n(n+1)}{2}\right) = \frac{1}{6}(n^3 + 3n^2 + n) = \\ &= \frac{1}{6}n(n+1)(2n+1). \end{aligned}$$

Пример 2.3. Сума геометријске прогресије

$$F(n) = \sum_{i=0}^{n} q^i = \frac{q^{n+1} - 1}{q - 1}$$

добија се одузимањем једнакости $F(n) = \sum_{i=0}^{n} q^i$ од $qF(n) = \sum_{i=0}^{n} q^{i+1} = \sum_{i=1}^{n+1} q^i$:

$$(q-1)F(n) = \sum_{i=1}^{n+1} q^i - \sum_{i=0}^{n} q^i = q^{n+1} - 1.$$

Пример 2.4. Полазећи од израза за суму геометријске прогресије може се израчунати сума $G(n) = \sum_{i=1}^{n} i2^i$. Диференцирањем, па множењем са x једнакости

$$\sum_{i=0}^{n} x^i = (x^{n+1} - 1)/(x - 1)$$

добија се

$$\sum_{i=1}^{n} ix^i = x\frac{(n+1)x^n(x-1) - (x^{n+1} - 1)}{(x-1)^2}.$$

Одатле се за $x = 2$ добија $G(n) = (n-1)2^{n+1} + 2$.

Пример 2.5. Свођењем на претходну може се израчунати сума

$$H(n) = \sum_{i=1}^{n} i2^{n-i}.$$

Заиста, сменом индекса сумирања $j = n - i$, $i = n - j$, при чему j пролази скуп вредности $n - 1, n - 2, \ldots, n - n = 0$, добија се

$$H(n) = \sum_{j=0}^{n-1}(n-j)2^j = n\sum_{j=0}^{n-1}2^j - \sum_{j=0}^{n-1}j2^j = nF(n-1) - G(n-1) =$$
$$= n(2^n - 1) - (n-2)2^n - 2 = 2^{n+1} - n - 2.$$

2.5. Диференцне једначине

Ако за чланове низа F_n, $n = 1, 2, \ldots$, важи једнакост

$$F_n = f(F_{n-1}, F_{n-2}, \ldots, F_1, n), \tag{2.3}$$

онда се каже да низ F_n задовољава *диференцну једначину* (или рекурентну релацију) (2.3). Специјално, ако се за неко $k \geq 1$ члан F_n изражава преко k претходних чланова низа,

$$F_n = f(F_{n-1}, F_{n-2}, \ldots, F_{n-k}, n), \tag{2.4}$$

онда је *k ред* те диференцне једначине. Математичком индукцијом се непосредно доказује да је диференцном једначином 2.4 и са првих k чланова F_1, F_2, ..., F_k низ F_n једнозначно одређен. Ако неки низ G_n задовољава исту диференцну једначину $G_n = f(G_{n-1}, G_{n-2}, \ldots, G_{n-k}, n)$, и важи $F_i = G_i$ за $i = 1, 2, \ldots, k$, онда за свако $n \geq 1$ важи $F_n = G_n$.

Једна од најпознатијих диференцних једначина је она која дефинише Фибоначијев низ,

$$F_n = F_{n-1} + F_{n-2}, \quad F_1 = F_2 = 1. \tag{2.5}$$

Да би се на основу ње израчунала вредност F_n потребно је извршити $n-2$ корака (сабирања), па је овакав начин израчунавања F_n непрактичан за велике n. Тај проблем је заједнички за све низове задате диференцним једначинама. Израз који омогућује директно израчунавање произвољног члана низа (дакле не преко претходних) зове се *решење диференцне једначине.* На диференцне једначине се често наилази при анализи алгоритама. Због тога ћемо размотрити неколико метода за њихово решавање.

2.5.1. Доказивање претпостављеног решења диференцне једначине.

Често се диференцне једначине решавају тако да се претпостави облик решења, или чак тачно решење, после чега се после евентуалног прецизирања решења индукцијом доказује да то јесте решење. При томе је често други део посла, доказивање, лакши од првог — погађања облика решења.

Пример 2.6. Посматрајмо диференцну једначину

$$T(2n) = 2T(n) + 2n - 1, \quad T(2) = 1. \tag{2.6}$$

Ова диференцна једначина дефинише вредности $T(n)$ само за индексе облика $n = 2^m$, $m = 1, 2, \ldots$. Поставимо себи скромнији циљ — проналажење неке горње границе за $T(n)$, односно функције $f(n)$ која задовољава услов $T(n) \leq f(n)$ (за индексе n који су степен двојке, што ће се на даље подразумевати), али тако да процена буде довољно добра. Покушајмо најпре са функцијом $f(n) = n^2$. Очигледно је $T(2) = 1 < f(2) = 4$. Ако претпоставимо да је $T(i) \leq i^2$ за $i \leq n$ (индуктивна хипотеза), онда је

$$T(2n) = 2T(n) + 2n - 1 \leq 2n^2 + 2n - 1 = 4n^2 - \big(2n(n-1) + 1\big) < 4n^2,$$

па је $f(n)$ горња граница за $T(n)$ за све (дозвољене) вредности n. Међутим, из доказа се види да је добијена граница груба: у односу на члан $4n^2$ одбачен је члан приближно једнак $2n^2$ за велике n. С друге стране, ако се претпостави да је горња граница линеарна функција $f(i) = ci$ за $i \leq n$, онда би требало доказати да она важи и за $i = 2n$:

$$T(2n) = 2T(n) + 2n - 1 \leq 2cn + 2n - 1.$$

Да би последњи израз био мањи од $c \cdot 2n$, морало би да важи $2n - 1 \leq 0$, што је немогуће за $n \geq 1$. Закључујемо да $f(n) = cn$ не може да буде горња граница за $T(n)$, без обзира на вредност константе c. Требало би наћи израз који је по асимптотској брзини раста између n и n^2. Испоставља се да је добра граница задата функцијом $f(n) = n \log_2 n$: $T(2) = 1 \leq f(2) = 2 \log_2 2 = 2$, а ако је $T(i) \leq f(i)$ тачно за $i \leq n$, онда је

$$T(2n) = 2T(n) + 2n - 1 \leq 2n \log_2 n + 2n - 1 = 2n \log_2(2n) - 1 < 2n \log_2(2n).$$

Дакле, $T(n) \leq n \log_2 n$ је тачно за свако n облика 2^k. На основу чињенице да је у овом доказу при преласку са n на $2n$ занемарен члан 1 у односу на $2n \log_2(2n)$, закључује се да је процена овог пута довољно добра.

"Диференцна неједначина"

$$T(2n) \leq 2T(n) + 2n - 1, \quad T(2) = 1$$

дефинише само горње границе за $T(n)$, ако је n степен двојке. Свака горња граница решења (2.6) је решење ове диференцне неједначине — докази индукцијом су практично идентични. Ова диференцна неједначина може се, као и (2.6), проширити тако да одређује горњу границу за $T(n)$ за сваки природан број n:

$$T(n) \leq 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n - 1, \quad T(1) = 0.$$

Може показати да за низ $T(n)$, који задовољава овај услов, неједнакост $T(n) \leq n \log_2 n$ важи за свако $n \geq 1$. Заиста, она је тачна за $n = 1$, а из претпоставке да је она тачна за бројеве мање од неког природног броја m следи да је за $m = 2k$

$$T(2k) \leq 2T(k) + 2k - 1 \leq 2k \log_2 k + 2k - 1 = 2k \log_2(2k) - 1 < (2k) \log_2(2k)$$

односно за $m = 2k + 1$

$$T(2k+1) \le 2T(k) + 2k \le 2k \log_2 k + 2k = 2k \log_2 (2k) < (2k+1) \log_2 (2k+1),$$

тј. та неједнакост је тада тачна и за $n = m$. Дакле, та неједнакост је доказана индукцијом.

2.5.2. Линеарне диференцне једначине. Размотрићемо сада хомогене линеарне диференцне једначине облика

$$F(n) = a_1 F(n-1) + a_2 F(n-2) + \cdots + a_k F(n-k). \tag{2.7}$$

Решења се могу тражити у облику

$$F(n) = r^n, \tag{2.8}$$

где је r погодно изабрани (комплексни) број. Заменом у (2.7) добија се услов који r треба да задовољи:

$$r^k - a_1 r^{k-1} - a_2 r^{k-2} - \cdots - a_k = 0,$$

такозвана *карактеристична једначина* линеарне диференцне једначине (2.7); полином са леве стране ове једначине је *карактеристични полином* ове линеарне диференцне једначине. У случају кад су сви корени карактеристичног полинома $r_1, r_2, \ldots, r_k$ различити, добијамо k решења (2.7) облика (2.8). Тада се свако решење може представити у облику

$$F(n) = \sum_{i=1}^{k} c_i r_i^n$$

(ово тврђење наводимо без доказа). Ако је првих k чланова низа $F(n)$ задато, решавањем система од k линеарних једначина добијају се коефицијенти c_i, $i = 1, 2, \ldots, k$, а тиме и тражено решење.

Пример 2.7. Општи члан низа задатог диференцном једначином $F_n = F_{n-1} + F_{n-2}$, и почетним условима $F_1 = F_2 = 1$ (Фибоначијевог низа) може се одредити на описани начин. Корени карактеристичне једначине $r^2 - r - 1 = 0$ су $r_1 = (1+\sqrt{5})/2$ и $r_2 = (1-\sqrt{5})/2$, па је решење ове диференцне једначине облика $F_n = c_1 r_1^n + c_2 r_2^n$. Коефицијенти c_1 и c_2 одређују се из услова $F_1 = F_2 = 1$, или нешто једноставније из услова $F_0 = F_2 - F_1 = 0$ и $F_1 = 1$:

$$\begin{aligned} c_1 \quad &+ c_2 \quad = 0 \\ c_1 r_1 \quad &+ c_2 r_2 \quad = 1. \end{aligned}$$

Решење овог система једначина је $c_1 = -c_2 = \frac{1}{\sqrt{5}}$, па је општи члан Фибоначијевог низа задат са

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right). \tag{2.9}$$

У општем случају, ако су корени карактеристичне једначине $r_1, r_2, \dots, r_s$ вишеструкости редом $m_1, m_2, \dots, m_s$ ($\sum m_i = k$), онда се свако решење једначине (2.7) може представити у облику

$$F(n) = \sum_{i=1}^{s} h_i(n),$$

где је

$$h_i(n) = \left(C_{i0} + C_{i1}n + C_{i2}n^2 + \cdots + C_{i,m_i-1}n^{m_i-1}\right) r_i^n$$

(ово тврђење такође наводимо без доказа).

Пример 2.8. Карактеристични полином диференцне једначине

$$F(n) = 6F(n-1) - 12F(n-2) + 8F(n-3)$$

је $r^3 - 6r^2 + 12r - 8 = (r-2)^3$, са троструким кореном $r_1 = 2$. Због тога су сва решења ове диференцне једначине облика $F(n) = (c_0 + c_1 n + c_2 n^2)2^n$, где су c_0, c_1, c_2 константе које се могу одредити ако се знају нпр. три прва члана низа $F(1)$, $F(2)$ и $F(3)$.

2.5.3. Диференцне једначине које се решавају сумирањем. Често се наилази на линеарну нехомогену диференцну једначину облика

$$F(n+1) - F(n) = f(n), \tag{2.10}$$

при чему $F(0)$ има задату вредност. Другим речима, разлика два узастопна члана траженог низа једнака је датој функцији од индекса n. Решавање овакве диференцне једначине своди се на израчунавање суме $\sum_{i=0}^{n-1} f(i)$. Заиста, ако у горњој једнакости n заменимо са i и извршимо сумирање њене леве и десне стране по i у границама од 0 до $n-1$, добијамо

$$\sum_{i=0}^{n-1} f(i) = \sum_{i=0}^{n-1} F(i+1) - \sum_{i=0}^{n-1} F(i) = \sum_{i=1}^{n} F(i) - \sum_{i=0}^{n-1} F(i) = F(n) - F(0).$$

Према томе, решење диференцне једначине (2.10) дато је изразом

$$F(n) = \sum_{i=0}^{n-1} f(i) + F(0). \tag{2.11}$$

Пример 2.9. Размотримо диференцну једначину $F(n+1) = F(n) + 2n + 1$ са почетним условом $F(0) = 0$. Њено решење је

$$F(n) = \sum_{i=0}^{n-1}(2i+1) = 2\sum_{i=0}^{n-1} i + \sum_{i=0}^{n-1} 1 = n(n-1) + n = n^2$$

У случају кад је тешко израчунати суму $\sum_{i=0}^{n-1} f(i)$ преко које се изражава решење диференцне једначине (2.10), а функција $f(x)$ од реалног аргумента је

монотоно нерастућа непрекидна функција за $x \geq 0$, тада се сабирањем неједнакости

$$f(i) \geq \int_i^{i+1} f(x)\,dx \geq f(i+1), \quad 0 \leq i \leq n-1,$$

добијају неједнакости

$$\sum_{i=0}^{n-1} f(i) \geq \int_0^n f(x)\,dx \geq \sum_{i=1}^{n} f(i),$$

а тиме и границе интервала у коме лежи сума $\sum_{i=0}^{n-1} f(i)$:

$$\int_0^n f(x)\,dx + f(0) - f(n) \geq \sum_{i=0}^{n-1} f(i) \geq \int_0^n f(x)\,dx.$$

Ако је пак функција $f(x)$ монотоно неопадајућа, у овим неједнакостима само треба променити знаке "$\geq$" у "$\leq$".

2.5.4. Диференцне једначине за алгоритме засноване на декомпозицији. Претпоставимо да нам је циљ анализа алгоритма A, при чему број операција $T(n)$ при примени алгоритма A на улаз величине n (временска сложеност) задовољава диференцну једначину типа

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k, \tag{2.12}$$

при чему је $a, b, c, k \geq 0$, $b \neq 0$, и задата је вредност $T(1)$. Оваква једначина добија се за алгоритам код кога се обрада улаза величине n своди на обраду a улаза величине n/b, после чега је потребно извршити још cn^k корака да би се од парцијалних решења конструисало решење комплетног улаза величине n. Јасно је зашто се за овакве алгоритме каже да су типа "завади па владај", (енглески divide–and–conquer), како се још зову алгоритми засновани на декомпозицији. Обично је дакле у оваквим диференцним једначинама b природан број.

Теорема 2.1. *Асимптотско понашање низа $T(n)$, решења диференцне једначине* (2.12) *дато је једнакошћу*

$$T(n) = \begin{cases} O(n^{\log_b a}) & \textit{за } a > b^k \\ O(n^k \log n) & \textit{за } a = b^k \\ O(n^k) & \textit{за } a < b^k \end{cases}. \tag{2.13}$$

Доказательство. Доказ теореме биће спроведен само за подниз $n = b^m$, где је m цели ненегативни број. Помноживши обе стране једнакости (2.12) са a^{-m}/c, добијамо диференцну једначину типа (2.10)

$$t_m = t_{m-1} + q^m, \quad t_0 = \frac{1}{c}T(1),$$

где су уведене ознаке $t_m = \frac{1}{c}a^{-m}T(b^m)$ и $q = b^k/a$. Њено решење је

$$t_m = t_0 + \sum_{i=1}^{m} q^i$$

(видети 2.5.3). За $q \neq 1$ је $\sum_{i=1}^{m} q^i = (1-q^{m+1})/(1-q) - 1$, па се асимптотско понашање низа t_m описује следећим једнакостима

$$t_m = \begin{cases} O(m), & \text{за } q = 1 \\ O(1), & \text{за } 0 < q < 1 \\ O(q^m), & \text{за } q > 1 \end{cases}.$$

Пошто је $T(b^m) = ca^m t_m$, $n = b^m$, односно $m = \log_b n$, редом се за $0 < q < 1$ ($b^k < a$), $q = 1$ ($b^k = a$, односно $\log_b a = k$) и $q > 1$ ($b^k > a$) добија

$$T(n) = \begin{cases} O(a^m) = O(b^{m\log_b a}) = O(n^{\log_b a}) & \text{за } a > b^k \\ O(ma^m) = O(\log_b n \cdot n^{\log_b a}) = O(n^k \log n) & \text{за } a = b^k \\ O((aq)^m) = O(b^{mk}) = O(n^k) & \text{за } a < b^k \end{cases},$$

чиме је тврђење теореме доказано. □

Алгоритми овог типа имају широку примену због своје ефикасности, па је корисно знати асимптотско понашање решења диференцне једначине (2.12).

2.5.5. Потпуна рекурзија. Диференцна једначина најопштијег облика (2.3) зове се *потпуна рекурзија* или *диференцна једначина са потпуном историјом*. Размотрићемо два примера оваквих диференцних једначина.

Пример 2.10. Нека за $n \geq 2$ важи $T(n) = c + \sum_{i=1}^{n-1} T(i)$, при чему су c и $T(1)$ задати. Основна идеја за решавање оваквих проблема је тзв. "елиминација историје" налажењем еквивалентне диференцне једначине са "коначном историјом" (такве код које се члан низа изражава преко ограниченог броја претходних). Заменом n са $n+1$ у овој једначини добија се $T(n+1) = c + \sum_{i=1}^{n} T(i)$. Одузимањем прве од друге једнакости постиже се жељени ефекат:

$$T(n+1) - T(n) = c + \sum_{i=1}^{n} T(i) - \left(c + \sum_{i=1}^{n-1} T(i)\right) = T(n)$$

(за $n \geq 2$), и коначно

$$T(n+1) = 2T(n) = 2^2T(n-1) = \cdots = 2^{n-1}T(2) = 2^{n-1}(c + T(1)).$$

Приметимо да се једнакост $T(n+1) = 2T(n)$ логаритмовањем и сменом $t_n = \log T(n)$ своди на $t_{n+1} = t_n + \log 2$, односно диференцну једначину типа (2.10).

Пример 2.11. Размотрићемо сада сложенији пример. Диференцна једначина

$$T(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} T(i), \quad n \geq 2, \ T(1) = 0, \tag{2.14}$$

је важна, јер се до ње долази при анализи просечне сложености сортирања раздвајањем, алгоритма за уређивање задатог скупа бројева по величини (*сортирање*, видети 5.3.4). Идеја о "елиминацији историје" може се и овде применити — због тога је згодно преписати задату једначину тако да се у њој уз суму $\sum_{i=1}^{n-1} T(i)$ не појављује променљиви чинилац n:

$$nT(n) = n(n-1) + 2\sum_{i=1}^{n-1} T(i).$$

Затим ову једнакост треба одузети од оне која се од ње добија заменом n са $n+1$:

$$(n+1)T(n+1) - nT(n) = (n+1)n + 2\sum_{i=1}^{n} T(i) - n(n-1) - 2\sum_{i=1}^{n-1} T(i) = 2n + 2T(n),$$

односно

$$T(n+1) = \frac{2n}{n+1} + \frac{n+2}{n+1}T(n), \quad n \geq 2.$$

Овим је први циљ постигнут: добијена диференцна једначина повезује само два узастопна члана низа. Решавање ове диференцне једначине може се свести познат проблем (2.10) дељењем са $n+2$ и сменом $t_n = T(n)/(n+1)$:

$$\frac{T(n+1)}{n+2} - \frac{T(n)}{n+1} = \frac{2n}{(n+1)(n+2)},$$

односно

$$t_{n+1} - t_n = \frac{2n}{(n+1)(n+2)}.$$

Заменом n са i и сумирањем по i у границама од 1 до n добија се ($t_1 = 0$)

$$t_{n+1} - t_1 = \sum_{i=1}^{n} \frac{2i}{(i+1)(i+2)}. \tag{2.15}$$

Да би се израчунала сума са десне стране ове једнакости, може се њен општи члан разложити на парцијалне разломке

$$\frac{2i}{(i+1)(i+2)} = \frac{A}{i+1} + \frac{B}{i+2},$$

где се непознати коефицијенти A и B добијају тако што се у идентитету $2i = A(i+2) + B(i+1)$ стави најпре $i = -1$ (добија се $A = -2$), а онда $i = -2$ (одакле је $B = 4$). Сада је тражена сума једнака

$$\sum_{i=1}^{n} \frac{2i}{(i+1)(i+2)} = -2\sum_{i=1}^{n} \frac{1}{i+1} + 4\sum_{i=1}^{n} \frac{1}{i+2} = -2\sum_{i=2}^{n+1} \frac{1}{i} + 4\sum_{i=3}^{n+2} \frac{1}{i} =$$
$$= 2\sum_{i=3}^{n+1} \frac{1}{i} - 2 \cdot \frac{1}{2} + \frac{4}{n+2} = 2H(n+1) - 4 + \frac{4}{n+2},$$

где је са

$$H(n) = \sum_{i=1}^{n} \frac{1}{i} = \ln n + \gamma + O(\frac{1}{n}), \tag{2.16}$$

означена парцијална сума хармонијског реда; $\gamma \simeq 0.577$ је Ојлерова константа. Заменом вредности суме у (2.15) добија се решење диференцне једначине (2.14)

$$T(n+1) = \big(2H(n+1) - 4 + \frac{4}{n+2}\big)(n+2) = 2(n+1)H(n+1) - 4(n+1),$$

односно

$$T(n) = 2nH(n) - 4n = 2n\ln n + (2\gamma - 4)n + O(1) = O(n\ln n). \tag{2.17}$$

У току извођења коришћен је познати асимптотски развој (2.16). Често је корисно знати и асимптотски развој за $n!$, Стирлингову формулу

$$n! = \sqrt{2\pi n}\Big(\frac{n}{e}\Big)^n\big(1 + O(\frac{1}{n})\big),$$

односно

$$\log(n!) = O(n\log n). \tag{2.18}$$

2.6. Резиме

Циљ анализе алгоритма је предвидети његово понашање, што није лако. Основни метод који се користи је апроксимација. Занемарују се многи детаљи, а циљ је издвојити само најважније карактеристике алгоритма. Асимптотска ознака O је корисна за таква израчунавања. Тежина анализе као пратиоца процеса конструкције алгоритма не би требало да буде изговор за одустајање од ње. Битно је добити бар некакву представу о ефикасности алгоритма.

У многим случајевима, нарочито кад се користи рекурзија, наилази се на диференцне једначине. Тада обично треба израчунати неколико првих чланова низа, што може да пружи неки увид у понашање низа, односно да сугерише могући облик општег члана. Постоје и други методи за решавање диференцних једначина. На срећу, већина алгоритама на које се у пракси наилази, доводи до диференцних једначина размотрених у овом поглављу.

Задаци

2.1. Доказати да ако је $f(n) = o(g(n))$, онда је $f(n) = O(g(n))$. Да ли је тачна обрнута импликација?

2.2. Полазећи од (2.1) доказати да за произвољне константе a и b важи $(\log_2 n)^a = O(n^b)$.

2.3. Одредити тачно решење следеће диференцне једначине

$$T(n) = T(n-1) + \frac{n}{2}, \quad T(1) = 1.$$

2.4. Решити диференцну једначину

$$T(n) = 8T(n-1) - 15T(n-2), \quad T(1) = 1,\ T(2) = 4.$$

2.5. Решити диференцну једначину

$$T(n+3) = 4T(n+2) - 5T(n+1) + 2T(n), \quad T(1)=1,\ T(2)=2,\ T(3)=3.$$

2.6. Решити диференцну једначину

$$T(2n) = 2\sum_{i=1}^{n} T(2i-1),\ T(2n+1) = 2\sum_{i=1}^{n} T(2i),\ T(1)=1.$$

2.7. Проблем P_n са параметром n (n је природан број) решава се применом алгоритама A и B. Алгоритам A решава P_n ($n > 1$) применом алгоритма B на P_{n-1}, при чему се за свођење проблема троши n временских јединица. Алгоритам B решава P_n ($n > 1$) применом алгоритма A на P_{n-1}, при чему се за свођење проблема троши n временских јединица. Проблем P_1 алгоритам A решава директно за једну, а алгоритам B за две временске јединице. Израчунати време извршавања алгоритма A при решавању проблема P_n.

2.8. Доказати да низ $T(n)$, решење диференцне једначине $T(n) = 2T(\lfloor n/2 \rfloor) + 2n\log_2 n$, $T(1) = 0$, задовољава једнакост $T(n) = O(n\log^2 n)$.

2.9. Решити диференцну једначину

$$T(n) = 2T(n/2) + c\log_2 n, \quad T(1) = 0;$$

може се претпоставити да је n облика 2^k.

2.10. (а) Показати да решење диференцне једначине $T(n) \le T(\lfloor n/2 \rfloor) + 1$, $T(1) = 1$, задовољава услов $T(n) \le \lfloor \log_2 n \rfloor + 1$.

(б) Доказати да решење исте диференцне једначине са промењеним почетним условом $T(1) = 0$ задовољава неједнакост $T(n) \le \lceil \log_2 n \rceil + 1$.

2.11. Решити следећу потпуну рекурзију

$$T(n) = n + \sum_{i=1}^{n-1} T(i) \quad (n \le 2), \quad T(1) = 1.$$

2.12. Ограничавајући са горње стране чланове суме одређеним интегралима, доказати да за сваки природни број k важи $\sum_{i=1}^{n} i^k \log_2 i = O(n^{k+1}\log n)$.

2.13. Показати контрапримером да следеће тврђење није тачно: ако је $f(n) = O(s(n))$ и $g(n) = O(r(n))$, онда је $f(n) - g(n) = O(s(n) - r(n))$.

2.14. Наћи контрапример за следеће тврђење ако је $f(n) = O(s(n))$ и $g(n) = O(r(n))$, онда је $f(n)/g(n) = O(s(n)/r(n))$.

2.15. Навести пример две монотоно растуће функције (низа) $f(n)$ и $g(n)$, такве да није ни $f(n) = O(g(n))$ ни $g(n) = O(f(n))$.

2.16. Размотримо диференцну једначину $T(n) = 2T(n/2) + 1$, $T(2) = 1$. Покушавајући да докажемо да је $T(n) = O(n)$ (ограничавамо се на случај кад је n степен двојке), претпостављамо да за $k < n$ важи $T(k) \le ck$ за неку (за сада непознату) константу c, и замењујемо $T(n)$ и $T(n/2)$ у диференцној једначини редом са cn и $cn/2$. За успешан доказ корака индукције морала би да важи неједнакост $cn \ge 2c(n/2) + 1$, која, међутим, очигледно није тачна. Наћи тачно решење ове диференцне једначине (може се претпоставити да је n степен двојке) и објаснити разлог неуспеха описаног покушаја.

2.17. Одредити асимптотско понашање низа $S(n)$, решења диференцне једначине

$$S(mn) \le cm\log_2 mS(n) + O(mn), \quad S(2) = 1,$$

при чему су m и c константе.

2.18. Доказати да решење диференцне једначине $T(n) = 2T(n-c) + k$, где су c и k целобројне константе, има облик $T(n) = O(d^n)$ за неку константу d.

2.19. Следећа диференцна једначина се појављује при анализи неких алгоритама заснованих на декомпозицији улаза на делове различите величине:

$$T(n) = \sum_{i=1}^{k} a_i T(n/b_i) + cn,$$

при чему константе a_i, b_i задовољавају услов $\sum_{i=1}^{k} a_i/b_i < 1$. Одредити асимптотско понашање низа $T(n)$.

2.20. Одредити асимптотско понашање решења диференцне једначине
(а) $T(n) = 4T\left(\lceil\sqrt{n}\rceil\right) + 1$, $T(2) = 1$;
(б) $T(n) = 2T\left(\lceil\sqrt{n}\rceil\right) + 2n$, $T(2) = 1$.

2.21. Доказати да диференцна једначина $T(n) = kT(n/2) + f(n)$, $T(1) = c$, има решење

$$T(n) = n^{\log_2 k}\left(c + g(2) + g(4) + \ldots + g(n)\right),$$

где је $g(m) = f(m)m^{-\log_2 k}$. Може се претпоставити да је n степен двојке.

2.22. Одредити асимптотско понашање решења $T(n)$ диференцне једначине

$$T(n) = T(n/2) + \sqrt{n}, \quad T(1) = 1.$$

Може се сматрати да n узима само вредности једнаке степенима двојке.

2.23. Одредити асимптотско понашање решења диференцне једначине

$$T(n) = 2T\left(\left\lfloor \frac{n}{\log_2 n} \right\rfloor\right) + 3n \quad (n > 2), \quad T(1) = 1,\ T(2) = 2,$$

тј. одредити функцију $f(n)$ такву да је $T(n) = \Theta(f(n))$.

2.24. Оцењујући са горње стране чланове суме одређеним интегралима, доказати да за низ $S(n) = \sum_{i=1}^{n} \lceil \log_2(n/i) \rceil$ важи $S(n) = O(n)$.

2.25. Израчунати тачно суму $\sum_{i=1}^{n} \lfloor \log_2 \frac{n}{i} \rfloor$ ако је n степен двојке.

2.26. Фибоначијеви бројеви $F(n)$ могу се дефинисати и за негативне вредности n користећи исту дефиницију $F(n+2) = F(n+1) + F(n)$ и $F(1) = 1$, $F(0) = 0$ (тако је нпр. $F(-1) = 1$, $F(-2) = -1$, итд.). Нека је $G(n) = F(-n)$ за $n \geq 0$. Написати диференцну једначину за низ $G(n)$. Показати да је њено решење $G(n) = (-1)^{n+1}F(n)$.

Глава 3

Структуре података

Структуре података могу се схватити као елементи над којима се граде алгоритми. Због тога је њихово познавање (односно познавање техника рада са њима и познавање сложености операција са њима) неопходно за савладавање техника конструкције алгоритама. Структуре података су широко проучавана област. Овде ће бити дат само увод у основне структуре података.

Користан појам који ћемо у вези са структурама података користити је *апстрактни тип података*. У програмима се обично тачно специфицира тип сваког податка (цели, реални, логички, и слично). Међутим, у неким случајевима при конструкцији алгоритама конкретан тип података није битан. На пример, ако се ради са FIFO листом (скраћено од first–in–first–out , односно први–унутра–први–напоље) од интереса су операције уписа на листу и скидања са ње. Елементи се са листе скидају истим редом којим су у њу уписивани; пример FIFO листе је ред за карте испред биоскопа. Корисно је да се у овом случају не прецизира тип података, него само операције са њима. Апстрактни тип података над којим се могу изводити ове две операције гради структуру података коју зовемо FIFO лист[illegible] Најважнија карактеристика апстрактног типа података је дакле списак операција које се над њим могу извршавати.

Други пример апстрактног типа података је листа у којој су члановима придружени бројеви – приоритети. Скидање елемената са листе врши се редом према приоритету чланова листе, а не према редоследом уписивања. Ова структура података зове се листа са приоритет[illegible] И у овом случају непотребно је специфицирати тип самих података – чланова листе; шта више, ни тип података приоритета – довољно је да су приоритети елементи потпуно уређеног скупа.

Концентришући пажњу на природу структура података са аспекта потребних операција, а занемарујући прецизну реализацију, добија се општији алгоритам. Тако, на пример, технике реализације листе са приоритетом мало зависе од конкретно изабраног типа података. Дакле, ако уочимо да нам је потребан неки апстрактни тип података, можемо одмах да га применимо. Тиме конструкција алгоритама постаје модуларнија — састоји се од међусобно мало зависних целина.

3.1. Елементарне структуре података

Под *елементом* подразумевамо податак коме тип није прецизиран (на пример цели број, скуп целих бројева, фајл и слично). Тако, на пример, кад се ради о сортирању (уређивању елемената линеарно уређеног скупа по величини), ако су једине операције упоређивање и копирање, онда се исти алгоритам може применити и на целе бројеве и на имена — разлика је само у реализацији, а идеје на којима се заснива алгоритам су исте. Ако нас првенствено интересују идеје за конструкцију алгоритама, разумно је игнорисати конкретне типове података елемената. Ако се нпр. ради о сортирању, за елементе се претпоставља да се могу упоређивати на једнакост, да су из потпуно уређеног скупа, и да се могу копирати. Поред тога, претпоставка је се да се све операције над елементима изводе у јединичном временском интервалу.

Вектор. *Вектор* (низ) је низ елемената истог типа. Величина вектора је број елемената у њему. Величина вектора се мора унапред задати (односно фиксирана је), јер се унапред зна величина додељене меморије. На пример, вектор од 100 елемената – имена, од којих се свако записује у 8 бајтова, заузима у меморији 800 бајтова. Ако је први елеменат на локацији x, а величина елемента је a бајтова, онда је k-ти елеменат вектора на локацији $x + (k-1)a$, што се лако израчунава — независно од редног броја елемента. Ово је ефикасна и због тога често коришћена структура података. Њена основна особина је да је једнако време приступа свим елементима. На вишем програмском језику се о позицији елемента и не размишља, јер тај део посла обавља преводилац (компајлер). Закључак је да вектор треба користити кад год је то могуће. Недостаци ове структуре података су да су сви њени елементи истог типа и да се њена величина не може мењати динамички у току извршавања алгоритма.

Слог. Слог (рекорд) је такође низ елемената, који међутим не морају бити истог типа — једино се фиксира комбинација типова елемената. Величина слога такође мора унапред бити дефинисана. Елементима слога може се као и елементима вектора приступити за константно време: за ову сврху користи се посебан вектор са почетним адресама елемената слога, дужине једнаке броју елемената слога. Тако, на пример, ако је слог дефинисан (као у паскалу) на слици 1, онда је адреса елемента $Int6$ једнака $2{\cdot}4{+}3{\cdot}20{\cdot}4{+}3{\cdot}4{+}1 = 261$. Адресе овог и осталих елемената преводилац израчунава само једном, приликом проласка кроз дефиницију слога, и смешта их у споменути помоћни вектор.

Повезана листа. Често је потребно да се број елемената динамички мења. Кад се унапред не зна број елемената вектора, могуће је резервисати довољно велики вектор и тако решити проблем — што је често добро решење. Ипак, неефикасно је резервисати меморијски простор према најгорем случају. Поред тога, често је потребно уметање елемената или брисање елемената из средине листе — што је неефикасно код вектора. Због наведених разлога користе се тзв. *динамичке структуре података.* Најједноставнија од њих је *повезана*

$primer =$ **record**
$Int1 : integer;$
$Int2 : integer;$
$Ar1 : array[1..20]\ of\ integer;$
$Ar2 : array[1..20]\ of\ integer;$
$Ar3 : array[1..20]\ of\ integer;$
$Int3 : integer;$
$Int4 : integer;$
$Int5 : integer;$
$Int6 : integer;$
$Ime1 : array[1..12]\ of\ character;$
$Ime2 : array[1..12]\ of\ character$
end

Рис. 1. Пример слога.

листа. Ако је циљ ефикасно уметање и избацивање елемената, мора се напустити секвенцијални запис који карактерише вектор — елементи се морају записивати независно, а међусобно се повезују (надовезују) помоћу *показивача*, променљиве која садржи адресу неког другог елемента.

Повезана листа је вектор парова (елеменат, показивач), при чему показивач садржи адресу наредног пара. Пар се представља помоћу слога. Пролазак кроз листу могућ је једино линеарно — редом од почетка, елеменат по елеменат, пратећи показиваче. Недостаци ове структуре података су у томе што захтева више простора у меморији (уз сваки елеменат иде и показивач), а k-том ($k > 1$) елементу може се приступити само преко свих претходних. Предност повезане листе је у томе што се упис и брисање лако реализују, за разлику од вектора, код кога ове операције захтевају велики број померања елемената. За уметање новог елемента између нека два елемента повезане листе довољно је само променити два показивача. На слици 2 приказано је уметање новог, елемента 30a, између 30. и 31. елемента повезане листе. Слично, за брисање елемента из листе довољно је променити један показивач. На слици 3 приказано је брисање 31. елемента повезане листе.

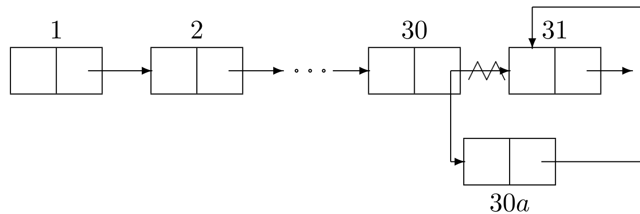


Рис. 2. Уметање новог елемента у повезану листу.

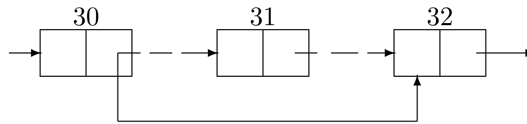


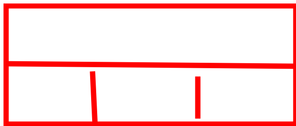
Рис. 3. Брисање елемента повезане листе.

Овде су због једноставности занемарени неки важни детаљи, који усложњавају реализацију повезане листе. Основни проблем је како открити крај листе. У ту сврху може се користити *празан показивач* **nil**, (показивач на ништа, односно вредност показивача таква да не може да представља важећу адресу), која се додељује показивачу последњег елемента у листи. Друго могуће решење је да се крај листе означи обичним слогом, чији пак податак није обичан, тзв. *празан слог.*

3.2. Стабла

Вектори и повезане листе су једнодимензионалне структуре података које одражавају само редослед елемената. Понекад је међутим потребно представити и сложеније односе између елемената. *Стабло* (дрво) је хијерархијска структура, али се може користити и за извођење неких операција над линеарним структурама. На овом месту бавићемо се само *хијерархијским или коренским* стаблом. Коренско стабло чини скуп чворова и грана, које повезују чворове на специјалан начин (видети пример на слици 4). Један чвор је издвојен, представља врх хијерархије и зове се *корен* стабла. Чворови везани са кореном чине ниво 1 хијерархије; (нови) чворови везани са чворовима нивоа 1 (сем корена) чине ниво 2 хијерархије, итд. Свака грана у стаблу повезује неки чвор са његовим (јединственим) претходником (*оцем*); једино корен нема оца. Основна карактеристика стабла је да нема циклуса (затворених путева), због чега између свака два његова чвора постоји јединствени пут. Чвор у стаблу везан је са оцем и неколико *синова*. За чвор v који се налази на путу од чвора u до корена каже се да је *предак* чвора u; у том случају је чвор u *потомак* чвора v. Максимални број синова чвора у графу зове се *степен* стабла. Обично је за синове сваког чвора дефинисан редослед, тако да се синови могу идентификовати својим редним бројем. Стабло степена два зове се *бинарно стабло*. Чвор у бинарном стаблу може да има највише два сина, левог и десног. Чвор без деце зове се *лист*, а чвор који није лист зове се *унутрашњи чвор*. *Висина стабла* је највећи ниво хијерархије у њему, тј. максимално растојање од корена до неког чвора. Чвор има *кључ* из неког потпуно уређеног скупа (нпр. цели или реални број). Сваки чвор може да има поље за *податак* — што зависи од примене.

У наставку ће бити речи о бинарном стаблу претраге и хипу, као примерима структура података заснованим на бинарном стаблу. Пре тога размотрићемо могуће начине представљања стабла. У зависности од тога да ли се користе показивачи или не, представљање стабла је експлицитно, односно имплицитно.



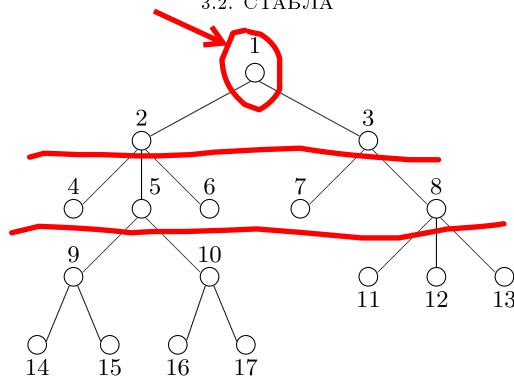


Рис. 4. Пример коренског стабла.

При *експлицитном* представљању се чвор са k синова представља слогом чији је део вектор са k показивача ка синовима; понекад је део слога и показивач ка оцу. Погодно је да сви чворови буду истог типа — са m показивача, где је m највећи број синова неког чвора, тј. степен стабла. Стабло степена већег од два може се представити коришћењем само два показивача по чвору: једног ка првом сину, а другог ка наредном *брату* — видети пример на слици 5.

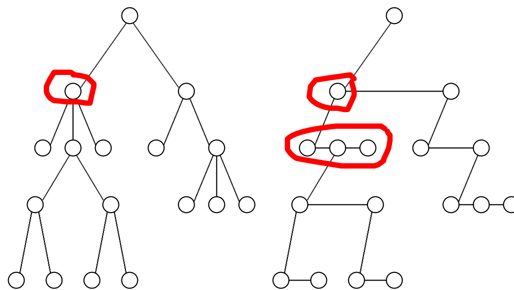


Рис. 5. Приказивање небинарног стабла са по највише два показивача по чвору.

За *имплицитно* представљање стабла не користе се показивачи: сви чворови се смештају у вектор, а везе између чворова одређене су њиховом позицијом у вектору. Ако је са A означен вектор у који се смештају чворови бинарног стабла, онда се корен смешта у $A[1]$; његов леви, односно десни син записују се у $A[2]$, односно $A[3]$, итд. (видети слику 6). Дакле, у вектору се чворови записују оним редом којим се пролазе с лева у десно, ниво по ниво; при томе се очигледно у вектору мора резервисати простор и за одсутне чворове. Индукцијом се може показати да ако је чвор v записан у елементу $A[i]$, онда су његов леви, односно десни син записани у елементу $A[2i]$, односно $A[2i+1]$.

Овакав начин представљања стабла је погодан због своје компактности. Ипак, ако је стабло неуравнотежено (односно неки листови су много даље

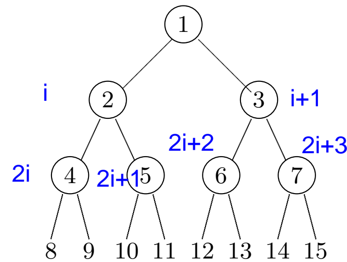


Рис. 6. Имплицитно представљање бинарног стабла. Број уз чвор означава његову позицију у вектору.

од корена него други), мора се резервисати и простор за много непостојећих чворова, па се тада оваквим представљањем стабла нерационално користи меморијски простор. Тако, на пример, за смештање стабла са осам чворова на слици 7 користи се вектор дужине чак 30.

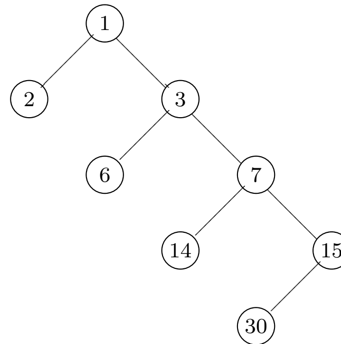


Рис. 7. Пример неуравнотеженог стабла, за које је имплицитно представљање неефикасно.

3.3. Хип

Хип је бинарно стабло које задовољава *услов хипа*: кључ сваког чвора је већи или једнак од кључева његових синова. Непосредна последица дефиниције (због транзитивности релације $\leq$) је да је у хипу кључ сваког чвора већи или једнак од кључева свих његових потомака.

Хип је погодан за реализацију *листе са приоритетом*, апстрактне структуре података за коју су дефинисане две операције:

- **umetni**(x) – уметни кључ x у структуру, и
- **ukloni**() – уклони (обриши) највећи кључ из структуре.

Хип се може реализовати са имплицитно или експлицитно задатим стаблом. Овде ће бити приказана реализација хипа на имплицитно задатом стаблу, јер

се две основне операције могу изводити на начин који обезбеђује да стабло увек буде уравнотежено. Дакле, сматраћемо да су кључеви чворова хипа смештени у вектору A, при чему је k горња граница за број елемената хипа (ако се та граница не зна унапред, онда се за хип мора користити повезана листа). Ако је n текући број елемената хипа, онда се за смештање елемената хипа користе локације у вектору A са индексима од 1 до n.

Уклањање са хипа елемента са највећим кључем. Кључ највећег елемента је $A[1]$, па га је лако пронаћи. После тога треба трансформисати преостали садржај вектора тако да представља исправан хип. То се може постићи тако да се најпре $A[n]$ прекопира у корен ($A[1] := A[n]$) и n смањи за један. Означимо са $x = A[1]$ нови кључ корена, а са $y = \max\{A[2], A[3]\}$ већи од кључева синова корена. Подстабла са коренима у $A[2]$ и $A[3]$ су исправни хипови; ако је $x \geq y$ (односно $x \geq A[2]$ и $x \geq A[3]$), онда је комплетно стабло исправан хип. У противном (ако је $x < y$), после замене места кључева x и y у стаблу (односно у вектору A), подстабло са непромењеним кореном остаје исправан хип, а другом подстаблу је промењен (смањен!) кључ у корену. На то друго подстабло примењује се рекурзивно (индукција) иста процедура која је примењена на полазно стабло. Индуктивна хипотеза је да ако је после i корака x на позицији $A[j]$, онда само подстабло са кореном $A[j]$ евентуално не задовољава услов хипа. Даље се x упоређује са $A[2j]$ и $A[2j+1]$ (ако постоје) и, ако није већи или једнак од оба ова кључа, замењује место са већим од њих. Дакле, кључ x спушта се наниже дотле док не доспе у лист или у корен неког подстабла у коме је већи или једнак од кључева оба сина. На слици 8 приказан је пример уклањања највећег елемента из хипа.

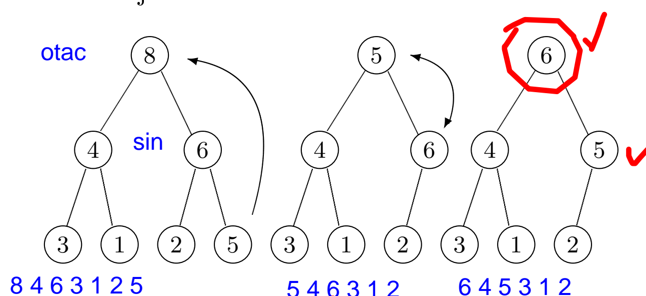


Рис. 8. Уклањање највећег елемента са хипа.

Уметање новог елемента у хип. Операција се изводи на сличан начин као и претходна. Најпре се n повећа за један и на слободну позицију $A[n]$ упише се нови кључ. Тај кључ упоређује се са кључем оца $A[i]$, где је $i = \lceil n/2 \rceil$, па ако је већи од њега, замењују им се места. Ова замена може само да повећа кључ $A[i]$, па ће чвор са кључем $A[i]$ бити корен исправног хипа. Може показати да је тачна следећа индуктивна хипотеза: ако је уметнути кључ после низа премештања доспео у елеменат $A[j]$, онда стабло са кореном $A[j]$ јесте исправан хип, а ако се то подстабло уклони, остатак стабла задовољава услов хипа.

Процес се наставља премештањем новог кључа навише, све док не буде мањи или једнак од кључа оца (или док не дође до корена) — тада комплетно стабло задовољава услов хипа; видети пример на слици 9. Максимални број поређења је ограничен висином стабла $\lfloor \log_2 n \rfloor$.

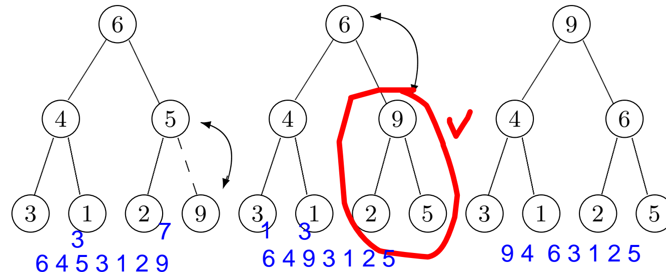


Рис. 9. Уметање елемента у хип.

Број упоређивања приликом извршења обе наведене операције са хипом је дакле ограничен висином стабла, односно са $O(\log n)$, што значи да се оне ефикасно извршавају. Недостатак хипа је у томе што се не може ефикасно извести тражење задатог кључа. На сликама 10 и 11 приказани су на псеудо–паскалу алгоритми за уклањање највећег кључа, односно уметање новог кључа у хип.

Примедба. У оваквим не сасвим формалним спецификацијама алгоритама на псеудо-паскалу често ће бити коришћен опис речима; за блокове су испуштени **begin** и **end** — сем на почетку и крају алгоритма. Декларације података се не наводе експлицитно, јер то најчешће није неопходно.

3.4. Бинарно стабло претраге

Сваки чвор бинарног стабла има највише два сина, па се може фиксирати неко пресликавање скупа његових синова у скуп {леви, десни}. Ово пресликавање може се искористити при исцртавању стабла у равни: леви син црта се лево, а десни син десно од оца; сви чворови исте генерације (нивоа) цртају се на истој хоризонталној правој, испод праве на којој су чворови претходне генерације. Левом, односно десном сину корена стабла одговара лево, односно десно подстабло. У *бинарном стаблу претраге* (БСП) кључ сваког чвора већи је од кључева свих чворова левог подстабла, а мањи од кључева свих чворова десног подстабла. Претпостављамо због једноставности да су кључеви свих чворова различити. БСП омогућује ефикасно извршавање следеће три операције:

- **Nadji**(x) – нађи елеменат са кључем x у структури, или установи да га тамо нема (претпоставља се да се сваки кључ у структури налази највише једном);
- **Umetni**(x) – уметни кључ x у структуру, ако он већ није у њој; у противном ова операција нема ефекта, и

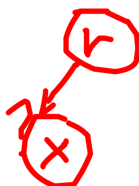
```
Алгоритам Skini_max_sa_hipa(A, n);
Улаз: A (вектор величине n за смештање хипа).
Излаз: Vrh_hipa (највећи елеменат хипа), A (нови хип), n (нова величина хипа; ако је
    n = 0 хип је празан).
begin
    if n = 0 then print "хип је празан"
    else
        Vrh_hipa := A[1];            <- umesto :=
        A[1] := A[n];
        n := n − 1;
        Otac := 1;
        if n > 1 then
            Sin := 2;
            while Sin ≤ n do
                if Sin + 1 ≤ n and A[Sin] < A[Sin + 1] then
                    Sin := Sin + 1;
                if A[Sin] > A[Otac] then
                    zameni (A[Otac], A[Sin]);
                    Otac := Sin;
                    Sin := 2 · Sin;
                else
    return Vrh_hipa Sin := n + 1 {да би се искочило из петље}
end
```

Рис. 10. Алгоритам за уклањање највећег елемента из хипа.

- **Ukloni**(x) – ако у структури података постоји елеменат са кључем x, уклони га.

Структура података са овим операцијама зове се *речник*. За њену реализацију може се искористити БСП. Због важности динамичких убацивања и брисања елемената из речника, и да се не би морала унапред задавати горња граница за број елемената, сматраћемо да је одговарајуће стабло представљено експлицитно. Сваки чвор стабла је слог са бар три поља: кључ, леви и десни. Друго, односно треће поље су показивачи ка другим чворовима, или **nil** (ако чвор нема левог, односно десног сина).

Налажење кључа. Ово је операција по којој је структура података бинарно стабло претраге добила име. Потребно је пронаћи у стаблу елеменат са задатим кључем x. Број x упоређује се са кључем r корена БСП. Ако је $r = x$ онда је тражење завршено. Ако је пак $x < r$ (односно $x > r$) онда се тражење рекурзивно наставља у левом (односно десном) подстаблу. На слици 12 приказан је псеудо код за овај алгоритам. Алгоритам је по идеји сличан са методом половљења интервала за нумеричко решавање једначине $f(x) = c$,



Алгоритам Upis_u_hip(A, n, x);
Улаз: A (вектор величине n за смештање хипа), x (број).
Излаз: A (нови хип), n (нова величина хипа).
begin
    $n := n + 1$; {претпоставка је да ново n није веће од величине A}
    $A[n] := x$;
    $Sin := n$;
    $Otac := n \operatorname{div} 2$;
    while $Otac \geq 1$ **do**
        if $A[Otac] < A[Sin]$ **then**
            zameni $(A[Otac], A[Sin])$;
            $Sin := Otac$;
            $Otac := Otac \operatorname{div} 2$;
        else
            $Otac := 0$ {за искакање из петље}
end

Рис. 11. Алгоритам за упис новог елемента у хип.

кад је $f(x)$ монотона непрекидна функција на интервалу $[a, b]$, при чему је $f(a) < 0$ и $f(b) > 0$. Вредност $f(d)$ у средњој тачки $d = (a + b)/2$ упоређује се са c, после чега се тражење наставља у левој или десној половини интервала $[a, b]$, у зависности од тога да ли је $f(d) < c$ или $f(d) > c$. Ако је $f(d) = c$ онда је решење једначине нађено.

Алгоритам Nadji_u_BSP$(Koren, x)$;
Улаз: $Koren$ (показивач на корен БСП), x (број).
Излаз: $Čvor$ (показивач на чвор који садржи кључ x, или **nil** ако таквог чвора нема).
begin
    if $Koren =$ **nil** **or** $Koren\hat{}.Ključ = x$ **then** $Čvor := Koren$
    {$Koren\hat{}$ је слог чију адресу садржи показивач $Koren$}
    else
        if $x < Koren\hat{}.Ključ$ **then** $Nadji_u_BSP(Koren\hat{}.Levi, x)$
        else $Nadji_u_BSP(Koren\hat{}.Desni, x)$
end

Рис. 12. Налажење броја у бинарном стаблу претраге.

Уметање је такође једноставна операција. Кључ x који треба уметнути најпре се потражи у БСП; ако се пронађе, уметање је завршено; претпоставља је да се у речнику не чувају поновљене копије истих елемената. Ако се x не

пронађе у БСП, онда се приликом тражења дошло до листа, или до чвора без једног сина, управо са оне стране где треба уметнути нови чвор. Тада се x умеће као леви, односно десни син тог чвора — ако је мањи, односно већи од кључа тог чвора (слика 13).

```
Алгоритам Umetni_u_BSP(Koren, x);
Улаз: Koren (показивач на корен БСП), x (број).
Излаз: У БСП се умеће чвор са кључем x на кога показује показивач Sin;
    ако већ постоји чвор са кључем x, онда Sin = nil.
begin
    if Koren = nil then
        креирај нови чвор на кога показује Sin;
        Koren := Sin;
        Koren^.Ključ = x; Koren^.levi := nil; Koren^.Desni := nil;
    else
        Čvor := Koren; {текући чвор у стаблу}
        Sin := Koren; {поставља се на вредност различиту од nil}
        while Čvor ≠ nil and Sin ≠ nil do
            if Čvor^.Ključ = x then Sin := nil
            else {силазак низ стабло за један ниво}
                Otac := Čvor;
                if x < Čvor^.Ključ then Čvor := Čvor^.Levi
                else Čvor := Čvor^.Desni;
        if Sin ≠ nil then {нови чвор је син чвора }Otac
            креирај нови чвор на кога показује Sin;
            Sin^.Ključ := x;
            Sin^.Levi = nil; Sin^.Desni = nil;
            if x < Otac^.Ključ then Otac^.Levi := Sin
                else Otac^.Desni := Sin
end
```

Рис. 13. Уметање броја у бинарно стабло претраге.

Брисање чвора из БСП коме је кључ једнак задатом броју је нешто компликованија операција од претходних (видети алгоритам на слици 15). Најједноставније је обрисати лист, слика 14(а). На сличан начин лако је обрисати унутрашњи чвор са само једним сином: тај чвор се уклања, а подстабло са кореном у његовом сину се подиже за један ниво тако да му корен буде на месту обрисаног чвора, слика 14(б). Ако треба уклонити унутрашњи чвор B са оба сина, може се најпре у левом подстаблу чвора B пронаћи "најдеснији" чвор x, односно чвор са највећим кључем (слика 14(ц); полазећи од корена тог стабла прелази се докле год је то могуће из чвора у његовог десног сина; на крају тог

пута долази се до чвора x). Кључ чвора x копира се у кључ B, а чвор x уклања се на већ описани начин јер је он лист или чвор са само једним сином. После ове операције БСП остаје конзистентно: кључ x већи је од кључева свих чворова у левом подстаблу испод своје нове локације. Чвор x зове се *претходник* чвора B у стаблу. Исти резултат може се постићи помоћу "најлевијег" чвора у десном подстаблу чвора B, његовог *следбеника*.

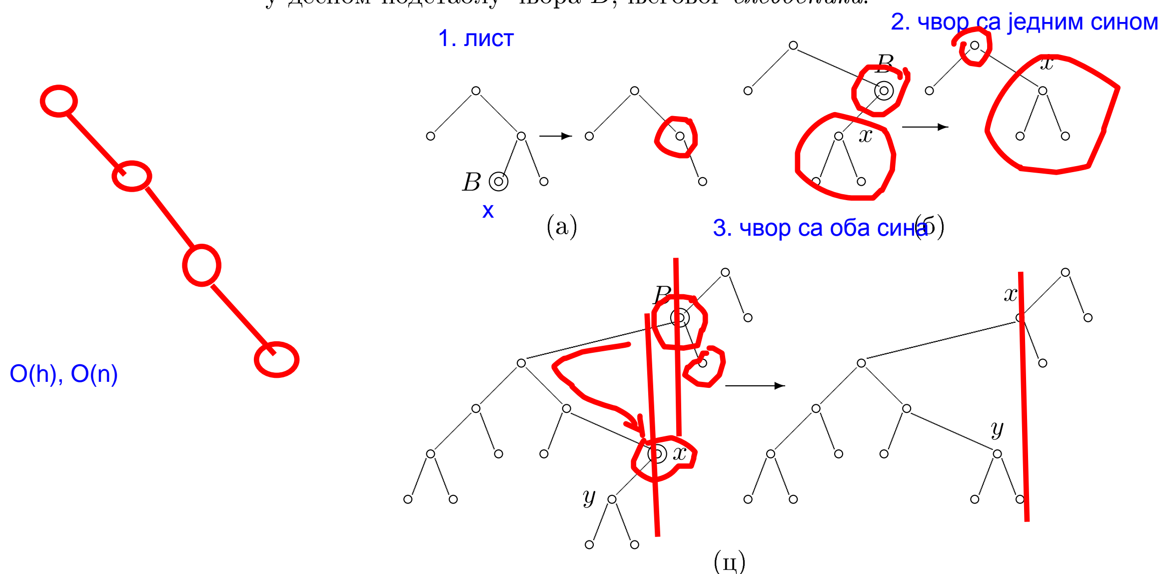


Рис. 14. Брисање елемента бинарног стабла претраге.

Сложеност. За све три разматране операције време извршења зависи од облика стабла и положаја чвора на коме се врши интервенција — у најгорем случају се трагање завршава у листу стабла. Све остало захтева само константан број елементарних операција (на пример, стварно уписивање кључа или замена кључева при брисању). Дакле, у најгорем случају је сложеност пропорционална висини стабла. Ако је стабло са n чворова у разумној мери уравнотежено (овај појам ће у наредном одељку бити прецизније дефинисан), онда је његова висина пропорционална са $\log_2 n$, па се све три операције ефикасно извршавају. Проблем настаје ако стабло није уравнотежено — на пример, стабло које се добија уметањем растућег низа кључева има висину $O(n)$! Ако се стабло добија уметањем n кључева у случајно изабраном поретку, онда је очекивана висина стабла $2 \ln n$, па су операције тражења и уметања ефикасне. Стабла са дугачким путевима могу да настану као резултат уметања у уређеном или скоро уређеном редоследу. Брисања могу да изазову проблеме чак и ако се изводе у случајном редоследу: разлог томе је асиметрија до које долази ако се увек користи претходник за замену обрисаног чвора. После честих брисања и уметања, стабло може да достигне висину $O(\sqrt{n})$, чак и у код

```
Алгоритам Ukloni_iz_BSP(Koren, x);
Улаз: Koren (показивач на корен БСП), x (број).
Излаз: Из БСП се уклања чвор са кључем x, ако такав постоји.
    {Претпоставка је да се корен никад не брише, и да су сви кључеви различити}
begin
    {прва фаза: тражење чвора са кључем x}
    Čvor := Koren;
    while Čvor ≠ nil and Čvor^.Ključ ≠ x do
        Otac := Čvor;
        if x < Čvor^.Ključ then Čvor := Čvor^.Levi
        else Čvor := Čvor^.Desni;
    if Čvor = nil then print("x није у БСП"); halt;
    {брисање нађеног чвора Čvor, или његовог претходника}
    if Čvor ≠ Koren then
        if Čvor^.Levi = nil then
            if x ≤ Otac^.Ključ then
                Otac^.Levi := Čvor^.Desni
            else Otac^.Desni := Čvor^.Desni
        else if Čvor^.Desni = nil then
            if x ≤ Otac^.Ključ then
                Otac^.Levi := Čvor^.Levi
            else Otac^.Desni := Čvor^.Levi
        else {случај са два сина}
            Čvor1 := Čvor^.Levi;
            Otac1 := Čvor;
            while Čvor1^.Desni ≠ nil do
                Otac1 := Čvor1;
                Čvor1 := Čvor1^.Desni;
            {а сада брисање; Čvor1 је претходник чвора Čvor}
            Čvor^.Ključ := Čvor1^.Ključ
            if Otac1 = Čvor then {}
                Otac1^.Levi := Čvor1^.Levi; {Čvor1 је леви син чвора Otac1}
            else
                Otac1^.Desni := Čvor1^.Levi; {Čvor1 је десни син чвора Otac1}
end
```

Рис. 15. Брисање задатог броја из бинарног стабла претраге.

случајних уметања и брисања. Асиметрија се може смањити ако се за замену обрисаног чвора наизменично користе претходник и следбеник.

На срећу, постоји начин да се избегну дугачки путеви у БСП. Један такав метод биће размотрен у следећем одељку.

3.5. АВЛ стабла

АВЛ стабла (која су име добила по ауторима, Адельсон–Вельский, Ландис, 1962.) су прва структура која гарантује да сложеност ни једне од операција тражења, уметања и брисања у *најгорем случају* није већа од $O(\log n)$, где је n је број елемената. Идеја је уложити допунски напор да се после сваке операције стабло *уравнотежи*, тако да висина стабла увек буде $O(\log n)$. При томе се уравнотеженост стабла дефинише тако да се може лако одржавати.

Дефиниција 3.1. АВЛ стабло је бинарно стабло претраге код кога је за сваки чвор апсолутна вредност разлике висина левог и десног подстабла мања или једнака од један.

Као што показује следећа теорема, висина АВЛ стабла је $O(\log n)$.

Теорема 3.1. *За АВЛ стабло са n унутрашњих чворова висина h задовољава услов $h < 1.4405 \log_2(n+2) - 0.327$.*

Доказательство. АВЛ стабло T_h висине $h \geq 2$ са најмањим могућим бројем чворова може се добити следећом индуктивном конструкцијом: његово подстабло мање висине $h-2$ такође треба да буде АВЛ стабло са минималним бројем чворова, дакле T_{h-2}; слично, његово друго подстабло треба да буде T_{h-1}. Стабла описана оваквом "диференцном једначином" зову се Фибоначијева стабла. Неколико првих Фибоначијевих стабала, таквих да је у сваком унутрашњем чвору висина левог подстабла мања, приказано је на слици 16. Означимо са n_h минимални број унутрашњих чворова АВЛ стабла висине h, тј. број унутрашњих чворова стабла T_h; $n_0 = 0$, $n_1 = 1$, $n_2 = 2$, $n_3 = 4$. По дефиницији Фибоначијевих стабала је $n_h = n_{h-1} + n_{h-2} + 1$, односно $n_h + 1 = (n_{h-1} + 1) + (n_{h-2} + 1)$, за $h \geq 2$. Видимо да бројеви $n_h + 1$ задовољавају исту диференцну једначину (2.5) као и Фибоначијеви бројеви F_h ($F_{h+2} = F_{h+1} + F_h$ за $h \geq 2$; $F_1 = F_2 = 1$). Узимајући у обзир и прве чланове низа, индукцијом је доказано да је $n_h + 1 = F_{h+2}$. Према томе, за број n унутрашњих чворова АВЛ стабла висине h важи неједнакост

$$n \geq F_{h+2} - 1.$$

Полазећи од израза (2.9) за општи члан Фибоначијевог низа, добијамо

$$n \geq \frac{1}{\sqrt{5}}\left(\alpha^{h+2} - \left(-\frac{1}{\alpha}\right)^{h+2}\right) - 1 > \frac{1}{\sqrt{5}}\alpha^{h+2} - 2,$$

где је са $\alpha = (\sqrt{5}+1)/2$ означен позитиван корен карактеристичне једначине линеарне диференцне једначине (2.5). Ова неједнакост еквивалентна је са

$$h < \frac{1}{\log_2 \alpha} \log_2(n+2) - \left(2 - \frac{\log_2 5}{2\log_2 \alpha}\right),$$

чиме је теорема доказана, јер је $1/\log_2 \alpha \simeq 1.44042$ и $\log_2 5/(2\log_2 \alpha) \simeq 0.3277$. □

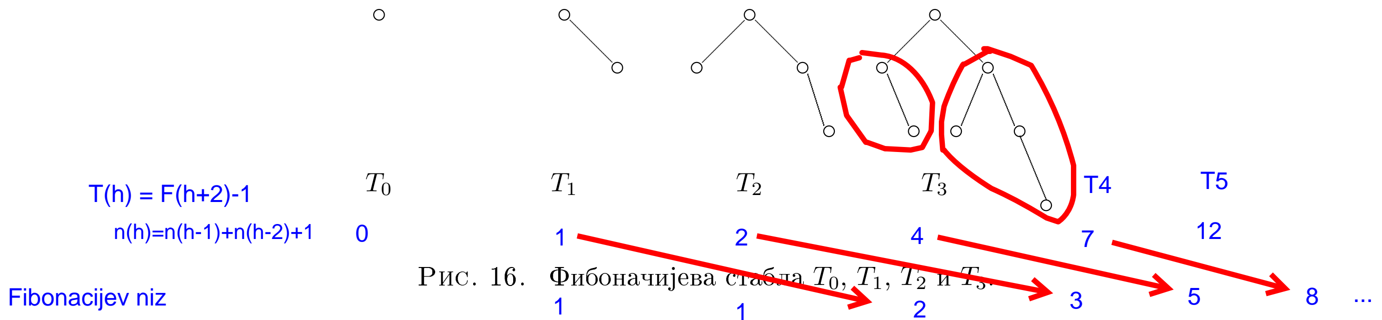


Рис. 16. Фибоначијева стабла T_0, T_1, T_2 и T_3.

Остаје још да видимо како се после уметања, односно брисања елемента из АВЛ стабла, може интервенисати тако да резултат и даље буде АВЛ стабло.

Приликом уметања новог броја у АВЛ стабло поступа се најпре на начин уобичајен за бинарно стабло претраге: проналази се место чвору, па се у стабло додаје нови лист са кључем једнаким задатом броју. Без умањења општости може се претпоставити да је нови чвор уметнут у лево подстабло. Ако је висина левог подстабла мања или једнака од висине десног подстабла, или се његова висина не мења после уметања, онда стабло и после уметања остаје уравнотежено — висина левог подстабла се уметањем новог листа не може повећати за више од један. У противном се, ако је висина левог подстабла већа, и после уметања се повећа за један, добија неуравнотежено стабло. Према томе да ли је нови лист додат левом или десном подстаблу левог подстабла, разликујемо два случаја, видети слику 17. У првом случају се на стабло примењује *ротација*: корен левог подстабла B (видети слику 18) подиже се и постаје корен стабла, а остатак стабла преуређује се тако да стабло и даље остане стабло претраге. Стабло T_1 се "подиже" за један ниво остајући и даље лево подстабло чвора B; стабло T_2 остаје на истом нивоу али уместо десног подстабла B постаје лево подстабло A; десно подстабло A спушта се за један ниво. Испрекидана цртица на слици 18 показује да ротација решава проблем и ако су висине стабала T_1, T_2 једнаке, и ако је висина T_1 за један већа од висине T_2. Други случај је компликованији; тада се стабло се може уравнотежити *двоструком ротацијом*, видети слику 19.

Запажамо да у оба случаја висина стабла после уравнотежавања остаје непромењена. Чвор A на овим сликама, тзв. *критичан чвор*, је дакле корен најмањег подстабла које садржи уметнути чвор, а које после његовог уметања постаје не–АВЛ. Уравнотежавање подстабла коме је корен критичан чвор

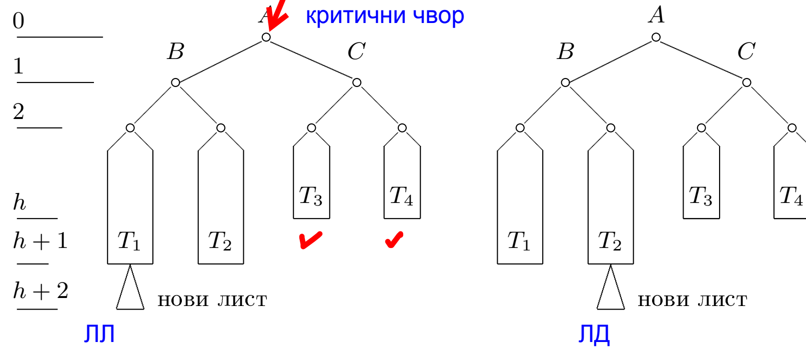


Рис. 17. Уметања која ремете АВЛ својство стабла.

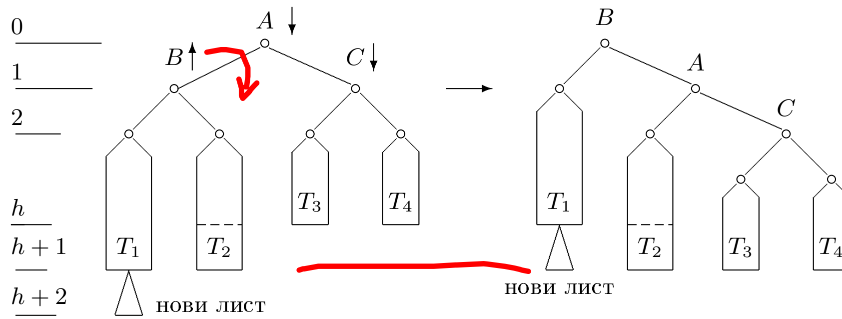


Рис. 18. Уравнотежавање АВЛ–стабла после уметања — ротација.

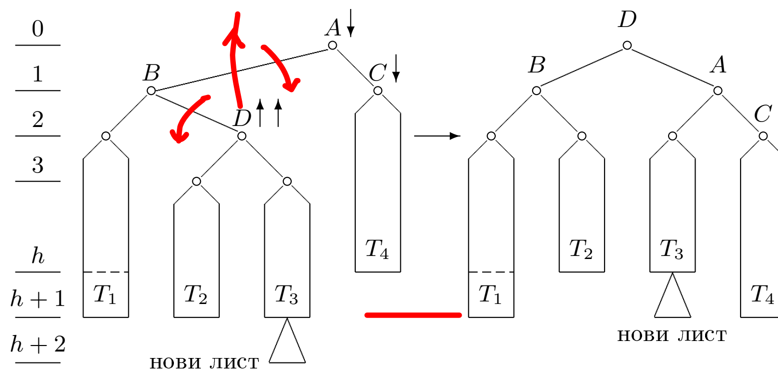


Рис. 19. Уравнотежавање АВЛ–стабла после уметања — двострука ротација.

због тога не утиче на остатак стабла. Према томе, после уметања новог чвора мора се пронаћи критичан чвор и установити који је од горе размотрених

случајева у питању. Уз сваки чвор стабла чува се његов *фактор равнотеже*, једнак разлици висина његовог левог и десног подстабла; за АВЛ стабло су те разлике елементи скупа $\{-1, 0, 1\}$. Уравнотежавање постаје неопходно ако је фактор неког чвора ± 1, а нови чвор се умеће на "погрешну" страну. Према томе, критичан чвор мора да има фактор равнотеже различит од нуле. После уравнотежавања висине изнад критичног чвора остају непромењене. Комплетан поступак уметања са уравнотежавањем дакле изгледа овако: идући наниже приликом уметања памти се последњи чвор са фактором различитим од нуле; кад се дође до места уметања, установљава се да ли је уметање на "доброј" или "погрешној" страни у односу на критичан чвор, а онда се још једном пролази пут уназад до критичног чвора, поправљају се фактори равнотеже и извршавају евентуалне ротације.

Брисање је компликованије, као и код обичног БСП. У општем случају се уравнотежавање не може извести помоћу само једне или две ротације. На пример, да би се Фибоначијево стабло F_h са n чворова уравнотежило после брисања "добро" изабраног чвора, потребно је извршити $h-2$, односно $O(\log n)$ ротација (видети слику 20 за $h = 4$). У општем случају је граница за потребан број ротација $O(\log n)$. На срећу, свака ротација захтева константни број корака, па је сложеност уравнотежавања АВЛ стабла после брисања ограничена са $O(\log n)$. На овом месту прескочићемо детаље.

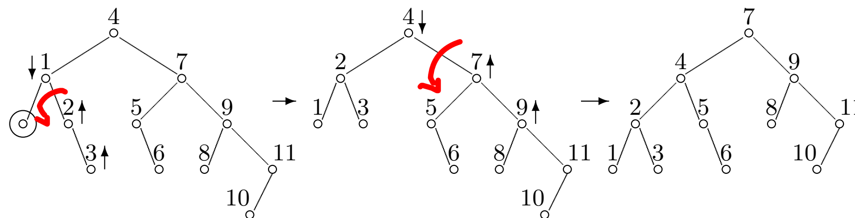


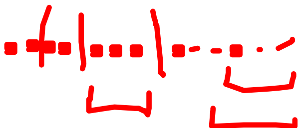
Рис. 20. Уравнотежавање Фибоначијевог стабла T_4 после брисања чвора, помоћу две ротације.

АВЛ стабло је ефикасна структура података. Чак и у најгорем случају АВЛ стабла захтевају највише за 45% више упоређивања од оптималних стабала. Емпиријска испитивања су показала да се просечно тражење састоји од око $\log_2 n + 0.25$ поређења. Основни недостатак АВЛ стабала је потреба за додатним меморијским простором за смештање фактора равнотеже, као и чињеница да су програми за рад са АВЛ стаблима доста компликовани. Постоје и друге варијанте уравнотежених стабала претраге ($2-3$ стабла, B-стабла, црвено-црна стабла и слично).

3.6. Хеш табеле

Хеш табеле спадају међу најкорисније структуре података. Користе се за уметање и тражење, а у неким варијантама и за брисање. Основна идеја је једноставна. Ако треба сместити податке са кључевима из опсега од 1 до n, а

на располагању је вектор дужине n, онда се податак са кључем i смешта на позицију i, $i = 1, 2, \ldots, n$. Ако су кључеви података из опсега од 1 до $2n$, још увек је згодно податке сместити на исти начин у вектор дужине $2n$ — тиме се постиже највећа ефикасност, која надокнађује утрошак меморијског простора. Ситуација се мења ако је опсег вредности кључева од 1 до M велики (ако је, на пример, M највећи природни број који се може сместити на конкретном рачунару): тада отпада варијанта са коришћењем вектора дужине M. Претпоставимо да треба сместити податке о 250 студената, при чему се сваки од њих идентификује својим матичним бројем са 13 декадних цифара. Уместо комплетног матичног броја као индекс у вектору могу се користити само његове три последње цифре: тада је за смештање података довољан вектор дужине 1000. Метод ипак није потпуно поуздан: могуће је да нека два студента имају исте три последње цифре матичног броја; начине решавања овог проблема размотрићемо касније. Могу се искористити и четири последње цифре матичног броја, или три последње цифре комбиноване са првим словом имена студента, да би се још више смањила могућност оваквог догађаја. С друге стране, коришћење више цифара захтева табелу веће димензије, са релативно мањим искоришћеним делом.



Претпоставимо да је дато n кључева из скупа U величине $| U |= M \gg n$. Идеја је да кључеве сместимо у табелу величине m, тако да m није много веће од n. Потребно је дакле дефинисати функцију $h : \{0, 1, \ldots, M - 1\} \to \{0, 1, \ldots, m - 1\}$, *хеш функцију* која за сваки податак одређује његову позицију на основу вредности одговарајућег кључа. У горњем примеру, као вредност хеш функције од кључа једнаког матичном броју, узете су његове три последње цифре. Ако се вредности ове функције лако израчунавају, биће лако приступити податку. Међутим, ако је величина табеле m недовољно велика, дешаваће се да различитим кључевима одговарају исте локације. Овакав непожељан догађај зове се *колизија*. Потребно је дакле решити два проблема: налажење хеш функција које минимизирају вероватноћу појаве колизија, и поступак за обраду колизија кад до њих ипак дође. Иако је величина M скупа U много већа од величине табеле m, стварни скуп кључева које треба обрадити обично није велики. Добра хеш функција пресликава кључеве равномерно по табели — да би се минимизирала могућност колизија проузрокованих нагомилавањем кључева у неким областима табеле. У просеку се у сваку локацију табеле овако изабраном хеш функцијом пресликава се велики број (из скупа свих могућих) кључева, њих M/m. Хеш функција треба да трансформише скуп кључева равномерно у скуп *случајних* локација из скупа $\{0, 1, \ldots, m - 1\}$. Униформност и случајност су битне особине хеш функције. Тако, на пример, погрешно је за хеш функцију од 13-цифреног матичног броја узети број који чине пета, шеста и седма цифра матичног броја: те три цифре су три најниже цифре године рођења, па за било коју величину групе студената узимају мање од десетак различитих вредности од 1000 могућих.

Хеш функције. Ако је величина табеле m прост број, а кључеви су цели бројеви, онда је једноставна и добра хеш функција дата изразом

$$h(x) = x \bmod m.$$

Ако пак m није прост број (на пример $m = 2^k$), онда се може користити функција

$$h(x) = (x \bmod p) \bmod m,$$

где је p прост број такав да је $m \ll p \ll M$. Неугодна је ситуација кад су сви кључеви облика $r + kp$, за неки цели број r, јер ће сви кључеви имати исту вредност хеш функције r. Тада има смисла увести још један ниво рандомизације ("разбацивања"), тиме што би се сама хеш функција бирала на случајан начин. На пример, број p могао би се бирати са унапред припремљене листе простих бројева. Друга могућност је да се користе хеш функције облика $h(x) = (ax + b \bmod p) \bmod m$, где су $a \neq 0$ и b случајно изабрани бројеви мањи од p. Вредности ове функције се израчунавају на сложенији начин, али је њена предност да је *у просеку* добра за све улазе. Разуме се да се за све приступе истој табели мора користити иста хеш функција. У много случајева потребно је више независних хеш табела, или се табеле често креирају и бришу. Тада се за сваку нову табелу може користити друга хеш функција.

Обрада колизија. Најједноставији начин за обраду колизија је тзв. *одвојено низање* сваки елеменат хеш табеле је показивач на повезану листу, која садржи све кључева смештене на ту локацију у табели. При тражењу задатог кључа најпре се израчунава вредност хеш функције, а затим се адресирана повезана листа линеарно претражује. На први поглед уписивање се поједностављује уметањем кључа на почетак листе. Међутим, листа се ипак мора прегледати до краја да би се избегло уписивање дупликата (елемената са кључем једнаким неком већ уписаном). Овакав поступак је неефикасан ако су повезане листе дугачке, до чега долази ако је табела мала у односу на број кључева, или је изабрана лоша хеш функција. Поред тога, потешкоће ствара и динамичка алокација меморије, а потребно је резервисати и простор за показиваче. С друге стране, одвојено низање ради без грешака и ако је изабрана погрешна величина табеле, за разлику од осталих (приближних) метода.

Друго једноставно решење је *линеарно попуњавање* ако је локација са адресом $h(x)$ већ заузета, онда се нови елеменат са кључем x записује на суседну локацију $h(x)+1 \bmod m$ (локација која следи иза $m-1$ је 0). Приликом тражења задатог кључа x неопходно је линеарно претраживање почевши од локације $h(x)$ до прве непопуњене локације. Ово решење је ефикасно ако је табела релативно ретко попуњена. У противном долазиће до много *секундарних колизија*, односно колизија проузрокованих кључевима са различитим вредностима хеш функције. На пример, нека је локација i заузета, а локација $i+1$ слободна. Нови кључ који се пресликава у i проузрокује колизију, и смешта се на локацију $i+1$. До овог тренутка проблеми су решени ефикасно, уз минимални напор. Међутим, ако се нови кључ преслика у $i+1$, имамо случај секундарне колизије, а локација $i + 2 \bmod m$ постаје заузета (ако већ није била). Сваки нови кључ

пресликан у i, $i+1$ или $i+2 \bmod m$, не само да изазива нову секундарну колизију, него и повећава величину овог попуњеног одсечка, што касније изазива још више секундарних колизија. Ово је тзв. ефекат груписања. Кад је табела скоро пуна, број секундарних колизија при линеарном попуњавању је врло велики, па се претрага деградира у линеарну.

При линеарном попуњавању се брисања не могу извести коректно. Ако при уписивању кључа y приликом тражења празног места прескочи нека локација $i = h(x)$, па затим кључ x буде избрисан, онда се после тога кључ y више не може пронаћи у табели: тражење се завршава на празној локацији i. Дакле, ако су потребна и брисања, мора се користити неки од метода за обраду колизија који користи показиваче.

Ефекат груписање може се ублажити *двоструким хеширањем.* Кад дође до колизије при упису x, $h(x) = i$, израчунава се вредност друге хеш функције $h_2(x)$, па се x смешта на прву слободну међу локацијама $i + h_2(x) \bmod m$, $i + 2h_2(x) \bmod m \ldots$, уместо $i + 1 \bmod m$, $i + 2 \bmod m \ldots$. Ако се други кључ y пресликава у нпр. $i + h_2(x) \bmod m$, онда се покушава се локацијом $i + h_2(x) + h_2(y) \bmod m$ уместо са $i + h_2(x) + h_2(x) \bmod m$. Ако је вредност $h_2(y)$ независна од $h_2(x)$ онда је груписање практично елиминисано. Приликом избора друге хеш функције мора се водити рачуна о томе да остаци $i + jh_2(x) \bmod m$ за $j = 0, 1, \ldots, m - 1$ покривају комплетну табелу, односно да вредности $h_2(x)$ буду узајамно просте са m.

3.7. Проблем формирања унија

Проблем формирања унијеа може да послужи као пример како добро изабрана структура података може да побољша ефикасност алгоритма. Нека је дато n елемената $x_1, x_2, \ldots, x_n$, који су подељени у дисјунктне подскупове. На почетку су подскупови једночлани. Над елементима и подскуповима могу се произвољним редоследом извршавати следеће две операције:

- **podskup**(i): даје име подскупа која садржи x_i;
- **unija**(A, B): формира унију подскупова A и B, подскуп са јединственим именом.

Циљ је наћи структуру података која омогућује што ефикасније извршавање обе ове операције.

Први кандидат за жељену структуру података је вектор. Пошто су сви елементи познати, могу се сместити у вектор X дужине n. Најједноставније решење постављеног проблема је сместити у $X[i]$ име подскупа која садржи елеменат x_i. Налажење подскупа коме припада x_i је тривијално. Међутим, формирање уније два подскупа је сложеније. Претпоставимо да је **unija**(A, B) нови подскуп са именом A. Тада свим елементима вектора X из подскупа B треба променити име у A.

Могућ је и други приступ овом проблему, заснован на индиректном адресирању, при чему је налажење уније једноставно. Сваки елеменат вектора X

је двочлани слог. Први елеменат слога је име подскупа коме елеменат припада, а други — показивач ка неком другом слогу. У сваком подскупу треба да постоји тачно један елеменат, чији слог садржи празан показивач **nil**; име тог елемента је по дефиницији име подскупа; за показивач тог елемента кажемо да је показивач подскупа. Ако је показивач елемента $X[i]$ усмерен ка елементу $X[j]$, онда је име подскупа који садржи $X[i]$ једнако имену подскупа који садржи $X[j]$. Претходни услов обезбеђује да ова дефиниција буде исправна: немогуће је да се подскуп састоји од елемената са индексима $i_1, i_2, \ldots, i_k$, и да поред тога показивач $X[i_j]$ буде усмерен ка $X[i_{j+1}]$, $j = 1, 2, \ldots, k-1$, а да је показивач $X[i_k]$ усмерен ка $X[i_1]$ — тада ни један елеменат подскупа не би имао показивач **nil**. На почетку су сви показивачи **nil**, тј. вектор X задаје n једночланих подскупова. Операција **unija**(A, B) изводи се тако што се показивач подскупа A усмери ка елементу, чији је показивач истовремено и показивач подскупа B (или обрнуто). После неколико формирања унија, вектор X задаје неколико стабала, видети слику 21. Свако стабло одговара подскупу, а сваки чвор елементу. Елеменат у корену стабла служи као име подскупа. Да би се пронашло име подскупа коме припада елеменат G, треба пратити ланац показивача до корена стабла, слога са показивачем **nil**. Ово подсећа на ситуацију кад неко промени адресу: уместо да свима јавља да је променио адресу, он на старој адреси оставља своју нову адресу (са молбом да му достављају приспелу пошту) — што је једноставније. Наравно, налажење праве адресе је сада нешто сложеније, односно операција налажења подскупа коме елеменат припада је мање ефикасна (нарочито ако су издужена стабла која представљају подскупове).

{1,2,6,7},{4,5},{3}

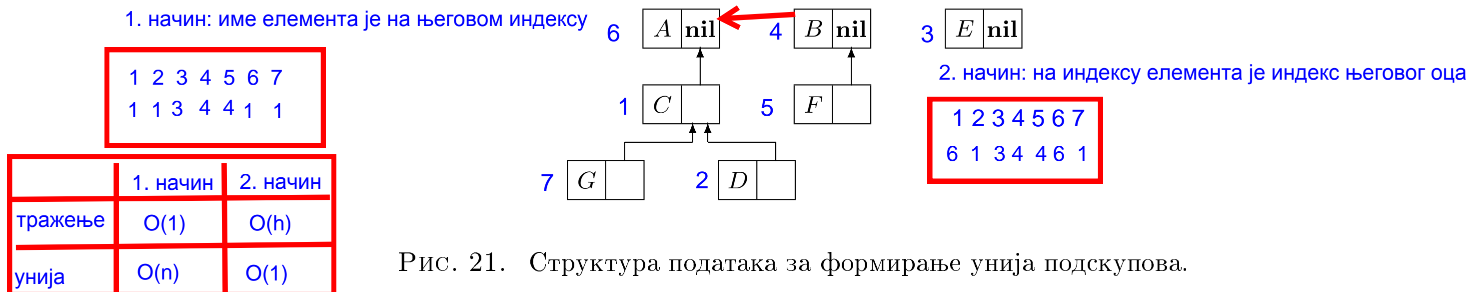
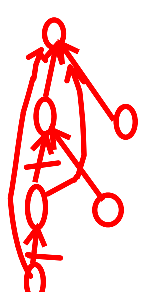


Рис. 21. Структура података за формирање унија подскупова.

Идеја на којој се заснива ефикасна структура за налажење унија је да се стабла уравнотеже, односно скрате. Унија подскупова A и B на слици 21 може се формирати усмеравањем показивача A ка B, или усмеравањем показивача B ка A. Очигледно је да се у другом случају добија уравнотеженије стабло. Закључак се лако уопштава: показивач корена стабла са мање елемената (односно произвољног од два стабла ако им је број елемената једнак) треба усмерити ка корену стабла са већим бројем елемената. Јасно је да при формирању сваке нове уније успут треба израчунавати и број елемената новог подскупа. Тај податак се смешта као део слога у корену стабла. Применом овог правила се при

формирању унија постиже да је висина стабла са m елемената увек мања или једнака од $\log_2 m$, што показује следећа теорема.

Теорема 3.2. *Ако се при формирању унија користи уравнотежавање, свако стабло висине h садржи бар 2^h елемената.*



Доказательство. Доказ се изводи индукцијом по броју формирања унија. Тврђење је очигледно тачно за прву унију: добија се стабло висине један са два елемента. Посматрајмо унију подскупова A и B. Нека су висине одговарајућих стабала $h(A)$, $h(B)$, и нека је број елемената у A већи или једнак од броја елемената у B. Висина комбинованог стабла једнака је већем од бројева $h(A)$ и $h(B)+1$. Ако је већи $h(A)$, онда ново стабло има висину као A, и више елемената него стабло A, па је тврђење теореме тачно. У противном, ново стабло има бар два пута више елемената од стабла B (јер A има бар толико елемената колико B), а висина му је за један већа од висине B, па је тврђење теореме такође тачно. □

Последица ове теореме је да се при тражењу подскупа коме припада елемент никад не пролази више од $\log_2 n$ показивача. Унија подскупова се формира за константно време. Према томе, за извршење низа од $m \geq n$ операција прве или друге врсте, број корака је највише $O(m \log n)$.

Ефикасност ове структуре података може се даље побољшати. Посматрајмо још једном аналогију са преусмеравањем пошиљки. У случају вишеструке промене адресе, пошиљка ће ићи од једне до друге адресе, док не доспе на коначно одредиште. Разумна идеја је да се по испоруци пошиљке уназад, на све успутне адресе, достави последња адреса. Дакле, кад се при одређивању подскупа неког елемента пређе пут до корена стабла, тај пут се јос једном пролази у обратном смеру, а показивачи свих успутних чворова преусмеравају се ка корену. Сложеност тражења тиме је повећана највише за константни фактор (тј. асимптотска сложеност тражења се не мења), а постиже се скраћивање путева при каснијим тражењима подскупа за све успутне чворове. Може се показати да се на овај начин сложеност низа од $m \geq n$ операција једне или друге врсте смањује на $O(m \log^* n)$, где се тзв. итерирани логаритам $\log^* n$ дефинише помоћу $\log^* 1 = \log^* 2 = 1$ и $\log^* n = 1 + \log^*(\lceil \log_2 n \rceil)$ за $n > 2$. Пошто за $n \leq 2^{65536}$ важи $\log^* n \leq 5$, видимо да је сложеност произвољног низа ових операција практично линеарна. Међутим, није познато да ли постоји алгоритам линеарне сложености за решавање овог проблема.

3.8. Графови

Алгоритми за рад са графовима предмет су поглавља 6. На овом месту биће приказане само структуре података погодне за представљање графова. Граф $G = (V, E)$ састоји се од скупа V *чворова* и скупа E *грана*. Грана одговара пару чворова. Другим речима, гране представљају релацију између чворова. На пример, граф може да представља скуп људи, а да грана повезује два човека ако се они познају. Граф је *усмерен* односно *неусмерен* ако су му гране уређени,

односно неуређени парови. Ако се усмерени граф представља цртежом, гранама се додају стрелице које воде ка другом чвору из уређеног пара. Једноставан пример графа је стабло. Ако на стаблу треба дефинисати хијерархију, онда се све гране могу оријентисати "од корена" (при чему је корен посебно издвојен чвор стабла). Таква стабла су *коренска стабла*. Могу се такође разматрати неусмерена стабла за која се не везује било каква хијерархија.

Уобичајена су два начина представљања графова. Први је *матрица повезаности* графа. Нека је $|V| = n$ и $V = \{v_1, v_2, \ldots, v_n\}$. Матрица повезаности графа G квадратна матрица $A = (a_{ij})$ реда n, са елементима $a_{ij} = 1$ акко $(v_i, v_j) \in E$; остали елементи матрице A су нуле. Ако је граф неусмерен, матрица A је симетрична. Врста i ове матрице је дакле вектор дужине n чија је j-та координата једнака 1 ако из чвора v_i води грана у чвор v_j, односно 0 у противном. Недостатак матрице повезаности је то што она увек заузима простор величине n^2, независно од тога колико грана има граф. Сваком чвору графа придружује се вектор дужине n. Ако је број грана у графу мали, већина елемената матрице повезаности биће нуле.

Уместо да се и све непостојеће гране експлицитно представљају у матрици повезаности, могу се формирати повезане листе од јединица из i-те врсте, $i = 1, 2, \ldots, n$. Овај други начин представљања графа зове се *листа повезаности*. Сваком чвору придружује се повезана листа, која садржи све гране суседне чвору (односно гране ка суседним чворовима). Листа може бити уређена према редним бројевима чворова на крајевима њених грана. Граф је представљен вектором листа. Сваки елеменат вектора садржи име (индекс) чвора и показивач на његову листу чворова. Ако је граф статички, односно нису дозвољена уметања и брисања, онда се листе могу представити векторима на следећи начин. Користи се вектор дужине $|V| + |E|$. Првих $|V|$ чланова вектора су придружени чворовима. Компонента вектора придружена чвору v_i садржи индекс почетка списка чворова суседних чвору v_i, $i = 1, 2, \ldots, n$. На слици 22 приказана су на једном примеру оба начина представљања графа. Са матрицама повезаности је једноставније радити. С друге стране, листе повезаности су ефикасније за графове са малим бројем грана. У пракси се често ради са графовима који имају знатно мање грана од максималног могућег броја ($n(n-1)/2$ неусмерених, односно $n(n-1)$ усмерених грана), и тада је обично боље користити листе повезаности.

3.9. Резиме

Структуре података могу се поделити на статичке и динамичке. Вектори (низови) су статичке структуре. Величина низа (или бар тачна горња граница за њу) мора се унапред знати. С друге стране, рад са низом је врло ефикасан. Повезане листе су динамичке. Оне се могу повећавати и смањивати, и нема ограничења на њихову величину, под условом да има довољно меморијског простора).

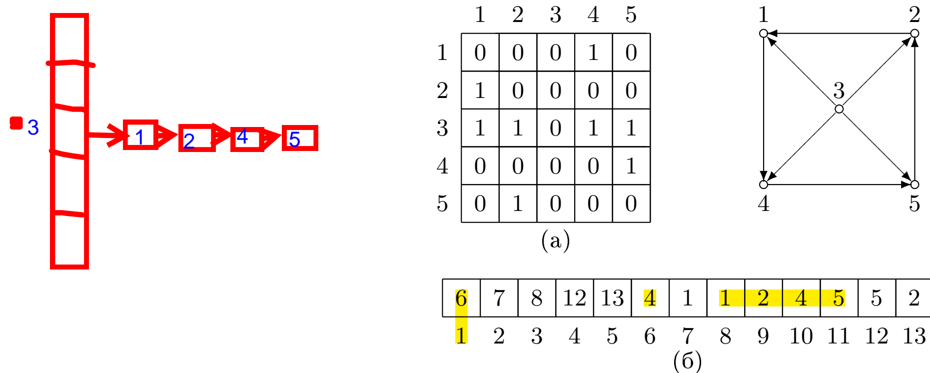


Рис. 22. Представљање графа матрицом повезаности (а), односно листом повезаности (б).

Структуре података могу се поделити на једнодимензионалне и вишедимензионалне структуре. Низови и повезане листе су једнодимензионални. Једина структура коју они могу да представљају је редослед елемената. Стабла представљају нешто више од једнодимензионалне структуре — хијерархију. Може се радити и са вишедимензионалним низовима, односно повезаним листама.

Појам апстрактног типа података је врло користан. Он омогућује да се пажња концентрише на операције над елементима структуре, независно од конкретног типа података. Описане су реализације речника, листе са приоритетом и структуре за формирање унија подскупова.

Хеш табеле су добро решење за смештање података без икакве посебне структуре. Хеш табеле се не могу користити ако приступ елементу не зависи само од кључа елемента. На пример, да би се нашао минимални елеменат у хеш табели, мора се прегледати цела табела.

Задаци

3.1. Трансформисати једноставну рекурзивну процедуру претраге бинарног стабла претраге у нерекурзивну процедуру.

3.2. Формирати неопадајући низ од кључева елемената бинарног стабла претраге. Колика је сложеност конструисаног алгоритма?

3.3. Елементи скупа A (сви су различити) смештени су у бинарно стабло претраге. Конструисати алгоритам који за дати елеменат $a \in A$ одређује следећи по величини елеменат скупа A.

3.4. Нека је A вектор који служи за имплицитно представљање хипа. Колики најмањи број елемената хипа може да заузме низ дужине 16?

3.5. Претпоставимо да се АВЛ стабло користи за реализацију листе са приоритетом. Колика је сложеност основних операција?

3.6. Описати реализацију апстрактног типа података који подржава следеће операције:

umetni(x): уметање се извршава и ако је број x већ једном претходно уметнут у структуру података; другим речима, структура података треба да памти дупликате;
ukloni(y): уклони *произвољан* елеменат из структуре података и додели га променљивој y. Ако постоји више копија истог елемента, уклања се само једна од њих.
Овај апстрактни тип података може се назвати **базен**. Он се може искористити за смештање послова. Нови послови се генеришу и умећу у базен, а кад неки радник постане расположив, уклања се неки посао. Све операције треба да се извршавају за време $O(1)$.

3.7. Модификовати базен, тип података из задатка 3.6, на следећи начин: претпоставља се да се произвољан елеменат може појавити највише једном у структури података. Уметање мора да буде праћено провером постојања дупликата. Реализовати исте операције као у претходном случају, али са провером постојања дупликата. Колика је сложеност сваке операције у најгорем случају?

3.8. Друга варијанта типа података базен (видети задатке 3.6, 3.7) је следећа. Претпоставимо да су свим елементима придружени бројеви из опсега од 1 до n, при чему n није велико, па је на располагању меморија величине $O(n)$. Сваки елеменат може се појавити највише једном. Конструисати алгоритме за *уметање* и *уклањање* (као што је дефинисано у задатку 3.6), временске сложености $O(1)$.

3.9. Конструисати алгоритам за формирање хипа који садржи све елементе два хипа величине n и m. Хипови су представљени експлицитно (сваки чвор има показиваче на два своја сина). Временска сложеност алгоритма у најгорем случају треба да буде $O(\log(m+n))$.

3.10. Описати реализацију апстрактног типа података који подржава следеће операције:
Umetni(x): уметни кључ x у структуру података, ако га тамо већ нема;
Obriši(x): обриши кључ x из структуре података (ако га има);
Naredni(x): пронађи најмањи кључ у структури података који је већи од x.
Извршавање сваке од ових операција треба да има временску сложеност $O(\log n)$ у најгорем случају, где је n број елемената у структури података.

3.11. Описати реализацију апстрактног типа података који подржава следеће операције:
Umetni(x): уметни кључ x у структуру података, ако га тамо већ нема;
Obriši(x): обриши кључ x из структуре података (ако га има);
Naredni(x, k): пронађи у структури података k-ти најмањи кључ, међу онима који су већи од x.
Извршавање сваке од ових операција треба да има временску сложеност $O(\log n)$ у најгорем случају, где је n број елемената у структури података.

3.12. Конкатенација је операција над два скупа који задовољавају услов да су сви кључеви у једном скупу мањи од свих кључева у другом скупу; резултат конкатенације је унија скупова. Конструисати алгоритам за конкатенацију два бинарна стабла претраге у једно. Временска сложеност у најгорем случају треба да буде $O(h)$, где је h већа од висина два стабла.

3.13. Конструисати алгоритам за конкатенацију (дефинисану у задатку 3.12) два АВЛ стабла у једно. Сложеност алгоритма у најгорем случају треба да буде $O(h)$, где је h већа од висина два АВЛ стабла.

3.14. (а) Како се може генерисати случајно АВЛ стабло, код кога фактор равнотеже у сваком унутрашњем чвору има вредност из скупа $\{-1, 0, 1\}$ са равномерном расподелом вероватноћа?

(б) За случајно АВЛ стабло задате висине h (са расподелом вероватноћа дефинисаном конструкцијом под (а)) показати да је очекивана дужина пута од критичног чвора до чвора уметања (новог чвора са случајно изабраним кључем) константна, и да не зависи од h.

3.15. Са улаза се задаје низ природних бројева чији се крај означава нулом. Конструисати алгоритам који бројеве датог низа смешта у бинарно стабло, при чему за сваки чвор стабла у сваком кораку алгоритма разлика броја чворова у левом и десном подстаблу не сме да премаши један.

3.16. Одредити изглед АВЛ стабла добијеног уметањем редом бројева $1, 2, \ldots, n$, полазећи од празног стабла. Колика је висина тог стабла?

3.17. Нека су T_1 и T_2 два произвољна бинарна стабла са по n чворова. Доказати да постоји низ од највише $2n$ ротација које T_1 трансформишу у T_2.

3.18. **Унија** два неусмерена графа $G = (V, E)$ и $H = (U, F)$ је нови граф $J = (W, D)$, такав да је $W = V \cup U$ и $D = E \cup F \cup (V \times U)$, тј. J поред грана G и H, садржи и све гране које повезују неки чвор из V са неким чвором из U. Предложите погодну репрезентацију графова која омогућује ефикасно израчунавање унија графова.

3.19. Нека је $S = \{s_1, s_2, \ldots, s_m\}$ врло велики скуп, издељен у k дисјунктних блокова. Претпоставимо да нам је на располагању процедура **Koji_blok** која за задати елеменат s_i даје **Koji_blok**(s_i), редни број блока који садржи s_i; ова процедура ради за константно време. Циљ је омогућити рад са малим подскуповима T скупа S, и то следеће операције над T: **Umetni**(s_i), **Obriši**(s_i), **Obriši_blok**(j) (брисање свих елемената T који припадају блоку j). На почетку је T празан скуп. Сложеност сваке операције треба да буде $O(\log n)$ у најгорем случају, где је n текући број елемената у T. **Obriši_blok** само уклања елементе из структуре података — није неопходно физичко брисање сваког од тих елемената. Оба броја m и k могу бити врло велики, па се не може користити табела величине m или k. Међутим, n је релативно мало, и на располагању је простор $O(n)$.

3.20. * Нека је $A[1]$, $A[2]$, $\ldots A[n]$ низ реалних бројева. Посматрајмо следеће две операције: **Dodaj**(i, y): додати вредност y i-том броју, и **Parcijalna_suma**(i): израчунати суму првих i бројева, $\sum_{j=1}^{i} A[j]$. Број елемената остаје фиксиран — нема уметања или брисања; једино се мењају неке вредности бројева. Конструисати алгоритам за извршавање низа ових двеју операција, тако да се обе извршавају за $O(\log n)$ корака. Може се користити помоћни низ дужине n.

3.21. Конструисати структуру података за чување скупа елемената од којих сваки има кључ и вредност. Структура података треба да подржава следеће операције:
Odredi_Vrednost(x): одредити вредност придружену елементу са кључем x (нула, ако x није у скупу);
Umetni(x, y): уметни нови елеменат са кључем x и вредношћу y;
Obriši(x): обриши елеменат са кључем x;
Dodaj(x, y): додај y текућем елементу са кључем x;
Dodaj_svima(y): додај y *свим* елементима скупа.
Време извршавања сваке од ових операција у најгорем случају треба да буде $O(\log n)$.

Глава 4

Конструкција алгоритама индукцијом

4.1. Увод

У овом поглављу биће наведени једноставни примери конструкције алгоритама коришћењем индуктивног приступа, који илуструју основне принципе и технике. Идеја је да се искористи принцип домина: све нанизане домине ће попадати ако се поруши прва, и ако се после рушења n-те обавезно руши $(n+1)$-а домина. Дакле, да би се решио неки проблем, треба решити неки његов мали случај, а затим показати како се решење задатог проблема може конструисати полазећи од (решених) мањих верзија истог проблема. Имајући на уму ово, пажња се може сконцентрисати на налажење начина за свођење проблема на мање проблеме.

4.2. Израчунавање вредности полинома

Најпре ћемо размотрити један једноставан проблем: израчунавање вредности задатог полинома у задатој тачки.

Проблем. Дати су реални бројеви a_n, $a_{n-1}, \ldots$, a_1, a_0 и реални број x. Израчунати вредност полинома $P_n(x) = \sum_{i=0}^{n} a_i x^i$.

Улазни подаци за проблем су $n+2$ броја. Индуктивни приступ решавању проблема састоји се у свођењу решења на решење мањег проблема. Дакле, проблем покушавамо да сведемо на мањи, који се затим решава рекурзивно. Најједноставније је свођење на упрошћени проблем добијен од полазног уклањањем a_n. Тада имамо проблем израчунавања вредности полинома

$$P_{n-1}(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0.$$

То је исти проблем, али са једним параметром мање.

Индуктивна хипотеза. Претпоставимо да знамо да израчунамо вредност полинома задатог коефицијентима $a_{n-1}, \ldots$, a_1, a_0 у тачки x, тј. знамо да израчунамо вредност $P_{n-1}(x)$.

Ова хипотеза је основа за решавање проблема индукцијом. Случај $n = 0$ (израчунавање вредности израза a_0) је тривијалан. Затим се мора показати како се полазни проблем (израчунавање $P_n(x)$) може решити коришћењем решења мањег проблема (вредности $P_{n-1}(x)$). У овом случају је то очигледно: треба израчунати x^n, помножити га са a_n и резултат сабрати са $P_{n-1}(x)$:

$P_n(x) = a_n x^n + P_{n-1}(x)$. Може се помислити да је коришћење индукције овде непотребно: оно само компликује врло једноставно решење. Описани алгоритам је еквивалентан израчунавању вредности полинома редом члан по члан. Испоставиће се ипак да овај приступ има своју снагу.

Описани алгоритам је тачан, али неефикасан, јер захтева $n+(n-1)+\cdots+1 = n(n+1)/2$ множења и n сабирања. Коришћењем индукције на други начин добија се боље решење.

Запажамо да у овом алгоритму има много поновљених израчунавања: степен x^n израчунава се сваки пут "од почетка". Велики број множења може се уштедети ако се при израчунавању x^n искористи x^{n-1}. Побољшање се реализује укључивањем израчунавања x^{n-1} у индуктивну хипотезу.

Појачана индуктивна хипотеза. Знамо да израчунамо вредност полинома $P_{n-1}(x)$ *и* вредност x^{n-1}.

Ова индуктивна хипотеза је јача, јер захтева израчунавање x^{n-1}, али се лакше проширује (јер је сада једноставније израчунати x^n). Да би се израчунало x^n, довољно је извршити једно множење. После тога следи још једно множење са a_n и сабирање са $P_{n-1}(x)$. Укупно је потребно извршити $2n$ множенја и n сабирања. Занимљиво је запазити да иако индуктивна хипотеза захтева више израчунавања, она доводи до смањења укупног броја операција. Добијени алгоритам изгледа добро у сваком погледу: ефикасан је, једноставан и једноставно се реализује. Ипак, постоји и бољи алгоритам. До њега се долази коришћењем индукције на нови, трећи начин.

Редукција проблема уклањањем највишег коефицијента a_n је очигледан корак, али то ипак није једина расположива могућност. Уместо тога се може уклонити коефицијент a_0, и свести проблем на израчунавање вредности полинома са коефицијентима $a_n, a_{n-1}, \ldots, a_1$, односно полинома

$$\tilde{P}_{n-1}(x) = \sum_{i=1}^{n} a_i x^{i-1} = a_n x^{n-1} + a_{n-1}x^{n-2} + \cdots + a_1.$$

Приметимо да је a_n овде $(n-1)$-и коефицијенат, a_{n-1} је $n-2$-и коефицијенат, и тако даље. Дакле, имамо нову индуктивну хипотезу.

Индуктивна хипотеза (обрнути редослед)**.** Умемо да израчунамо вредност полинома $\tilde{P}_{n-1}(x)$ са коефицијентима $a_n, a_{n-1}, \ldots, a_1$ у тачки x.

Оваква индуктивна хипотеза је погоднија јер се лакше проширује. Пошто је $P_n(x) = x\tilde{P}_{n-1}(x) + a_0$, за израчунавање $P_n(x)$ полазећи од $\tilde{P}_{n-1}(x)$ довољно је једно множење и једно сабирање. Израчунавање се може описати следећим изразом:

$$a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 = ((\cdots((a_n x + a_{n-1})x + a_{n-2})\cdots)x + a_1)x + a_0,$$

познатим као Хорнерова шема. Програм на псеудо паскалу који реализује овај алгоритам приказан је на слици 1.

Алгоритам Vrednost_polinoma($\mathbf{a}, x$);
Улаз: $\mathbf{a} = a_0, a_1, \ldots, a_n$ (коефицијенти полинома) и x (реални број).
Излаз: P (вредност полинома у тачки x).
begin
    $P := a_n$;
    for $i := 1$ **to** n **do**
        $P := x \cdot P + a_{n-i}$
end

Рис. 1. Израчунавање вредности полинома у задатој тачки.

Сложеност. Алгоритам обухвата n множења, n сабирања и захтева само једну нову меморијску локацију. Последњи алгоритам је не само ефикаснији од претходних, него је и одговарајући програм једноставнији.

Примедба. Индукција нам омогућује да се сконцентришемо на проширивање решења мањих на решења већих проблема. Претпоставимо да хоћемо да решимо $P(n)$, односно проблем P који зависи од неког параметра n (обично је то величина проблема). Тада полазимо од произвољног примера проблема $P(n)$ и покушавамо да га решимо користећи претпоставку да је $P(n-1)$ већ решен. Постоји много начина да се постави индуктивна хипотеза, а исто тако и више начина да се она искористи. Приказаћемо више таквих метода, и демонстрирати њихову моћ при конструкцији алгоритама.

Пример израчунавања вредности полинома илуструје флексибилност која нам је на располагању при коришћењу индукције. Идеја која је довела до Хорнерове шеме састоји се у томе да се улаз обрађује слева удесно, уместо интутивног редоследа здесна улево. Друга уобичајена могућност избора редоследа појављује се при разматрању структуре у облику стабла: она се може пролазити одозго наниже или одоздо навише. Даље, могуће је параметар n повећавати за два (или више), уместо само за један, а постоје и многобројне друге могућности. Штавише, понекад најбољи редослед примене индукције није исти за све улазе. Може се показати корисним конструкција алгоритма који би проналазио најбољи редослед примене индукције. Овакве могућности илустроваћемо неким примерима.

4.3. Максимални индуковани подграф

Посматрајмо следећи проблем. Организујете конференцију за научнике који се баве разнородним дисциплинама и имате списак особа које бисте желели да позовете. Претпоставка је да ће се сваки од њих одазвати позиву ако му се буде указала могућност плодне размене идеја. За сваког научника можете да направите листу оних (са списка) са којима ће он вероватно имати интеракцију. Нека је k задати број. Ви бисте волели да позовете што је могуће више

људи, а да при томе гарантујете свакоме од њих могућност да размени мисли са бар k других особа. Ваш проблем није да организујете те интеракције, нити да обезбедите довољно времена за њих. Ви једино желите да обезбедите њихово присуство на конференцији. Како изабрати особе које треба позвати? Описани проблем одговара следећем графовском проблему. Нека је $G = (V, E)$ неусмерени граф. Граф $H = (U, F)$ је индуковани подграф графа G ако $U \subseteq V$ и F садржи све гране из E којима су оба краја у U, видети пример на слици 2. Степен чвора је број њему суседних чворова. Чворови графа одговарају научницима, а грана између два чвора постоји ако одговарајући научници могу да размењују идеје. Индуковани подграф одговара подскупу научника.

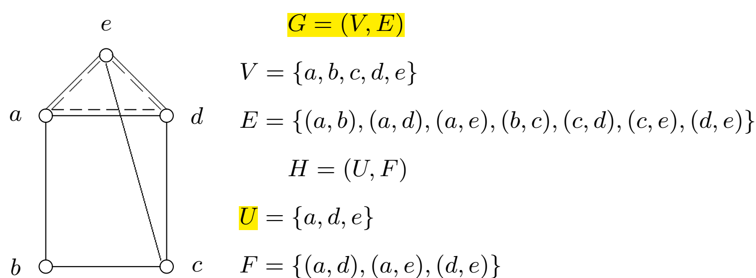


Рис. 2. Пример индукованог подграфа. Гране су неуређени парови чворова.

Проблем. За задати неусмерени граф $G = (V, E)$ и задати природни број k пронаћи максимални индуковани подграф $H = (U, F)$ графа G (тј. индуковани подграф са максималним бројем чворова) уз услов да сви чворови подграфа H имају степен бар k, или установити да такав индуковани подграф не постоји.

Директни приступ решавању овог проблема састоји се у уклањању свих чворова степена мањег од k. После уклањања ових чворова, заједно са гранама које су им суседне, степени преосталих чворова се у општем случају смањују. Кад степен неког чвора постане мањи од k, тај чвор треба уклонити из графа. Није међутим јасан редослед којим чворове треба уклањати. Треба ли најпре уклонити све чворове степена $< k$, а затим обрађивати чворове са умањеним степеновима? Или треба уклонити један чвор степена $< k$, па наставити са чворовима којима су смањени степенови? (Ова два приступа одговарају претрази у ширину, односно претрази у дубину, техникама које ће бити размотрене у одељку 6.3). Да ли ће оба приступа дати исти резултат? Да ли ће резултујући граф имати највећи могући број чворова? На ова питања лако је дати одговор коришћењем следећег приступа.

Уместо да алгоритам посматрамо као низ корака које треба извршити да би се дошло до резултата, можемо да себи поставимо за циљ *доказивање теореме* да алгоритам постоји. Потребно је пронаћи максимални индуковани подграф који задовољава задате услове.

Индуктивна хипотеза. Умемо да пронађемо максимални индуковани подграф са чворовима степена $\geq k$ за граф са мање од n чворова.

Треба да докажемо да је ова "теорема" тачна за базни случај, а затим да из њене тачности за $n-1$ следи њена тачност за n. Први нетривијални случај појављује се за $n = k+1$, јер ако је $n \leq k$ онда су степени свих чворова мањи од k. Ако је $n = k+1$ онда само у једном случају постоји индуковани подграф са степенима чворова $\geq k$: то је случај потпуног графа (графа са свим могућим гранама). Претпоставимо даље да је $G = (V, E)$ произвољан граф са $n > k+1$ чворова. Ако су степени свих чворова у G већи или једнаки од k, онда граф $H = G$ задовољава услове и доказ је завршен. У противном постоји чвор v степена $< k$. Очигледно је да степен чвора v остаје $< k$ и у сваком индукованом подграфу графа G, па према томе v не припада ни једном подграфу који задовољава услове проблема. Дакле, v се може уклонити из G заједно са свим њему суседним гранама, не мењајући услове теореме. После уклањања v граф има $n-1$ чвор, па према индуктивној хипотези ми умемо да решимо проблем до краја.

Коментар. Један од могућих начина за упрошћавање неког проблема је елиминација неког од његових елемената. У овом примеру примена индукције била је једноставна, јер је било јасно које чворове и како треба елиминисати. У општем случају процес елиминације не мора да буде једноставан. Видећемо пример у коме се два елемента обједињавају, чиме се број елемената смањује за један (одељак 5.5); затим пример у коме се елиминишу ограничења на проблем уместо да се елиминишу делови улаза (одељак 6.7); и на крају пример у коме се најпре конструише посебан алгоритам за избор елемената који се могу елиминисати (одељак 4.5). У примеру који следи такође ће бити илустровано елиминисање одговарајућих елемената.

4.4. Налажење бијекције

Нека је f функција која пресликава коначан скуп A у самог себе, $f : A \to A$. Без губитка општости може се претпоставити да је $A = \{1, 2, \ldots, n\}$. Претпостављамо да је функција f представљена вектором f дужине n, тако да $f[i]$ садржи вредност $f(i) \in A$, $i = 1, 2, \ldots, n$. Функција f је бијекција, ако за произвољан елеменат $j \in A$ постоји елеменат $i \in A$ такав да је $f(i) = j$. Функција се може представити дијаграмом као на слици 3, тако да сваком елементу A одговара по један леви и један десни чвор, а гране дефинишу пресликавање. Функција приказана на слици 3 очигледно није бијекција.

Проблем. Нека је задат коначан скуп A и пресликавање $f : A \to A$. Одредити подскуп $S \subseteq A$ са највећим могућим бројем елемената тако да буде $f(S) \subseteq S$ и да рестрикција f на S буде бијекција.

Ако је f бијекција, онда комплетан скуп A задовољава услове проблема, па је максимални подскуп $S = A$. Ако је пак $f(a) = f(b)$ за неке $a \neq b$, онда S не може да садржи и a и b. На пример, скуп S у проблему са слике 3 не може

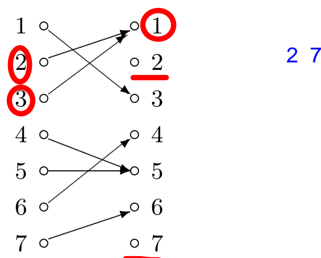


Рис. 3. Пример пресликавања скупа у самог себе.

да садржи оба елемента 2 и 3, јер је $f(2) = f(3) = 1$. Међутим, није свеједно који ће елеменат од ова два бити елиминисан. Ако на пример елиминишемо 3, онда се због $f(1) = 3$ мора елиминисати и 1. Затим се на исти начин због $f(2) = 1$ мора елиминисати и 2. Добијени подскуп сигурно није максималан, јер је могуће елиминисати само елеменат 2 уместо 1, 2 и 3. Решење проблема на слици 3 је подскуп $\{1, 3, 5\}$, што се може установити нпр. провером свих могућих подскупова $S \subset A$. Поставља се питање налажења општег метода за одлучивање које елементе треба укључити у S.

На срећу, имамо известан маневарски простор приликом одлучивања како проблем свести на мањи. Величина проблема може се смањити било налажењем елемента који припада скупу S, било налажењем елемента који не припада скупу S. Испоставља се да је друга могућност боља. Користићемо следећу једноставну индуктивну хипотезу.

Индуктивна хипотеза. Знамо да решимо проблем за скупове са $n-1$ елемената.

База индукције је тривијална: ако скуп A има један елемент, он се мора пресликавати у себе, па је f на A бијекција. Претпоставимо даље да имамо произвољан скуп A од n елемената и да тражимо подскуп S који задовољава услове проблема. Тврдимо да ако елеменат a не припада скупу $f(A)$, онда a не може припадати S; другим речима, елеменат a за који у одговарајући чвор на десној страни дијаграма не улази ни једна грана, не може бити у S. Заиста, из услова проблема следи да је $f(S) = S$, а то је, ако се претпостави да $a \in S$ — немогуће, јер по претпоставци $a \notin f(S)$. Дакле, ако постоји такав елеменат a, ми га елиминишемо из скупа. Сада имамо скуп $A' = A \setminus \{a\}$ са $n-1$ елементом, који f пресликава у самог себе; према индуктивној хипотези ми знамо да решимо проблем са скупом A'. Ако такво a не постоји, онда је пресликавање бијекција, и проблем је решен.

Суштина овог решења је да се a *мора* уклонити. Доказали смо да a не може припадати S. У томе је снага индукције: оног тренутка кад се елеменат уклони и тиме смањи величина проблема, задатак је решен. При томе се мора водити рачуна да мањи проблем буде потпуно исти као полазни, изузимајући величину. Једини услов који f и A треба да задовољавају је $f(A) \subseteq A$. Овај

услов задовољен је и на скупу $A' = A \setminus \{a\}$, јер се ни један елеменат не пресликава у a. У тренутку кад више нема елемената који се могу уклонити, скуп S је пронађен.

Реализација. Алгоритам смо описали као рекурзивну процедуру. У сваком кораку проналазимо елеменат у кога се ни један други не пресликава, уклањамо га и настављамо рекурзивно. Реализација алгоритма ипак не мора да буде рекурзивна. Сваком елементу i, $1 \le i \le n$ придружује се бројач $c[i]$, чија је почетна вредност једнака броју елемената који се пресликавају у i. Вредности $c[i]$ могу се израчунати у n корака пролазећи кроз вектор f и инкрементирајући (повећавајући за један) при томе одговарајуће бројаче. Затим се у листу стављају сви елементи којима је вредност бројача добила вредност 0. У сваком кораку се први елеменат j из листе уклања из листе и из скупа, декрементира се $c[f[j]]$, а ако $c[f[j]]$ добије вредност 0, $f[j]$ се ставља у листу. Решавање је завршено у тренутку кад је листа празна. Алгоритам је приказан на слици 4. Није тешко проверити да се за улаз представљен на слици 3 добија $S = \{1, 3, 5\}$.

```
Алгоритам Bijekcija(f, n);
Улаз: f (низ природних бројева између један и n).
Излаз: S (подскуп улазног скупа такав да је f бијекција на S).
begin
    S := A; {A = {1, 2, ... n}}
    for j := 1 to n do c[j] := 0;
    for j := 1 to n do инцремент c[f[j]];
    for j := 1 to n do
        if c[j] = 0 then  стави j у Списак  {листа елемената за елиминацију}
    while Списак није празан do
        скини i са врха листе Списак;
        S := S \ {i};
        c[f[i]] = c[f[i]] − 1;
        if c[f[i]] = 0 then  стави f[i] у Списак
end
```

Рис. 4. Алгоритам *Bijekcija*.

Сложеност. За уводни део (иницијализацију) треба извршити $O(n)$ корака. Сваки елеменат се у листу може ставити највише једном, а операције потребне да се елеменат уклони из листе извршавају се за време ограничено константом. Према томе, укупан број корака је $O(n)$.

Коментар. У овом примеру величина проблема је смањена уклањањем елемената из скупа. Тражили смо најједноставнији начин за уклањање елемента тако да се не промене услови проблема. Пошто је једини захтев био да

функција пресликава A у самог себе, избор елемента у кога се ни један други елеменат не пресликава је природан.

4.5. Проблем проналажења звезде

Следећи пример је интересантан по томе што за решавање проблема није неопходно прегледати све улазне податке, па чак ни њихов значајан део. Међу n особа *звезда* је особа која никога не познаје, а коју сви остали познају. Проблем је идентификовати звезду (ако она постоји) постављајући питања облика "Извините, да ли познајете ону особу?" Претпоставља се да су сви одговори тачни, а да ће чак и звезда одговарати на питања. Циљ је минимизирати укупан број питања. Пошто парова особа има укупно $n(n-1)/2$, у најгорем случају треба поставити $n(n-1)$ питања, ако се питања постављају без неке стратегије. На први поглед није јасно може ли се жељени циљ постићи са гарантовано мањим бројем постављених питања.

Проблем се може преформулисати у графовски. Формирамо усмерени граф са чворовима који одговарају особама, у коме грана од особе A ка особи B постоји ако A познаје B. Звезда одговара **понору** графа, чвору У који улази $n-1$ грана, а из кога не излази ни једна грана. Јасно је да граф може имати највише један понор. Улаз проблема се описује $n \times n$ матрицом повезаности M, чији елеменат M_{ij} је једнак 1 ако особа i познаје особу j, односно 0 у противном; дијагонални елементи су једнаки нули.

Проблем. Дата је $n \times n$ матрица повезаности M. Установити да ли постоји индекс i, такав да су у M сви елементи i-те колоне (сем i-тог) једнаки 1, и да су сви елементи i-те врсте једнаки 0.

Базни случај са две особе је једноставан. Посматрајмо као и обично разлику између проблема са $n-1$ особом и проблема са n особа. Претпостављамо да знамо да пронађемо звезду међу првих $n-1$ особа индукцијом. Пошто може да постоји највише једна звезда, постоје три могућности:

(1) звезда је једна од првих $n-1$ особа;
(2) звезда је n-та особа;
(3) звезда не постоји.

Први случај је најједноставнији: треба само проверити да ли n-та особа познаје звезду, односно да ли је тачно да звезда не познаје n-ту особу. Преостала два случаја су тежа, јер да би се проверило да ли је n-та особа звезда, треба поставити $2(n-1)$ питања. Ако поставимо $2(n-1)$ питања у n-том кораку, онда ће укупан број питања бити $n(n-1)$, што хоћемо да избегнемо. Потребно је дакле променити приступ проблему.

Идеја је да се проблем посматра "уназад". Пошто је тешко идентификовати звезду, покушајмо да пронађемо особу која није звезда: у сваком случају, број не-звезда много је већи од броја звезда. Ако елиминишемо некога из разматрања, параментар n који дефинише величину проблема смањује се на $n-1$. При томе уопште није битно кога ћемо елиминисати! Претпоставимо да особу A питамо да ли познаје особу B. Ако A познаје B, онда A није звезда, а ако пак

A не познаје B, онда B није звезда. У оба случаја се једна особа елиминише постављањем само једног питања!

Ако се сада вратимо на три могућа случаја при преласку са $n-1$ на n, видимо да је новост у томе да n-ту особу не бирамо произвољно. Елиминацијом особе A или B проблем сводимо на случај са $n-1$ особом, при чему смо сигурни да до случаја 2. неће доћи, јер елиминисана особа не може бити звезда. Ако је наступио случај 3, односно нема звезде међу првих $n-1$ особа, онда нема звезде ни међу свих n особа. Остаје први случај, који је лак: ако међу првих $n-1$ особа постоји звезда, онда се са два допунска питања може проверити да ли је то звезда и за комплетан скуп. Ако није, онда нема звезде.

Алгоритам се састоји у томе да питамо особу A да ли познаје особу B и елиминишемо или особу A или особу B, зависно од добијеног одговора. Нека је елиминисана нпр. особа A. Индукцијом (рекурзивно) се проналази звезда међу преосталих $n-1$ особа. Ако међу њима нема звезде, решавање је завршено. У противном, проверава се да ли је тачно да A познаје звезду, а да звезда не познаје A.

Реализација. Као и у случају алгоритма из претходног одељка, ефикасније је реализовати алгоритам за налажење звезде итеративно, него рекурзивно. Алгоритам се дели у две фазе. У првој фази се елиминишу све особе сем једног кандидата за звезду, а друга фаза је неопходна да се установи да ли кандидат јесте звезда. Почињемо са n кандидата, за које можемо да претпоставимо да су смештени на стек (листу из које се најпре вади последњи уписани елеменат). За сваки пар кандидата ми можемо да елиминишемо једног, питањем да ли познаје другог. Почињемо узимајући прва два кандидата са стека и елиминишући једног од њих. Даље у сваком кораку формирамо пар од преосталог кандидата и (све док је стек непразан) наредне особе са стека, и елиминишемо једног од њих. У тренутку кад стек постане празан, имамо само једног кандидата. Затим се проверава да ли је тај кандидат звезда. Алгоритам је приказан на слици 5. Стек је реализован експлицитно коришћењем индекса i, j, и $Naredni$.

Сложеност. Поставља се највише $3(n-1)$ питања: $n-1$ питања у првој фази да би се елиминисала $n-1$ особа, и највише $2(n-1)$ питања за проверу да ли је преостали кандидат звезда. Приметимо да величина улаза није n, него $n(n-1)$, број елемената матрице. Приказано решење показује да је могуће одредити звезду прегледајући највише $O(n)$ елемената матрице повезаности, иако је јасно да решење мора битно да зависи од свих $n(n-1)$ елемената матрице.

Коментар. Кључна идеја овог елегантног решења је паметно смањивање параметра n за 1. Пример показује да се понекад исплати уложити труд (у овом случају — једно питање) да би се редукција ефикасније извела. Не полази се од *произвољног*, него се бира *посебан* скуп од $n-1$ особе. Видећемо још примера у којима се значајно време троши на конструкцију погодног редоследа индукције, што се показује корисним.

```
Алгоритам Zvezda(Poznaje);
Улаз: Poznaje (n × n Булова матрица са нулама на дијагонали).
Излаз: Zvezda.
begin
    i := 1; {особа A}
    j := 2; {особа B}
    Naredni := 3; {следећи који ће се проверавати}
     {прва фаза: елиминишу се сви кандидати сем једног}
    while Naredni ≤ n + 1 do
        if Poznaje[i, j] then i := Naredni {елиминација i}
        else j := Naredni; {елиминација j}
        Naredni := Naredni + 1
     {по изласку из петље је или i = n + 1 или j = n + 1}
    if i = n + 1 then Kandidat := j
    else Kandidat := i
     {друга фаза: провера да ли је Кандидат звезда}
    Jeste := true; {true ако Kandidat јесте звезда}
    k := 1;
    while Jeste and k ≤ n do
        if Poznaje[Kandidat, k] then Jeste := false;
        if not Poznaje[k, Kandidat] then
            if Kandidat ≠ k then Jeste := false;
        k := k + 1;
    if Jeste then Zvezda := Kandidat
    else Zvezda := 0 {нема звезде}
end
```

Рис. 5. Алгоритам за проналажење звезде.

разлагања

4.6. Пример примене декомпозиције: максимум скупа правоугаоника

До сада смо имали примере са графовима и нумеричке проблеме. Сада ћемо се позабавити примером у коме треба исцртати неке фигуре. Нека су L, D и V реални бројеви такви да је $L < D$ и $V > 0$. Правоугаона функција (правоугаоник) са параметрима (L, D, V) је функција

$$f_{L,D,V}(x) = \begin{cases} 0, & x < L \text{ или } x \geq D \\ V, & L \leq x < D \end{cases} .$$

Проблем. Дато је n правоугаоника $f_{L_i,D_i,V_i}(x)$, $i = 1, 2, \ldots, n$. Потребно је одредити "контуру" овог скупа правоугаоника, односно функцију

$$f(x) = \max\{f_{L_i,D_i,V_i}(x) \,|\, i = 1, 2, \ldots, n\}$$

Пример улаза и излаза приказан је на слици 6: улаз су четири правоугаоника $(1,4,4)$, $(2,5,6)$, $(3,3,8)$ и $(7,4,10)$, слика 6(а), а резултујућа контура приказана је на слици 6(б). Излазна функција $f(x)$ може се описати скупом интервала на којима она има константну вредност, и вредностима у тим интервалима. Другим речима, $f(x)$ се задаје низом парова (x_i, v_i), $i = 1, 2, \ldots, k$ (при чему је $v_k = 0$), тако да је $f(x) = v_i$ за $x_i \leq x < x_{i+1}$, $i = 1, 2, \ldots, k$. Због једноставности претпоставља да је још $x_0 = -\infty$, $v_0 = 0$, $x_{k+1} = \infty$. У примеру са слике 6(б) функција $f(x)$ описује се низом парова $(1,4)$, $(2,5)$, $(6,3)$, $(7,4)$, $(10,0)$.

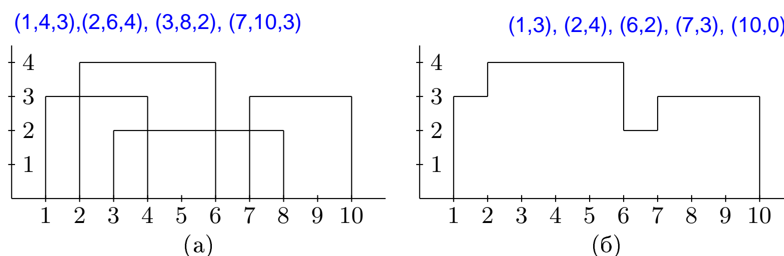


Рис. 6. Проблем налажења максимума скупа правоугаоника: (а) улаз; (б) излаз.

Директни приступ решавању овог проблема заснива се на проширивању решења корак по корак, тј. проширивању решења за првих $n-1$ правоугаоника новим, n-тим правоугаоником. База индукције $n = 1$ је једноставна: правоугаоник $f_{L_1,D_1,V_1}(x)$ претвара се у функцију $f_{(L_1,V_1),(D_1,0)}(x)$. Да би се максимуму

$$f_{n-1}(x) = f_{(x_1,v_1),(x_2,v_2),\ldots,(x_k,v_k)}(x)$$

прикључио n-ти правоугаоник $f_{L_n,D_n,V_n}(x)$, треба пронаћи такве индексе i, j, да је $x_i \leq L_n < x_{i+1}$, $x_j \leq D_n < x_{j+1}$, $0 \leq i \leq j \leq k$: вредност функције $f_{n-1}(x)$ може се при преласку на $f_n(x)$ променити само у оним интервалима константности са којима интервал $[L_i, D_i))$ има непразан пресек. Нови максимум је

$$f_n(x) = \begin{cases} f_{n-1}(x), & x < L_n \text{ или } x \geq D_n \\ \max\{f_{n-1}(x), V_n\}, & L_n \leq x < D_n \end{cases}.$$

Интервали константности $[x_i, x_{i+1})$ и $[x_j, x_{j+1})$ функције $f_{n-1}(x)$ у општем случају бивају тачкама L_n и D_n подељени на подинтервале $[x_i, L_n)$, $[L_n, x_{i+1})$, односно $[x_j, D_n)$, $[D_n, x_{j+1})$, у којима је вредност $f_n(x)$ константна; поред тога, у интервалима $[x_l, x_{l+1})$, $l = i+1, \ldots, j-1$ у којима $f_{n-1}(x)$ има вредност мању од V_n, $f_n(x)$ добија нову (у односу на $f_{n-1}(x)$) вредност V_n. У примеру на слици 7 је $n = 5$, $(L_n, D_n, V_n) = (5, 9, 4.5)$, и

$$\begin{aligned} f_{n-1}(x) &= f_{(1,4),(2,5),(5,6),(9,4),(10,0)}(x), \\ f_n(x) &= f_{(1,4),(2,5),(6,4.5),(9,4),(10,0)}(x). \end{aligned}$$

(6,3.5), (9,3)
(1,3), (2,4), (6,2), (7,3), (10,0)
5 9

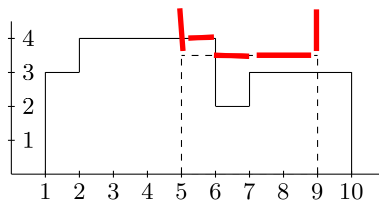


Рис. 7. Додавање новог правоугаоника максимуму претходних $n-1$ са слике 6.

Овај алгоритам је коректан, али није ефикасан: додавање n-тог правоугаоника функцији $f_{n-1}(x)$ захтева $O(n)$ корака, па је укупна сложеност $O(n^2)$. Алгоритам се може усавршити применом метода декомпозиције (односно "завади, па владај, енглески divide–and–conquer). Уместо преласка са $n-1$ на n, ефикасније је проширивати решење са $n/2$ на n правоугаоника. Базни случај је већ размотрен. Метод декомпозиције обухвата поделу улаза на мање подскупове, њихову обраду рекурзивним позивом процедуре и обједињавање добијених решења. Најчешће се проблем дели у потпроблеме подједнаке величине. Као што смо видели у поглављу 2, решење диференцне једначине $T(n) = T(n-1) + cn$ је $T(n) = O(n^2)$, док је решење једначине $T(n) = 2T(n/2) + cn$ дато са $T(n) = O(n \log n)$. Дакле, ако поделимо проблем на два једнака потпроблема и добијена решења потпроблема објединимо за линеарно време, онда је сложеност алгоритма $O(n \log n)$. Техника декомпозиције је врло корисна, у шта ћемо се више пута уверити.

разлагање

У проблему са правоугаоницима декомпозиција се заснива на запажању да је за додавање новог правоугаоника функцији $f_{n-1}(x)$ потребно у најгорем случају линеарно време, а да се за линеарно време могу обединити и две различите функције $f'_{n/2}(x)$ и $f''_{n/2}(x)$, решења два потпроблема за по $n/2$ правоугаоника. За асимптотски исто време постиже се много више користећи други приступ. Две у деловима константне функције могу се комбиновати у основи истим алгоритмом којим је комбинован један правоугаоник са једном таквом функцијом (слика 8). Прекидне тачке двеју функција $f'_{n/2}(x)$ и $f''_{n/2}(x)$ се пролазе истовремено слева у десно, идентификују се x-координате крајева интервала константности функције $f_n(x)$ и одређују вредности $f_n(x)$ у тим интервалима. Обједињавање се извршава за линеарно време, па је сложеност комплетног алгоритма $O(n \log n)$ у најгорем случају. Овај алгоритам је сличан алгоритму сортирања обједињавањем, одељак 5.3.3 (видети задатак 4.9).

Коментар. Трудите се да за уложено увек добијете што је могуће више. Ако ваш алгоритам садржи корак који је општији него што је потребно, размотрите могућност да тај корак примените на најтежи део проблема. Декомпозиција је овде корисна, јер до максимума користи корак обједињавања. Диференцне једначине из одељка 2.5.4 покривају већину уобичајених алгоритама заснованих на декомпозицији, па је корисно знати њихова решења.

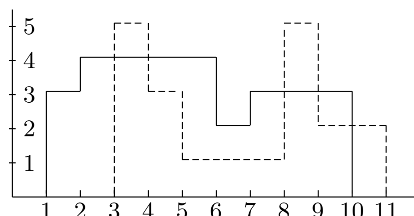


Рис. 8. Одређивање максимума две у деловима константне функције.

4.7. Израчунавање фактора равнотеже бинарног стабла

Нека је T бинарно стабло са кореном r. **Висина** чвора v је растојање од v до најдаљег потомка. Висина стабла је висина његовог корена. **Фактор равнотеже** чвора v дефинише се као разлика висина његовог левог и десног подстабла. У одељку 3.5 разматрана су АВЛ стабла — бинарна стабла чији сви чворови имају факторе равнотеже -1, 0 или 1. Овде ћемо се бавити произвољним бинарним стаблима. На слици 9 приказано је бинарно стабло код кога је уз сваки чвор уписан пар h/b, где је h висина чвора, а b његов фактор равнотеже.

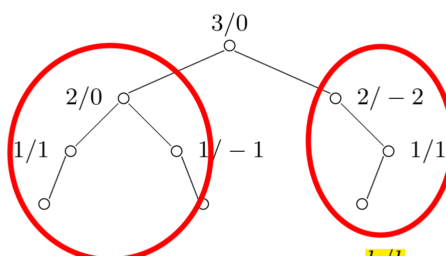


Рис. 9. Бинарно стабло са ознакама h/b уз унутрашње чворове (h – висина, b – фактор равнотеже чвора).

Проблем. За задато стабло T са n чворова израчунати факторе равнотеже свих чворова.

Покушаћемо са индуктивним приступом и најједноставнијом индуктивном хипотезом.

Индуктивна хипотеза. Знамо да израчунамо факторе равнотеже свих чворова у стаблима са $< n$ чворова.

Базни случај $n = 1$ је тривијалан. Ако је дато стабло са $n > 1$ чворова, можемо да уклонимо корен, затим да решимо проблем (индуктивно) за два подстабла која су преостала. Определили смо се за уклањање корена због тога што фактор равнотеже чвора v зависи само од чворова испод v. Због тога знамо факторе равнотеже свих чворова изузев корена. Међутим, фактор равнотеже корена зависи не од фактора равнотеже, него од висина његових

синова. Закључујемо да у овом случају једноставна индукција не решава проблем. Потребно је да знамо *висине* синова корена. Идеја је да налажење висина чворова прикључимо полазном проблему.

Појачана индуктивна хипотеза. Знамо да израчунамо факторе равнотеже и висине свих чворова у стаблима која имају $< n$ чворова.

Базни случај је поново тривијалан. Кад сада посматрамо корен произвољног стабла, његов фактор равнотеже можемо лако да одредимо као разлику висина његових синова. Можемо да одредимо и висину корена: то је већа од висина његових синова, увећана за један. Лако је проверити да се случај кад нема левог или десног подстабла, може решити тако да се висина непостојећег сина замени са -1.

Карактеристика овог алгоритма је да он решава нешто општији проблем. Уместо да рачунамо само факторе равнотеже, ми израчунавамо и висине чворова. Испоставља се да је општији проблем лакши, јер се висине лако рачунају. Ако је решење општије (јер је проблем проширен), онда индуктивни корак може бити једноставнији, јер поседујемо снажнија средства. Уобичајена грешка при решавању ширег проблема је да се заборави на постојање два параметра, и да се сваки од њих мора посебно израчунати.

4.8. Налажење максималног узастопног подниза

Проблем. Задат је низ $x_1,\ x_2,\dots x_n$ реалних бројева (не обавезно позитивних). Одредити подниз $x_i,\ x_{i+1},\dots x_j$ узастопних елемената са највећом могућом сумом.

За овакав подниз рећи ћемо да је **максимални подниз**. На пример, у низу $(2,-3,1.5,-1,3,-2,-3,3)$ максимални подниз је $1.5,-1,3$, са сумом 3.5. Може да постоји више максималних поднизова датог низа. Ако су сви чланови низа негативни, онда је максимални подниз празан (по дефиницији, сума празног низа је 0). Волели бисмо да имамо алгоритам који решава овај проблем и прегледа (пролази низ редом слева у десно) низ само једном.

Индуктивна хипотеза. Знамо да нађемо максимални подниз у низовима дужине $< n$.

За $n = 1$ максимални подниз је број x_1 ако је $x_1 \geq 0$, односно празан подниз ако је $x_1 < 0$. Посматрајмо низ $S = (x_1, x_2, \dots, x_n)$ дужине $n > 1$. Према индуктивној хипотези ми умемо да пронађемо максимални подниз S'_M низа $S' = (x_1, x_2, \dots, x_{n-1})$. Ако је S'_M празан подниз, онда су сви бројеви у низу S' негативни, па остаје да се размотри само x_n. Претпоставимо сада да је $S'_M = (x_i, x_{i+1}, \dots, x_j)$ за неке индексе i и j такве да је $1 \leq i \leq j \leq n-1$. Ако је $j = n-1$ (односно максимални подниз је суфикс), онда је лако ово решење проширити на S: ако је број x_n позитиван, онда он продужује S'_M, а у противном подниз S'_M је максималан и у S. Међутим, ако је $j < n-1$, онда постоје две могућности. Или S'_M остаје максимални подниз у S, или постоји други подниз који није максималан у S', али после додавања x_n постаје максималан у S.

Кључна идеја је да се **појача индуктивна хипотеза**. Она ће бити демонстрирана на примеру налажења максималног подниза, а затим ће нешто општије бити размотрена у следећем одељку. Проблем са размотреном индуктивном хипотезом је у томе што x_n може да продужи подниз који није максималан у S' и тако формира нови максимални подниз у S. Према томе, није довољно само познавање максималног подниза у S'. Међутим, x_n може да продужи само подниз који се завршава са x_{n-1}, односно суфикс низа S'. Претпоставимо да смо појачали индуктивну хипотезу тако да обухвати и налажење максималног суфикса, означеног са $S'_E = (x_k, x_{k+1}, \ldots, x_{n-1})$.

Појачана индуктивна хипотеза. У низовима дужине $< n$ умемо да пронађемо максимални подниз, и максимални суфикс.

Ако за подниз S' знамо оба ова подниза, алгоритам постаје јасан. Максимални суфикс проширујемо бројем x_n. Ако је добијена сума већа од суме (глобално) максималног подниза, онда имамо нови максимални подниз (такође и нови максимални суфикс). У противном, задржавамо претходни максимални подниз. Посао тиме није завршен: потребно је одредити и нови максимални суфикс. У општем случају нови максимални суфикс се не добија увек проширивањем старог бројем x_n. Може се догодити да је максимални суфикс који се завршава сабирком x_n негативан. Тада је нови максимални суфикс уствари празан подниз, са сумом 0. Алгоритам за налажење максималног подниза приказан је на слици 10.

```
Алгоритам Max_uzast_podniz(X, n);
Улаз: X (низ дужине n).
Излаз: Glob_max (сума максималног подниза).
begin
    Glob_max := 0; {почетна вредност глобално макс. суме}
    Suf_max := 0; {почетна вредност суме макс. суфикса}
    for i := 1 to n do
        if x[i] + Suf_max > Glob_max then
            Suf_max := Suf_max + x[i];
            Glob_max := Suf_max;
        else if x[i] + Suf_max > 0 then
            Suf_max := Suf_max + x[i];
        else Suf_max := 0
end
```

Рис. 10. Алгоритам за налажење максималног подниза задатог низа.

4.9. Појачавање индуктивне хипотезе

Појачавање индуктивне хипотезе је једна од најважнијих техника за доказивање теорема индукцијом. Приликом тражења доказа често се наилази на следећу ситуацију. Нека је теорема коју треба доказати означена са P. Индуктивна хипотеза може се означити са $P(< n)$, а доказ теореме своди се на доказивање тачности тврђења $P(< n) \Rightarrow P(n)$. У много случајева може се увести допунска претпоставка Q, која олакшава доказ. Другим речима, лакше је доказати $[P \wedge Q](< n) \Rightarrow P(n)$ него $P(< n) \Rightarrow P(n)$. Претпоставка може да изгледа коректна, али није јасно како се може доказати. Идеја се састоји у томе да се Q укључи у индуктивну хипотезу. Сада је потребно доказати да $[P \wedge Q](< n) \Rightarrow [P \wedge Q](n)$. Конјункција $P \wedge Q$ је јаче тврђење него само P, али се често јача тврђења лакше доказују. Процес се наставља, и после одређеног броја уведених претпоставки може да доведе до завршетка доказа. Проблем налажења максималног подниза је добар пример како се овај принцип може искористити за усавршавање алгоритама.

Најчешћа грешка приликом коришћења ове технике је превид чињенице да је додата допунска претпоставка у тренутку кад треба кориговати доказ. Другим речима, ми доказујемо тврђење $[P \wedge Q](< n) \Rightarrow P(n)$, заборављајући да је уведена допунска претпоставка Q. У примеру са максималним поднизом до ове грешке би дошло ако не бисмо израчунали нови максимални суфикс. Слично, у примеру са налажењем фактора равнотеже чворова бинарног стабла, оваква грешка била би ако се не израчуна висина корена стабла. Према томе, важно је прецизно регистровати промене индуктивне хипотезе.

Збир подскупа

4.10. Динамичко програмирање, проблем ранца

Претпоставимо да ранац треба напунити стварима. Ствари могу да буду различитог облика и величине, а циљ је да се у ранац упакује што је могуће више ствари. Уместо ранца то може да буде камион, брод или силицијумски чип, а проблем је паковање елемената. Постоји много варијанти овог проблема; овде ће бити разматрана варијанта са једнодимензионалним предметима.

Проблем. Дат је природан број K и n предмета различитих величина (тежина), тако да i-ти предмет има величину k_i, $1 \leq i \leq n$. Пронаћи подскуп предмета чија је сума величина једнака тачно K, или установити да такав подскуп не постоји.

Означимо проблем са $P(n, K)$, где n означава број предмета, а K величину (носивост) ранца. Величине предмета подразумевају се имплицитно, односно њихове величине нису део експлицитне ознаке проблема. На тај начин $P(i, k)$ означава проблем са првих i предмета и ранцем велицине k. Због једноставности ограничавамо са на проблем одлучивања да ли решење постоји. Започињемо са најједноставнијим индуктивним приступом.

Индуктивна хипотеза (први покушај)**.** Умемо да решимо $P(n-1, K)$.

Базни случај је једноставан: решење постоји акко је $k_1 = K$. Ако постоји решење проблема $P(n-1, K)$, онда је то и решење $P(n, K)$: у то решење не улази предмет величине k_n. У противном, ако не постоји решење $P(n-1, K)$, поставља се питање — може ли се искористити овај негативни резултат? Најпре закључујемо да n-ти предмет мора бити укључен у збир. Тада остали предмети морају стати у мањи ранац величине $K - k_n$. Проблем је тиме сведен на два мања проблема $P(n-1, K)$ и $P(n-1, K-k_n)$. Да би се решење комплетирало, требало би појачати индуктивну хипотезу. Потребно је имати решења не само за ранац величине K, него и за ранце величине мање од K.

Индуктивна хипотеза (други покушај)**.** Умемо да решимо $P(n-1, k)$ за све k, $0 \leq k \leq K$.

Ова индуктивна хипотеза омогућује нам да решимо $P(n, k)$ за $0 \leq k \leq K$. Базни случај $P(1, k)$ се лако решава: за $k = 0$ увек постоји решење (тривијално, нула сабирака), а у противном решење постоји акко је $k_1 = k$. Проблем $P(n, k)$ своди се на два проблема $P(n-1, k)$ и $P(n-1, k-k_n)$ (други проблем отпада ако је $k - k_n < 0$). Оба ова проблема се могу решити индукцијом. Пошто је свођење комплетно, алгоритам је конструисан. Међутим, овај алгоритам је неефикасан. Проблем величине n сведен је на *два* проблема величине $n-1$ (истина, у једном од потпроблема смањена је вредност k). Сваки од нових проблема своди се на по два наредна проблема, итд, што доводи до експоненцијалног алгоритма.

На срећу, у много случајева могуће је побољшати ефикасност приликом решавања оваквих проблема. Запажа се да укупан број различитих проблема не мора бити превелики: први параметар може да има највише n, а други највише K различитих вредности. Укупно је број различитих проблема највише nK. Експоненцијално време извршавања алгоритма је последица дуплирања броја проблема после сваког свођења. Пошто укупно има nK различитих проблема, јасно је да су неки од њих (ако је $2^n > nK$) више пута решавани. Корисна идеја је памтити сва нађена решења да би се избегло понављање решавања истих проблема. Овај приступ је комбинација појачавања индуктивне хипотезе и коришћења потпуне индукције. Размотрићемо сада како се овај приступ може практично реализовати.

За смештање свих израчунатих резултата користи се посебна матрица димензије $n \times K$. Елеменат (i, k) матрице садржи информације о решењу $P(i, k)$: Свођење по другој варијанти индуктивне хипотезе еквивалентно је израчунавању једне врсте на основу претходне врсте матрице. Сваки елеменат се израчунава на основу два елемента из претходне врсте. Ако је потребно одредити и подскуп са задатим збиром, онда уз сваки елеменат матрице треба сачувати и информацију о томе да ли је одговарајући елеменат скупа укључен у збир у том кораку. Пратећи ове информације уназад од елемента (n, K), може се реконструисати подскуп са збиром K. Алгоритам је приказан на слици 11, а у табели 1 приказан је један пример. Матрица P је због удобности димензије $(n+1) \times (k+1)$ и садржи елементе $P[i, k]$, $0 \leq i \leq n$, $0 \leq k \leq K$.

```
Алгоритам Ranac(S, K);
Улаз: S (вектор дужине n са величинама предмета) и K (величина ранца).
Излаз: P (матрица, тако да је P[i,k].Postoji = true акко постоји решење проблема
 ранца са првих i предмета, за величину ранца k; P[i,k].Pripada = true акко i-ти
 предмет припада том решењу).
begin
    P[0,0].Postoji := true;
    for k := 1 to K do
        P[0,k].Postoji := false;
        {елементи P[i,0] за i ≥ 1 биће израчунати на основу P[0,0]}
    for i := 1 to n do
        for k := 0 to K do  {израчунавање елемента P[i,k]}
            P[i,k].Postoji := false; {полазна вредност}
            if P[i-1,k].Postoji then
                P[i,k].Postoji := true;
                P[i,k].Pripada := false
            else if k - S[i] ≥ 0 then
                if P[i-1,k-S[i]].Postoji then
                    P[i,k].Postoji := true;
                    P[i,k].Pripada := true;
end
```

Рис. 11. Алгоритам за решавање проблема ранца.

	0	1	2	3	4	5	6	7	8	9	10	11
$k_1 = 8$	O	-	-	-	-	-	-	-	I	-	-	-
$k_2 = 5$	O	-	-	-	-	I	-	-	O	-	-	-
$k_3 = 4$	O	-	-	-	I	O	-	-	O	I	-	-
$k_4 = 3$	O	-	-	I	O	O	-	I	I	O	-	I

Таблица 1. Пример табеле формиране при решавању проблема ранца. Улаз су $n = 4$ предмета величина $8, 5, 4$ и 3, а величина ранца је $K = 11$. Симбол "I" значи да постоји решење које обухвата предмет из одговарајуће врсте; "O": постоји решење, али само без тог предмета; "-": не постоји решење.

Метод коршћен за решавање овог проблема је специјални случај опште технике која се зове **динамичко програмирање**. Суштина динамичког програмирања је формирање великих табела (у општем случају вишедимензионалних) са свим претходним резултатима. Табеле се конструишу итеративно. Сваки елеменат се израчунава на основу већ израчунатих елемената. Основни проблем је организовати израчунавање елемената табеле на најефикаснији начин. Други пример динамичког програмирања биће приказан у одељку 5.7.

Сложеност. У табели има nK елемената. Сваки од њих израчунава се за константно време на основу друга два елемента, па је укупна временска сложеност алгоритма $O(nK)$. Ако предмети нису превелики, онда K не може бити превелико, па је $nK \ll 2^n$. Ако је K јако велико, или су величине предмета реални бројеви, онда је овај приступ непримењљив; на овај проблем вратићемо се у поглављу 11). Ако је потребно само установити да ли решење постоји, онда је одговор садржан у елементу $P[n, K]$. Ако пак треба одредити подскуп са збиром K, онда треба прећи пут уназад, полазећи од позиције (n, K), користећи поље *Postoji* елемената матрице из програма. Предмет тежине k_n припада скупу акко је $P[n, K].Pripada$ тачно; ако је $P[n, K].Pripada$ тачно, односно нетачно, онда се даље на исти начин посматра елеменат $P[n-1, K-k_n]$, односно $P[n-1, K]$ из $n-1$-е врсте, итд. Подскуп се реконструише за $O(n)$ корака.

Просторна сложеност овог алгоритма је $O(nK)$. Ако се захтева само одговор на питање да ли постоји подскуп са збиром K, лако је модификовати алгоритам тако да му просторна сложеност буде $O(K)$: наредна врста се израчунава на основу претходне, па је за израчунавање елемента $P[n, K]$ довољно имати простор за смештање две узастопне врсте матрице. Даље усавршавање доводи до алгоритма просторне сложености $O(K)$ који омогућује и ефективно проналажење подскупа са збиром K, видети задатак 4.15.

Коментар. Динамичко програмирање је ефикасно кад се проблем може свести на неколико мањих, али не сасвим малих потпроблема. Решавају се сви могући потпроблеми. Резултати израчунавања чувају се у одговарајућој табели. Дакле, динамичко програмирање је практично изводљиво само ако број потпроблема није превелики. Чак и тада динамичко програмирање ради са великим табелама, па обично захтева велики меморијски простор. У неким случајевима, као у алгоритму *Ranac* на слици 11, могуће је решити проблем користећи мањи простор, тако што се у меморији у једном тренутку чува само мањи део (две врсте) матрице. Временска сложеност је обично бар квадратна.

4.11. Уобичајене грешке

Осврнућемо се на неке могуће грешке приликом индуктивне конструкције алгоритама. Грешке приликом извођења доказа индукцијом већ су размотрене у одељку 1.11. Све те грешке овде имају своје аналогоне. На пример, често се заборавља на базу индукције. У случају рекурзивне процедуре базни случај је од суштинског значаја, јер се кроз њега излази из рекурзије. Друга честа грешка је проширивање решења за n на решење специјалног, уместо произвољног случаја проблема за $n+1$.

Ненамерна промена индуктивне хипотезе је такође честа грешка. Демонстрираћемо то на једном примеру. Граф G је **бипартитни** ако се његов скуп чворова може представити као дисјунктна унија два подскупа, тако да не постоји грана између два произвољна чвора из истог поскупа. За повезан бипартитни граф је овакво разлагање у дисјунктну унију једнозначно, видети задатак 6.22.

Проблем. За задати повезани неусмерени граф G установити да ли је бипартитни, а ако јесте, пронаћи одговарајуће разлагање његових чворова у дисјунктну унију.

Погрешно решење: Уклањамо чвор v и разлажемо остатак графа индукцијом, ако је могуће. Први подскуп зваћемо *бели*, а други *црни*. Ако је v повезан само са белим чворовима, прикључујемо га црном скупу, и обрнуто — ако је v повезан само са црним чворовима, прикључујемо га белом скупу. Ако је v повезан са чворовима из оба подскупа, онда граф није бипарититан, јер је разлагање једнозначно.

Основна грешка у овом решењу (а то је грешка коју смо намеравали да илуструјемо) је у томе што после уклањања чвора v граф не мора да остане повезан. Према томе, мањи пример проблема није исти као полазни, па се индукција не може користити. Ако бисмо уклонили чвор који не "разбија" граф, решење би било коректно.

4.12. Резиме

У овом поглављу приказано је више техника за конструкцију алгоритама, које су уствари варијанте једног истог приступа. Нове технике и бројни примери биће дати у наредним поглављима. Најбољи начин да се те технике савладају је да се оне користе за решавање проблема. Приликом решавања проблема корисно је имати на уму следеће сугестије:

- Принцип индукције користи се да се улаз за проблем сведе на један или више мањих. Ако се свођење може увек извести, а базни случај се може решити, онда је алгоритам дефинисан индукцијом, односно добијен је рекурзивни алгоритам. Основна идеја је да се пажња сконцентрише на смањивање проблема, а не на његово директно решавање.
- Један од најлакших начина да се смањи проблем је елиминација неких његових елемената, што се може покушати на више начина. Поред елиминације елемената који су очигледно сувишни (као у одељку 4.3), могуће је спајање два елемента у један, проналажење елемената који се могу обрадити на посебан начин, или увођење новог елемента који преузима улогу два или више полазних елемената (одељак 5.5).
- Величина проблема може се смањити на више начина. Међутим, не добија се сваком од тих редукција алгоритам исте ефикасности. Зато је корисно размотрити различите редоследе примене индуктивне хипотезе.
- Један од најефикаснијих начина за смањивање величине проблема је његова подела на два или више подједнаких делова. Декомпозиција је ефикасна ако се проблем може поделити, тако да се од решења потпроблема лако добија решење целог проблема. Алгоритми засновани на декомпозицији разматрани су у одељцима 5.3, 5.4, 7.2, 7.4, 8.4, 8.5.
- Приликом свођења на мање потпроблеме, треба тежити да потпроблеми буду што независнији. На пример, проблем сортирања може се

свести на проналажење и уклањање најмањег елемента; редослед осталих не зависи од уклоњеног елемента (видети одељке 5.3,6.5).

- На крају, све набројане технике треба користити заједно, комбинујући их на разне начине. На пример, може се користити декомпозиција са појачавањем индуктивне хипотезе, тако да се добијени потпроблеми лакше комбинују (одељак 7.4).

Задаци

4.1. Конструисати алгоритам за израчунавање вредности полинома заснован на декомпозицији. Колики је број сабирања и множења потребан?

4.2. Размотрити алгоритам *Bijekcija* са слике 4. Да ли је могуће да скуп S постане празан на крају извршавања алгоритма? Навести такав пример, или доказати да се то не може десити.

4.3. Шта је инваријанта петље у првој **while** петљи алгоритма *Zvezda* (слика 5)?

4.4. Дато је бинарно стабло T. Конструисати ефикасан алгоритам који даје одговор на питање да ли је T АВЛ стабло.

4.5. Унутрашњи чвор v бинарног стабла је С-АВЛ чвор ако лево и десно подстабло имају једнаке висине. Конструисати алгоритам линеарне сложености који означава све С-АВЛ чворове датог бинарног стабла, чији ни један потомак није С-АВЛ чвор.

4.6. За бинарно стабло кажемо да је Л-АВЛ стабло ако за сваки његов унутрашњи чвор лево подстабло има висину за један већу од десног. Конструисати алгоритам линеарне сложености за проверу да ли је дато стабло Л-АВЛ стабло.

4.7. Конструисати алгоритам који проверава да ли су једнака два задата бинарна стабла.

4.8. Прецизирати решење проблема ранца коришћењем поља *Pripada* (слика 11).

4.9. Нека су дате две функције, $f'(x)$ и $f''(x)$,

$$
\begin{aligned}
f'(x) &= f_{(x'_1,h'_1),(x'_2,h'_2),\ldots,(x'_k,h'_k)}(x), \\
f''(x) &= f_{(x''_1,h''_1),(x''_2,h''_2),\ldots,(x''_l,h''_l)}(x),
\end{aligned}
$$

максимуми две групе од по $n/2$ правоугаоних функција (видети одељак 4.6); $k, l \le n$, $h'_k = h'_l = 0$. Функција $f'(x)$ (и слично $f''(x)$) има у интервалима $[x'_i, x'_{i+1})$ константну вредност h'_i, $0 \le i \le k$, при чему се због удобности узима да је $x'_0 = x''_0 = -\infty$, $h'_0 = h''_0 = 0$, $x'_{k+1} = +\infty$. Конструисати алгоритам сложености $O(n)$ за одређивање функције $f(x) = \max\{f'(x), f''(x)\}$.

4.10. Модификација проблема максимум скупа правоугаоника: Уместо правоугаоника (L, D, V) са x-координатама левог L и десног краја D, односно висином V, који "лежи" на x-оси, сада (L, D, V) представља фигуру добијену додавањем "крова" (једнакокрако-правоуглог троугла са хипотенузом дужине $D - L$ на горњој ивици правоугаоника) на правоугаоник. Задатак остаје исти — одредити максимум n оваквих функција алгоритмом сложености $O(n \log n)$.

4.11. Нека је $x_1, x_2, \ldots, x_n$ низ реалних (не обавезно позитивних) бројева. Конструисати алгоритам сложености $O(n)$ за одређивање подниза $x_i, x_{i+1}, \ldots, x_j$ са највећим могућим производом, $1 \le i \le j \le n$. Производ празног подниза је по дефиницији једнак 1.

4.12. За чвор бинарног стабла (које у општем случају не мора да буде АВЛ стабло) каземо да је АВЛ чвор ако је његов фактор равнотеже из скупа $\{0, \pm 1\}$. Конструисати алгоритам који у датом бинарном стаблу T означава све критичне чворове (чворове који нису АВЛ чворови, а чији су сви потомци АВЛ чворови).

4.13. Нека је $G = (V, E)$ стабло са n чворова. Циљ је формирати симетричну квадратну матрицу реда n, чији елеменат (i, j) је једнак растојању између чворова v_i и v_j. Конструисати алгоритам сложености $O(n^2)$ који решава овај проблем ако је стабло задато листом повезаности.

4.14. Растојање између два чвора стабла је број грана на јединственом путу који их повезује. Дијаметар стабла $T = (V, E)$ је највеће међу растојањима нека два његова чвора. Конструисати ефикасан алгоритам за одређивање дијаметра датог стабла, и одредити његову временску сложеност.

4.15. Како се може побољшати коришћење меморијског простора у алгоритму *Ranac* са слике 11? Колика је просторна сложеност усавршеног алгоритма?

4.16. Варијанта проблема ранца. Дати су бројеви $x_1, x_2, \ldots, x_n$ и K; потребно је представити K у облику збира неких бројева x_i, $1 \le i \le n$, при чему се сабирци могу понављати произвољан број пута.

4.17. Варијанта проблема ранца. Претпоставке су исте као и у задатку 4.16 (n типова предмета тежина $x_1, x_2, \ldots, x_n$; залихе тих предмета су неограничене, и задата је сума K тежина предмета у ранцу), али је сада предметима типа i придружена вредност v_i, $i = 1, 2, \ldots, n$. Конструисати алгоритам који ранац пуни до врха, и то предметима са највећим збиром вредности; другим речима, одредити ненегативне целе бројеве $a_1, a_2, \ldots, a_n$ такве да је $\sum_{i=1}^{n} a_i k_i = K$, а да сума $\sum_{i=1}^{n} a_i v_i$ има максималну вредност.

4.18. Нека је дат скуп природних бројева $\{x_1, x_2, \ldots, x_n\}$ и нека је $S = \sum_{i=1}^{n} x_i$. Конструисати алгоритам сложености $O(nS)$ који разлаже овај скуп у два дисјунктна подскупа са једнаким сумама елемената, или утврђује да такво разлагање није могуће.

4.19. На располагању нам је алгоритам у облику *црне кутије* (што значи да не знамо његове детаље) са следећим особинама: за дати скуп од n природних бројева и природни број k, алгоритам даје одговор "да" или "не", зависно од тога да ли постоји подскуп датих бројева са сумом једнаком k. Како се може одредити подскуп бројева са сумом k (ако постоји), користећи црну кутију $O(n)$ пута?

4.20. Проблем ханојских кула је стандардни пример нетривијалног проблема који има једноставно рекурзивно решење. Укупно n дискова различите величине сложени су на усправни штапић A, тако да им величине опадају одоздо навише. Поред тога, постоје и два празна штапића B и C. Циљ је преместити све дискове, један по један, на други штапић (B или C), на следећи начин. Дискови се премештају са врха једног на врх другог штапића, али само ако диск долази на диск већи од себе (или на празан штапић). Број премештања дискова треба да буде најмањи могући.

(а) Конструисати индукцијом алгоритам за налажење најкраћег низа потеза који решава проблем ханојских кула са n дискова.

(б) Колико потеза траје извршавање алгоритма? Формирати диференцну једначину за број потеза и решити је.

(ц) Доказати да је добијени број потеза у (б) оптималан, односно да не постоји алгоритам који решава проблем у мањем броју потеза.

4.21. Размотримо следећу варијанту проблема ханојских кула (задатак 4.20). У почетном положају дискови не морају више бити на једном штапићу. Могу бити на произвољан начин распоређени на три штапића, уз услов да им величине опадају (одоздо навише) на сваком штапићу. Циљ који треба постићи остаје исти, да се сви дискови преместе на задати штапић (уз иста ограничења као и у основном проблему), користећи што је мање потеза могуће. Конструисати алгоритам за одређивање најкраћег низа потеза који решава ову варијанту проблема ханојских кула са n дискова.

Глава 5

Алгоритми за рад са низовима и скуповима

5.1. Увод

У овом поглављу бавићемо се проблемима за које су улази коначни низови или скупови. Разлика између низова и скупова је у томе што је код низова битан редослед елемената, а код скупова није. Такође, код скупова се претпоставља да се један исти елеменат појављује највише једном, а код низова такве претпоставке нема. Пошто су улазни подаци обично дати неким редом, можемо их сматрати низом. Ипак, улаз се може сматрати скупом ако нас редослед елемената не занима. Ако се другачије не нагласи, у овом поглављу се претпоставља да је представа улаза (улазних података) вектор задате величине. Најчешће се претпоставља да су то елементи потпуно уређеног скупа (нпр. цели или реални бројеви), тако да се могу упоређивати. Разматрају се питања као што су максималност, редослед, специјални поднизови, компресија података и упоређивање низова.

Разматра се више алгоритама са широком применом. Наводе се нови примери методологије конструкције алгоритама уведене у поглављу 4, али и неки важни алгоритми. То су важни и универзално примењиви алгоритми (на пример бинарна претрага и сортирање), важни алгоритми са применом у специфичним областима (компресија података, упоређивање низова), као и не много важни алгоритми који илуструју важне технике (налажење два највећа броја у скупу, проблем муцавог подниза).

Први пример у овом поглављу је бинарна претрага — фундаментални и елегантан алгоритам који се појављује у много облика и на кога се наилази у многим ситуацијама. Затим се прелази на сортирање — један од најшире проучаваних проблема, ранговске статистике, компресију података, два проблема повезана са обрадом текста и пробабилистичке алгоритме.

5.2. Бинарна претрага и варијације

Бинарна претрага је за област алгоритама оно што је точак за механизме: она је једноставна, елегантна, неизмерно важна, и откривана је више пута. Основна идеја бинарне претраге је подела простора на два приближно једнака дела постављањем само једног питања. У овом одељку разматра се неколико варијација бинарне претраге.

5.2.1. Чиста бинарна претрага.

Проблем. Нека је $x_1, x_2, \ldots, x_n$ низ реалних бројева такав да је $x_1 \leq x_2 \leq \cdots \leq x_n$. За задати реални број z треба установити да ли се z појављује у низу, а ако је одговор "да", потребно је пронаћи индекс i такав да је $x_i = z$.

Због једноставности, тражимо само један индекс i такав да је $z = x_i$. У општем случају циљ може да буде проналажење свих таквих индекса, најмањег или највећег међу њима и слично. Идеја је преполовити простор који се претражује тако што се најпре провери средњи члан низа. Претпоставимо због једноставности да је n паран број. Ако је z мање од $x_{n/2+1}$, онда z може бити само у првој половини низа; у противном, z може бити само у другој половини низа. Проналажење z у првој или другој половини низа је проблем величине $n/2$, који се решава рекурзивно. Базни случај $n = 1$ решава се непосредним упоређивањем z са елементом. Алгоритам је дат на слици 1.

```
Алгоритам Binarna_Pretraga(X, n, z);
Улаз: X (низ од n бројева, уређених неопадајуће) и z (број који се тражи).
Излаз: Poz (индекс i такав да је X[i] = z, или 0 ако такав индекс не постоји).
begin
    Poz := Nadji(z, 1, n);
end

function Nadji(z, Levi, Desni) : integer
begin
    if Levi = Desni then
        if X[Levi] = z then Nadji := Levi
        else Nadji := 0
    else
        Srednji = ⌈(Levi + Desni)/2⌉;
        if z < X[Srednji] then
            Nadji = Nadji(z, Levi, Srednji − 1)
        else
            Nadji = Nadji(z, Srednji, Desni)
end
```

Рис. 1. Бинарна претрага.

Сложеност. После сваког упоређивања опсег могућих индекса се полови, па је потребан број упоређивања за проналажење задатог броја у низу величине n $O(\log n)$. Ова варијанта бинарне претраге одлаже проверу једнакости до самог краја. Алтернатива је провера једнакости са z у сваком кораку. Проблем са приказаном варијантом је у томе што се претрага не може завршити

пре него што се опсег за претрагу сузи на само један број; предност јој је пак да се у сваком кораку врши само једно упоређивање. Оваква претрага је због тога обично бржа. Иако је једноставније направити рекурзивни програм, лако је овај програм превести у нерекурзивни. Бинарна претрага није тако ефикасна за мале вредности n, па је тада боље задати низ претражити линеарно, члан по члан.

Следећи проблем своди се на бинарну претрагу.

Проблем. За задати растуће уређени низ целих бројева $a_1, a_2, \ldots, a_n$ утврдити да ли постоји индекс i, такав да је $a_i = i$.

Заиста, низ $x_i = a_i - i$ је неопадајући (јер је $x_{i+1} - x_i = a_{i+1} - a_i - 1 \geq 0$), а услов $a_i = i$ је еквивалентан услову $x_i = 0$. Према томе, задати проблем решава се бинарном претрагом низа x_i, у коме се тражи број $z = 0$.

5.2.2. Бинарна претрага циклички уређеног низа. За низ $x_1, x_2, \ldots, x_n$ каже се да је **циклички уређен** ако важе неједнакости

$$x_i < x_{i+1} < \cdots < x_n < x_1 < \cdots < x_{i-1},$$

где је x_i најмањи елеменат низа.

Проблем. За задати циклички уређени низ пронаћи позицију минималног елемента низа (због једноставности може се претпоставити да је та позиција јединствена).

Да бисмо пронашли минимални елеменат x_i у низу, користимо идеју бинарне претраге да једним упоређивањем елиминишено половину низа. Узмимо произвољна два броја x_k и x_m таква да је $k < m$. Ако је $x_k < x_m$, онда i не може бити у интервалу $k < i \leq m$, јер је x_i најмањи елеменат низа (у том случају било би $x_i < x_m < x_k$ — супротно претпоставци да је $x_k < x_m$). У противном, ако је $x_k > x_m$, онда i *мора* бити у интервалу $k < i \leq m$, јер је тада монотоност низа прекинута негде у интервалу индекса $[k, m]$, видети слику 2. Према томе, једним упоређивањем може се елиминисати много елемената. Одговарајућим избором k и m, i се може одредити помоћу $O(\log n)$ упоређивања, видети алгоритам на слици 3. Инваријанта главне петље алгоритма је услов да се i увек налази у интервалу $[Levi, Desni]$.

5.2.3. Бинарна претрага низа непознате дужине. Посматрајмо варијанту проблема претраге кад се задати број z тражи у уређеном низу непознате дужине. Да би се проблем свео на већ решен, потребно је пронаћи бар један индекс i такав да је $x_i > z$: тада се може прећи на бинарну претрагу опсега индекса од 1 до i.

Може се поступити на следећи начин. Најпре се z упоређује са x_1. Ако је $z \leq x_1$, онда z може бити једнако само броју x_1. Претпоставимо да знамо индекс j такав да је $z > x_j$. После упоређивања z са x_{2j} постоје две могућности. Ако је $z \leq x_{2j}$, онда знамо да је $x_j < z \leq x_{2j}$, па се z може пронаћи помоћу

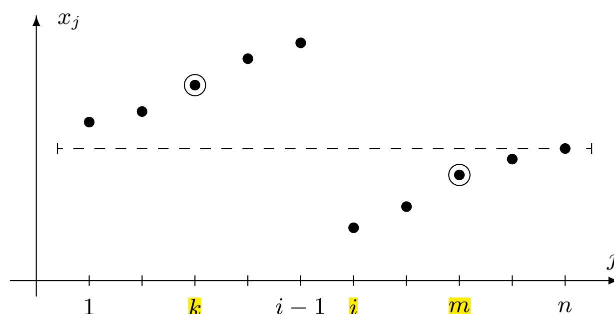


Рис. 2. Циклички уређен низ — илустрација.

Алгоритам Cikl_Binarna_Pretraga(X, n);
Улаз: X (вектор са циклички уређеним низом од n различитих бројева).
Излаз: poz (индекс минималног елемента у X).

```
begin
    poz := c_Nadji(1, n);
end

function c_Nadji(Levi, Desni) : integer
begin
    if Levi = Desni then
        c_Nadji := Levi
    else
        Srednji = ⌊(Levi + Desni)/2⌋;
        if X[Srednji] < X[Desni] then
            c_Nadji = c_Nadji(Levi, Srednji)
        else
            c_Nadji = c_Nadji(Srednji + 1, Desni)
end
```

Рис. 3. Бинарна претрага циклички уређеног низа.

$O(\log_2 j)$ упоређивања. Ако је пак $z > x_{2j}$, онда је простор за претраживање удвостручен, и треба наставити (индукцијом) тако што се j замени са $2j$. Претпоставимо да је i најмањи индекс такав да је $z \leq x_i$. Тада је довољно $O(\log_2 i)$ упоређивања да се (удвостручавањем) пронађе такво x_j које је веће или једнако од z, и нових $O(\log_2 i)$ упоређивања да се пронађе z.

Исти алгоритам може употребити и ако је дужина низа позната, али очекујемо да је индекс i врло мали. Да би овај алгоритам био бољи од обичне бинарне претраге, потребно је да буде приближно $2\log_2 i < \log_2 n$, односно $i < \sqrt{n}$.

5.2.4. Проблем муцавог подниза. Принцип бинарне претраге може се искористити и у проблемима који на први поглед немају везе са бинарном претрагом. Нека су $A = a_1a_2 \dots a_n$ и $B = b_1b_2 \dots b_m$ два низа знакова из коначног алфабета и $m \leq n$. Каже се да је B подниз A ако постоје индекси $i_1 < i_2 < \cdots < i_m$ такви да је $b_j = a_{i_j}$ за све индексе j, $1 \leq j \leq m$, односно ако се низ B може "уклопити" у низ A. Лако је установити да ли је B подниз низа A: у низу A тражи се прва појава знака b_1, затим од те позиције даље прва појава знака b_2, итд. Исправност овог алгоритма лако се доказује индукцијом. Пошто се елементи оба низа пролазе тачно по једном, временска сложеност алгоритма је $O(m + n)$. За задати низ B, нека B^i означава низ у коме се редом сваки знак низа B понавља i пута. На пример, ако је $B = xyzzx$, онда је $B^3 = xxxyyyzzzzzzxxx$.

Проблем. Дата су два низа A и B. Одредити максималну вредност i такву да је B^i подниз низа A.

Овај проблем може се назвати **проблем муцавог подниза**. Иако на први поглед тежак, он се лако може решити помоћу бинарне претраге.

За сваку задату вредност i лако се конструише низ B^i, и проверава да ли је B^i подниз низа A. Поред тога, ако је B^j подниз A, онда је за свако i, $1 \leq i \leq j$, B^i подниз A. Највећа вредност за i коју треба разматрати мања је или једнака од n/m, јер би у противном низ B^i био дужи од низа A. Дакле, може се применити бинарна претрага. Почињемо са $i = \lceil n/(2m) \rceil$ и проверавамо да ли је B^i подниз A. Затим настављамо са бинарном претрагом, елиминишући доњи, односно горњи подинтервал ако је одговор да, односно не. После $\lceil \log_2(n/m) \rceil$ тестова биће пронађено максимално i. Временска сложеност алгоритма је $O((n+m)\log(n/m)) = O(n \log(n/m))$. Проблеми са упоређивањем низова разматрају се и у одељку 5.7.

Решење овог проблема лако је уопштити. Претпоставимо да ако број i задовољава услов $P(i)$, онда и сви бројеви j, $1 \leq j \leq i$ задовољавају тај услов. Ако тражимо максималну вредност i која задовољава услов $P(i)$, довољно је да имамо алгоритам који утврђује да ли *задато* i задовољава тај услов. Тада се проблем решава бинарном претрагом могућих вредности за i. Ако се не зна горња граница за i, може се искористити поступак са удвостручавањем: почиње се са $i = 1$, а затим се i удвостручава, све док се не добије нека горња граница. Резултујући алгоритам, иако ефикасан, често није оптималан, и фактор $\log n$ из израза за временску сложеност може се елиминисати (као у случају проблема муцавог подниза).

5.2.5. Нумеричко решавање једначина. Поступак бинарне претраге сличан је нумеричком решавању једначине $f(x) = 0$ на интервалу $[a, b]$ методом бисекције, при чему је нпр. $f(a)f(b) < 0$ и функција $f(x)$ је непрекидна. Циљ је наћи решење једначине са задатом тачношћу.

Пошто је функција непрекидна, у интервалу $[a, b]$ једначина има бар једно решење. Израчунава се вредност функције у тачки $x_1 = (a+b)/2$. Ако је $f(x_1) = 0$ (у границама задате тачности), онда је решење пронађено. У противном,

функција у крајевима једног од подинтервала $[a, x_1]$ или $[x_1, b]$ има вредности супротног знака, па у том подинтервалу постоји бар једно решење једначине. После k корака долази се до интервала дужине $(b-a)/2^k$ који садржи решење. Са половивљењем интервала наставља се до постизања захтеване тачности.

5.2.6. Интерполациона претрага. У оквиру бинарне претраге простор претраге се сваки пут полови, што гарантује логаритамску временску сложеност. Међутим, ако се у току претраге наиђе на вредност врло блиску траженом броју z, изгледа разумно да се претрага усредсреди на "околину" тог индекса уместо да се настави са половљењем наслепо. Специјално, ако је вредност z врло мала, може се започети са претрагом негде на почетку низа уместо од средњег индекса.

Посматрајмо начин на који отварамо књигу кад тражимо неку страницу. Нека, на пример, у књизи од 800 страна тражимо страну 200. Она се налази око прве четвртине књиге, па ми ту чињеницу користимо кад отварамо књигу. Вероватно је да нећемо одмах погодити страну 200; претпоставимо да смо књигу отворили на страни 250. Сада је претрага сужена на опсег од 250 страна, а тражена страна је на око 4/5 опсега. Ми покушавамо да отворимо књигу на око петине пре краја опсега. Процес настављамо док се не приближимо довољно страни 200, после чега до ње долазимо окрећући страну по страну. Описани поступак је управо суштина интерполационе претраге. Уместо да половимо простор претраге, ми га делимо тако да буде највећа "вероватноћа" успеха. Однос у коме се опсег индекса дели одређује се интерполацијом, што је илустровано на слици 4. Првом интерполацијом добија се индекс 6; испоставља да је $X[6] < z$. Следећа интерполација у интервалу индекса $[6, 17]$ доводи до решења $X[7]$. Алгоритам, заједно са изразом по коме се врши интерполација, дат је на слици 5.

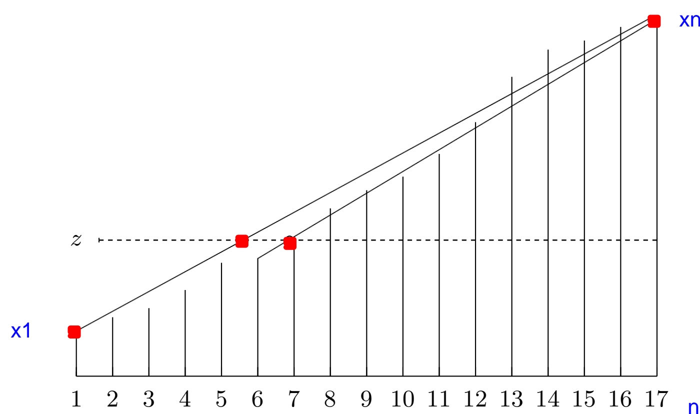


Рис. 4. Интерполациона претрага — илустрација.

Алгоритам Interpolaciona_Pretraga(X, n, z);
Улаз: X (низ од n неопадајуће уређених бројева) и z (број који се тражи).
Излаз: Poz (индекс i такав да је $X[i] = z$, или 0 ако такав индекс не постоји).
begin
  if $z < X[1]$ **or** $z > X[n]$ **then** $Poz := 0$ {неуспешна претрага}
  else $Poz := i_Nadji(z, 1, n)$;
end

function i_Nadji$(z, Levi, Desni)$: *integer*
begin
  if $X[Levi] = z$ **then** $i_Nadji := Levi$
  else if $Levi = Desni$ **or** $X[Levi] = X[Desni]$ **then**
    $i_Nadji := 0$
  else
    $Srednji := \left\lceil Levi + \frac{(z - X[Levi])(Desni - Levi)}{X[Desni] - X[Levi]} \right\rceil$;
    if $z < X[Srednji]$ **then**
      $i_Nadji = i_Nadji(z, Levi, Srednji - 1)$
    else
      $i_Nadji = i_Nadji(z, Srednji, Desni)$
end

Рис. 5. Интерполациона претрага.

Сложеност. Ефикасност интерполационе претраге не зависи само од дужине низа, него и од свих његових елемената, односно од комплетног улаза. Постоје улази за које интерполациона претрага проверава практично сваки члан низа (видети задатак 5.5). С друге стране, интерполациона претрага је врло ефикасна за улазе са приближно униформно расподељеним елементима (односно ако су тачке (i, x_i) груписане у околини праве; пример са странама у књизи очигледно спада у ову категорију). Показује се да је средњи број упоређивања у току интерполационе претраге (усредњавање по свим могућим низовима) једнак $O(\log \log n)$. Ипак, интерполациона претрага није много боља од обичне бинарне претраге из два разлога. Најпре, сем за јако велике вредности n, већ је $\log_2 n$ мала вредност, па њен логаритам није много мањи. Друго, код избора наредног индекса у току интерполационе претраге потребно је извршити нешто компликованије аритметичке операције.

5.3. Сортирање

Сортирање је један од најшире проучаваних проблема у рачунарству. Сортирање је основа за многе алгоритме и троши велики део рачунарског времена

у многим типичним апликацијама. Постоје многе верзије проблема сортирања, као и десетине алгоритама за сортирање. Овде ћемо размотрити само неколико познатијих алгоритама.

Проблем. Датих n бројева $x_1, x_2, \dots, x_n$ треба уредити у неопадајући низ. Другим речима, треба пронаћи низ различитих индекса $1 \le i_1, i_2, \dots i_n \le n$ таквих да је $x_{i_1} \le x_{i_2} \le \cdots \le x_{i_n}$.

Због једноставности, ако се не нагласи другачије, претпостављамо да су бројеви x_i различити. Сви описани алгоритми раде исправно и са бројевима који нису различити. За алгоритам сортирања каже се да је **у месту** ако се не користи допунски меморијски простор (осим оног у коме је смештен улазни низ).

5.3.1. Сортирање разврставањем и сортирање вишеструким разврставањем. Најједноставнији поступак сортирања састоји се у томе да се обезбеди довољан број локација, и да се онда сваки елеменат смести на своју локацију. Тај поступак зове се сортирање разврставањем. Ако се, на пример, сортирају писма према одредиштима, онда је довољно обезбедити једну преграду за свако одредиште, и сортирање је врло ефикасно. Али ако писма треба сортирати према петоцифреном поштанском броју, онда овај метод захтева око 100000 преграда, што поступак чини непрактичним. Према томе, сортирање разврставањем ради добро ако су елементи из малог, једноставног опсега, који је унапред познат. Прелазимо на детаљнији опис овог алгоритма.

Нека је дато n елемената, који су цели бројеви из опсега од 1 до $m \ge n$. Резервише се m локација, а онда се за свако i број x_i ставља на локацију x_i која одговара његовој вредности. После тога се прегледају све локације и из њих се редом покупе елементи. Сложеност овог једноставног алгоритма је дакле $O(m+n)$. Ако је $m = O(n)$, добијамо алгоритам за сортирање линерне сложености. С друге стране, ако је m велико у односу на n (као у случају поштанских бројева), онда је и $O(m)$ такође велико. Поред тога, алгоритам захтева меморију величине $O(m)$, што је још већи проблем за велике m.

Природно уопштење ове идеје је сортирање вишеструким разврставањем. Размотримо још једном пример са поштанским бројевима. Сортирање разврставањем у овом случају није погодно, јер је превелики опсег могућих поштанских бројева. Како смањити потребан опсег? Применићемо индукцију по опсегу на следећи начин. Најпре користимо 10 преграда и сортирамо писма према првој цифри поштанског броја. Свака преграда сада покрива 10000 различитих поштанских бројева (одређених са преостале четири цифре поштанског броја). Број операција за ову етапу је $O(n)$. На крају прве етапе имамо 10 преграда, од којих свака одговара мањем опсегу. Даље се проблем за сваку преграду решава индукцијом. Пошто се опсег после сваке етапе смањује за фактор 10, и пошто поштански бројеви имају пет цифара, довољно је пет етапа. Кад се садржаји преграда сортирају, лако их је објединити у сортирану (уређену) листу.

Размотримо сада другу варијанту исте идеје. Напоменимо да се опсег може поделити на било који одговарајући начин. У примеру са поштанским бројевима подела је извршена у складу са декадним приказом поштанских бројева. Ако су елементи стрингови које треба сортирати лексикографски, можемо их упоређивати знак по знак, па се добија **лексикографско сортирање**. Размотрена верзија сортирања вишеструким разврставањем (цифре се пролазе слева удесно) позната је као **сортирање обратним вишеструким разврставањем**.

Рекурзивна реализација сортирања обратним вишеструким разврставањем захтева помоћне локације (око 50 преграда у примеру са поштанским бројевима; сваки ниво рекурзије има своје преграде). Други начин реализације сортирања вишеструким разврставањема заснива се на примени индукције обрнутим редоследом: сортирање се ради здесна улево, полазећи од најнижих уместо од највиших цифара. Претпостављамо да су елементи велики бројеви, представљени са k цифара у систему са основом d (цифре су из опсега од 0 до $d-1$). Индуктивна хипотеза је врло природна.

Индуктивна хипотеза. Умемо да сортирамо елементе са мање од k цифара.

Разлика између овог метода и сортирања обратним вишеструким разврставањем је у начину на који се проширује хипотеза (идеја примене индуктивне хипотезе обрнутим редоследом слична је као код Хорнерове шеме, одељак 4.2). Код датих бројева са k цифара ми најпре игноришемо *највишу* (прву) цифру и сортирамо бројеве према остатку цифара индукцијом. Тако добијамо листу елемената сортираних према најнижих $k-1$ цифара. Затим пролазимо још једном кроз све елементе, и разврставамо их према највишој цифри у d преграда. Коначно, обједињујемо редом садржаје преграда. Овај алгоритам зове се сортирање директним вишеструким разврставањем. Показаћемо да су елементи на крају сортирани по свих k цифара.

Тврдимо да су два елемента сврстана у различите преграде у исправном поретку. За то нам није потребна индуктивна хипотеза, јер је највиша цифра првог од њих већа од највише цифре другог. С друге стране, ако два елемента имају исте највише цифре, онда су они према индуктивној хипотези доведени у исправан редослед пре последњег корака. Битно је да елементи стављени у исту преграду остају у истом редоследу као и пре разврставања. Ово се може постићи употребом листе за сваку преграду и обједињавањем d листа на крају сваке етапе у једну глобалну листу од свих елемената (сортираних према i најнижих цифара). Алгоритам је приказан на слици 6.

Сложеност. Потребно је n корака за копирање елемената у глобалну листу GL, и d корака за иницијализацију листа $Q[i]$. У главној петљи алгоритма, која се извршава k пута, сваки елеменат се вади из глобалне и ставља у неку од листа $Q[i]$. На крају се све листе $Q[i]$ поново обједињавају у GL. Укупна временска сложеност алгоритма је $O(nk)$.

Алгоритам DVR_sort(X, n, k);
Улаз: X (низ од n целих ненегативних бројева са по k цифара).
Излаз: X (сортирани низ).
begin
  Претпостављамо да су на почетку сви елементи у глобалној листи GL
  $\{GL$ се користи због једноставности; листа се може реализовати у $X\}$
  for $i := 0$ **to** $d-1$ **do**
  $\{d$ је број могућих цифара; $d = 10$ у декадном случају$\}$
    иницијализовати листу $Q[i]$ као празну листу;
  for $i := k$ **downto** 1 **do**
    while GL није празна **do** $\{$разврставање по i-тој цифри$\}$
      скини x из GL; $\{x$ је први елеменат са листе$\}$
      $c := i$–та цифра x; $\{$гледано слева удесно$\}$
      убаци x у $Q[c]$;
    for $t := 0$ **to** $d-1$ **do** $\{$обједињавање локалних листа$\}$
      укључи $Q[t]$ у GL; $\{$додавање на крај листе$\}$
  for $i := 1$ **to** n **do** $\{$преписивање елемената из GL у низ $x\}$
    скини $X[i]$ из GL
end

Рис. 6. Сортирање директним вишеструким разврставањем.

У наставку ћемо разматрати технике сортирања које користе упоређивање елемената, не узимајући у обзир њихову структуру. Ради се дакле о општијим алгоритмима, који за елементе претпостављају само да се могу упоређивати.

5.3.2. Сортирање уметањем и сортирање избором. Користићемо директну индукцију. Претпоставимо да умемо да сортирамо $n-1$ елемената и да је дато n елемената. После сортирања првих $n-1$ елемената, n-ти елеменат се умеће на место које му припада, тако што се прегледа $n-1$ сортираних бројева, до проналажења правог места за уметање. Овакав поступак зове се **сортирање уметањем**. Он је једноставан и ефикасан само за мале вредности n. У најгорем случају n-ти број упоређује се са свих претходних $n-1$ бројева. Укупан број упоређивања при сортирању n елемената може да буде дакле $1 + 2 + \cdots + n = n(n-1)/2 = O(n^2)$. Поред тога, уметање n-тог броја укључује померање осталих бројева. У најгорем случају у n-том кораку премешта се $n-1$ бројева, па је и број премештања (копирања) $O(n^2)$. Сортирање уметањем може се побољшати смештањем елемената у вектор и коришћењем бинарне претраге $n-1$ сортираних бројева да би се пронашло место за уметање. Претрага се састоји од $O(\log n)$ упоређивања по уметању, па је укупан број упоређивања тада $O(n \log n)$. Међутим, број премештања података остаје непромењен, па је сложеност алгоритма и даље квадратна.

Индуктивни приступ може се побољшати избором посебног n-тог броја. Највећи број је добар избор, јер се зна где му је место — на крају низа. Алгоритам се састоји од избора највећег броја и његовог постављања на право место заменом са бројем који је тренутно на том месту; после тога се рекурзивно сортира остатак бројева. Овај алгоритам зове се сортирање избором. Његова предност над сортирањем уметањем је у томе што користи само $n-1$ замена бројева, у односу на $O(n^2)$ копирања у најгорем случају код сортирања уметањем. С друге стране, пошто је потребно $n-1$ упоређивање да се пронађе највећи елеменат (налажење највећег броја разматра се у одељку 5.4.1), укупан број упоређивања је увек $O(n^2)$, док је код сортирања уметањем са бинарном претрагом потребно само $O(n \log n)$ упоређивања.

За ефикасно уметање и избор могу се користити уравнотежена стабла (видети поглавље 3). Ако се, на пример, искористе АВЛ стабла, свако уметање захтева $O(\log n)$ корака. Ишчитавање свих бројева из АВЛ стабла у сортираном редоследу може се извести у $O(n)$ корака рекурзивном процедуром која ишчитава све елементе левог подстабла, па корен, па све елементе десног подстабла. Ако умемо да формирамо АВЛ стабло за $n-1$ бројева, онда је потребно само уметнути n-ти број, односно $O(\log n)$ корака. Укупно је дакле потребно $O(n \log n)$ корака за уметање n бројева у празно АВЛ стабло и $O(n)$ корака за њихово ишчитавање у сортираном редоследу. За велике n ово је много ефикасније и од сортирања уметањем и од сортирања избором, али захтева више простора за смештање показивача. Овај алгоритам сортирања није у месту, компликован је, и није тако добар као алгоритми који ћемо размотрити.

5.3.3. Сортирање обједињавањем. Ефикасност сортирања уметањем може се побољшати на следећи начин. За време потребно за прегледање већ сортираних бројева и проналажење места за уметање једног броја, могу се наћи места за више бројева. Ова идеја је већ коришћена у одељку 4.6. Два сортирана низа могу се објединити у сортирани низ једним пролазом. Обједињавање се врши тако што се бројеви из другог низа разматрају редом: тражи се место у првом низу за најмањи број, па за следећи итд. Прецизније, означимо низове са $a_1, a_2, \ldots, a_n$, односно $b_1, b_2, \ldots, b_m$, и претпоставимо да су оба низа сортирана у растућем поретку. Елементи првог низа се прегледају док се не пронађе место за b_1, па се умеће b_1. Затим се наставља са прегледањем од те позиције даље, док се не пронађе место за b_2, итд. Пошто су бројеви у другом низу сортирани, нема потребе за враћањем у првом низу. Укупан број упоређивања у најгорем случају једнак је збиру дужина два низа. Проблем су премештања: неефикасно је премештати елементе при сваком уметању, јер се тада елементи морају премештати више пута. Уместо тога, пошто обједињавање даје један по један елеменат у сортираном редоследу, елементе копирамо у привремени низ; сваки елеменат се копира тачно једном. Укупно је за обједињавање два сортирана низа дужина n и m потребно $O(n+m)$ упоређивања и премештања података. Претпоставља се да је расположив меморијски простор за помоћни низ.

Описани поступак обједињавања може се искористити као основа за алгоритам сортирања заснован на декомпозицији, познат као сортирање обједињавањем. Алгоритам ради на следећи начин. Најпре се низ подели на два једнака или приближно једнака дела (ако му је дужина непарна). Затим се оба дела рекурзивно сортирају. На крају се сортирани делови обједињавају у један сортирани низ на претходно описани начин. Детаљније је алгоритам приказан на слици 7, а пример његовог извршавања у табели 1 (копирања нису приказана).

Сложеност. Нека је $T(n)$ број упоређивања потребних за сортирање обједињавањем у најгорем случају. Претпоставимо због једноставности да је n степен двојке. Да би се израчунало $T(n)$, треба решити следећу диференцну једначину:

$$T(2n) = 2T(n) + cn, \quad T(2) = 1.$$

Њено решење је $T(n) = O(n \log n)$ (видети поглавље 2), што је асимптотски боље од $O(n^2)$, временске сложености сортирања уметањем и сортирања избором. Број премештања података је такође $O(n \log n)$, што је више него код сортирања избором.

Иако је сортирање обједињавањем боље од сортирања уметањем за велике n, ипак има и неке озбиљне недостатке. Поред тога што се не реализује лако, корак обједињавања захтева допунску меморију, тј. ово није сортирање у месту. Копирање се ради при сваком обједињавању мањих скупова, што чини процедуру споријом.

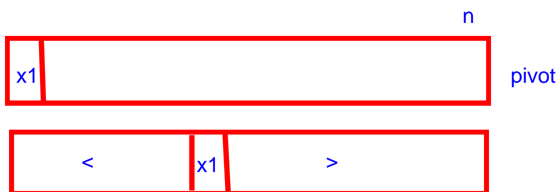
5.3.4. Сортирање раздвајањем. Сортирање обједињавањем пример је ефикасности декомпозиције. Ако нам пође за руком да поделимо проблем на

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

Таблица 1. Пример сортирања обједињавањем. Прва врста је улазни низ. У свакој врсти приказани су само бројеви са којима се тренутно ради.

```
Алгоритам Sort_Objed(X, n);
Улаз: X (низ од n бројева).
Излаз: X (сортирани низ).
begin
    M_sort(1, n)
end
procedure M_Sort(Levi, Desni);
begin
    if Desni − Levi = 1 then
     {провера која није неопходна, али чини програм ефикаснијим}
        if X[Levi] > X[Desni] then zameni (X[Levi], X[Desni])
    else if Levi ≠ Desni then
        Srednji := ⌈(Levi + Desni)/2⌉;
        M_Sort(Levi, Srednji − 1);
        M_Sort(Srednji, Desni);
         {сада се два сортирана дела низа обједињавају}
        i := Levi; {показивач на леву половину низа}
        j := Srednji; {показивач на десну половину низа}
        k := 0; {показивач на помоћни вектор Pom}
        while (i ≤ Srednji − 1) and (j ≤ Desni) do
            k := k + 1;
            if X[i] ≤ X[j] then
                Pom[k] := X[i];
                i := i + 1
            else
                Pom[k] := X[j];
                j := j + 1
        if j > Desni then
             {копирање остатка леве половине на крај вектора Pom}
             {ако је i ≥ Srednji онда је десна страна већ на свом месту}
            for t := 0 to Srednji − 1 − i do
                X[Desni − t] := X[Srednji − 1 − t];
         {копирање помоћног вектора Pom назад у X}
        for t := 0 to k − 1 do
            X[Levi + t] := Pom[t + 1]
end
```

Рис. 7. Сортирање обједињавањем.



два приближно једнака потпроблема, решимо сваки од њих независно и њихова решења комбинујемо за линеарно време $O(n)$, добијамо алгоритам сложености $O(n \log n)$. Недостатак сортирања обједињавањем је потреба за допунском меморијом, јер се приликом обједињавања не може предвидети који ће елеменат где завршити. Може ли се декомпозиција применити на други начин, тако да се положаји елемената могу предвидети? Идеја сортирања раздвајањем је да се главнина времена утроши на фазу поделе, а врло мали део на обједињавање.

Претпоставимо да знамо број x, такав да је пола елемената низа мање или једнако, а пола веће од x. Број x може се упоредити са свим елементима низа ($n-1$ упоређивање) и тиме их поделити у две групе, према резултату упоређивања. Пошто две групе имају исти број елемената, једна од њих се може сместити у прву, а друга у другу половину низа. Као што ћемо одмах видети, ово раздвајање може се извршити без допунског меморијског простора; то је фаза поделе. Затим се оба подниза рекурзивно сортирају. Фаза обједињавања је тривијална, јер оба дела заузимају своја места у низу. Према томе, није потребан допунски меморијски простор.

При описаној конструкцији претпостављено је да је вредност x позната, што обично није случај. Лако се увиђа да исти алгоритам добро ради без обзира на то који се број користи за раздвајање. Број x је тзв. **пивот**. Циљ је раздвојити елементе низа на два дела, један са бројевима мањим или једнаким, а други са бројевима већим од пивота. Раздвајање (партиција) се може се остварити помоћу следећег алгоритма. Користе се два показивача на низ, L и D. На почетку L показује на леви, а D на десни крај низа. Показивачи се "померају" један према другом. Следећа индуктивна хипотеза (тј. инваријанта петље) гарантује исправност раздвајања.

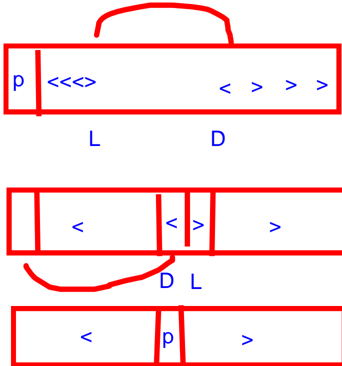
Индуктивна хипотеза. У кораку k алгоритма важи $pivot \geq x_i$ за све индексе $i < L$ и $pivot < x_j$ за све индексе $j > D$.

Хипотеза је тривијално испуњена на почетку, јер ни једно i, односно j, не задовољавају наведене услове. Циљ је у кораку $k+1$ померити L удесно, или D улево, тако да хипотеза и даље буде тачна.

У тренутку кад се деси $L = D$, раздвајање је скоро завршено, изузев евентуално елемента x_L, на шта ћемо се касније вратити. Претпоставимо да је $L < D$. Постоје две могућности. Ако је било $x_L \leq pivot$ било $x_D > pivot$, одговарајући показивач може се померити, тако да хипотеза и даље буде задовољена. У противном је $x_L > pivot$ и $x_D \leq pivot$. Тада *замењујемо* x_L са x_D и померамо оба показивача даље према унутра. У оба случаја имамо померање бар једног показивача. Према томе, показивачи ће се у једном тренутку сусрести, и циљ је постигнут.

Остаје проблем избора доброг пивота и прецизирања последњег корака алгоритма кад се показивачи сретну. Алгоритми засновани на декомпозицији најбоље раде кад делови имају приближно исте величине, што сугерише да што је пивот ближи средини, то ће алгоритам бити ефикаснији. Могуће је наћи медијану низа (о томе ће бити речи у следећем одељку), али то није вредно

труда. Као што ћемо видети из анализе алгоритма, избор случајног елемента низа је добро решење. Ако је полазни низ добро испретуран, онда се за пивот може узети први елеменат низа. Због једноставности је у алгоритму на слици 8 управо тако и урађено.



function **Razdvajanje**$(X, Levi, Desni)$;
Улаз: X (низ), $Levi$ (лева граница низа) и $Desni$ (десна граница низа).
Излаз: X и $S = Razdvajanje$, такав индекс да је $X[i] \leq X[S]$ за све $i \leq S$ и $X[j] > X[S]$ за све $j > S$.
begin
  $pivot := X[Levi]$;
  $L := Levi$; $D := Desni$;
  while $L < D$ **do**
    while $X[L] \leq pivot$ **and** $L \leq Desni$ **do** $L := L + 1$;
    while $X[D] > pivot$ **and** $D \geq Levi$ **do** $D := D - 1$;
    if $L < D$ **then**
      замени $X[L]$ и $X[D]$;
  $Razdvajanje := D$;
  замени $X[Levi]$ са $X[S]$
end

Рис. 8. Алгоритам раздвајање.

Ако се први елеменат изабере за пивот, он се може заменити са X_L у последњем кораку раздвајања, и тако поставити на своје место. Друге могућности биће размотрене приликом анализе сложености. У сваком случају, пивот који је изабран на неки други начин, може се најпре заменити са првим елементом низа, после чега се може применити описани алгоритам.

Пример рада алгоритма *Razdvajanje* дат је у табели 2. Пивот је први број (6). Бројеви који управо треба да буду замењени су подебљани После три замене и померања L показује на $X[7] = 1$, а D на $X[6] = 1$ (показивач D прескаче L). Последња је замена средњег броја ($X[D] = 1$) и пивота (6). После ове замене сви бројеви лево од пивота су мањи или једнаки од њега, а сви бројеви десно су већи од њега. Два подниза (од 1 до 6 и од 7 до 16) могу да се рекурзивно сортирају. Сортирање раздвајањем је према томе сортирање у месту. Алгоритам је приказан на слици 9, а пример његовог рада у табели 3.

Сложеност. Време извршења сортирања раздвајањем зависи од конкретног улаза и избора пивота. Ако пивот увек раздваја низ на два једнака дела, онда је диференцна једначина за сложеност $T(n) = 2T(n/2) + cn$, $T(2) = 1$, што за последицу има $T(n) = O(n \log n)$. Уверићемо се касније да је временска сложеност овог алгоритма $O(n \log n)$ и под много слабијим претпоставкама. Ако је пак пивот близу неком од крајева низа, онда је број корака много већи.

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

Таблица 2. Раздвајање низа око пивота 6.

Алгоритам Quicksort(X, n);
Улаз: X (низ од n бројева).
Излаз: X (сортирани низ).
begin
  $Q_sort(1, n)$
end
procedure Q_Sort$(Levi, Desni)$;
begin
  if $Levi < Desni$ **then**
    $S = Razdvajanje(X, Levi, Desni)$; {$S$ је индекс пивота после раздвајања}
    $Q_Sort(Levi, S - 1)$;
    $Q_Sort(S + 1, Desni)$;
end

Рис. 9. Сортирање раздвајањем.

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

Таблица 3. Пример сортирања раздвајањем. Пивоти су подебљани. Елементи који се у некој фази не мењају, нису приказани.

На пример, ако је пивот најмањи елеменат у низу, онда прво раздвајање захтева $n - 1$ упоређивања и смешта само пивот на право место. Ако је низ већ

уређен растуће и за пивот се увек бира први елеменат, онда је број корака алгоритма $O(n^2)$. Квадратна сложеност у најгорем случају за сортиране или скоро сортиране низове може се избећи упоређивањем првог, последњег и средњег елемента низа и избором средњег међу њима (другог по величини) за пивот. Још поузданији метод је *случајни* избор неког елемента низа за пивот. Временска сложеност сортирања ће и тада у најгорем случају бити $O(n^2)$, јер и даље постоји могућност да пивот буде најмањи елеменат низа. Ипак, вероватноћа да дође до овог најгорег случаја је мала. Размотрићемо то сада мало прецизније.

Претпоставимо да се сваки елеменат x_i са једнаком вероватноћом може изабрати за пивот. Ако је пивот i-ти најмањи елеменат, онда после раздвајања треба рекурзивно сортирати два дела низа дужина редом $i-1$, односно $n-i$. Ако бројимо само упоређивања, сложеност раздвајања је $n-1$ (под претпоставком да се после замене $X[L]$ са $X[D]$ у алгоритму *Razdvajanje* са слике 8 индекси L и D најпре без упоређивања померају за по једно место). Према томе, сложеност сортирања раздвајањем задовољава једнакост $T(n) = n-1+T(i-1)+T(n-i)$. Ако претпоставимо да позиција пивота после раздвајања може бити свако $i = 1, 2, \ldots, n$ са једнаком вероватноћом $1/n$, **просечан број корака** је

$$
\begin{aligned}
T(n) &= n-1+\frac{1}{n}\sum_{i=1}^{n}(T(i-1)+T(n-i)) = \\
&= n-1+\frac{1}{n}\sum_{i=1}^{n}T(i-1)+\frac{1}{n}\sum_{i=1}^{n}T(n-i) = n-1+\frac{2}{n}\sum_{i=0}^{n-1}T(i).
\end{aligned}
$$

Ово је потпуна рекурзија, и то управо она која је разматрана у одељку 2.5.5. Њено решење је $T(n) = O(n\log n)$. Дакле, сортирање раздвајањем је заиста ефикасно у просеку.

У пракси је сортирање раздвајањем веома брзо, па заслужује своје друго име (квиксорт, тј. брзо сортирање). Основни разлог за његову брзину, осим елегантне примене декомпозиције, је у томе што се много елемената упоређује са једним истим елементом, пивотом. Пивот се због тога може сметити у регистар, што убрзава упоређивања.

Један од начина да се сортирање раздвајањем убрза је **добар избор базе индукције**. Идеја је да се са индукцијом не почиње увек од јединице. Сортирање раздвајањем, као што је речено, позива се рекурзивно до базног случаја, који обухвата низове дужине 1. Међутим, једноставни алгоритми, као што су сортирање уметањем или сортирање избором, раде сасвим добро за кратке низове. Према томе, може се изабрати да базни случај за сортирање раздвајањем буду низови дужине веће од један (нпр. између 10 и 20, што зависи од конкретне реализације), а да се базни случај обрађује сортирањем уметањем. Другим речима, услов "**if** $Levi < Desni$" замењује се условом "**if** $Levi < Desni - Prag$" (где $Prag$ има вредност између 10 и 20), и додаје се део "**else** ", који извршава сортирање уметањем). Ефекат ове промене је поправка брзине сортирања раздвајањем за мали константни фактор.

5.3.5. Хипсорт. Хипсорт је још један пример брзог алгоритма за сортирање. У пракси он, за велике n, обично није тако брз као сортирање раздвајањем, али није ни много спорији. С друге стране, за разлику од сортирања раздвајањем, његова ефикасност је гарантована. Као код сортирања обједињавањем, сложеност хипсорта у најгорем случају је $O(n \log n)$. За разлику од сортирања обједињавањем, хипсорт је алгоритам сортирања у месту. У вези са хипсортом обратићемо посебно пажњу на почетни део алгоритма — формирање хипа. Алгоритам за формирање хипа илуструје начин на који треба комбиновати конструкцију и анализу алгоритма.

Хип као структура података разматран је у поглављу 3. Претпоставља се да се хип представља имплицитно; његови елементи смештени су у вектор A дужине n, који стаблу одговара на следећи начин: корен је смештен у $A[1]$, а синови чвора записаног у $A[i]$ смештени су у $A[2i]$ и $A[2i+1]$. Стабло задовољава **услов хипа** ако је вредност сваког чвора већа или једнака од вредности његових синова.

Алгоритам хипсорт извршава се на исти начин као и сортирање избором; разлика је у употребљеној структури података за смештање низа. Најпре се елементи низа преуређују тако да чине хип; формирање хипа размотрићемо нешто касније. Ако је хип у вектору A, онда је $A[1]$ највећи елеменат хипа. Заменом $A[1]$ са $A[n]$ постиже се да $A[n]$ садржи највећи елеменат низа. Затим се разматра низ $A[1], A[2], \ldots, A[n-1]$; преуређује се, тако да и даље задовољава услов хипа (проблем је једино нови елеменат $A[1]$). После тога се са тим низом рекурзивно понавља поступак примењен на низ $A[1], A[2], \ldots, A[n]$. Све у свему, алгоритам се састоји од почетног формирања хипа и $n-1$ корака, у којима се врши по једна замена и преуређивање хипа. Преуређивање хипа је у основи исти поступак као алгоритам за уклањање највећег елемента из хипа, одељак 3.3. Формирање хипа је само по себи интересантан проблем, па ће бити посебно разматран. Временска сложеност алгоритма је дакле $O(n \log n)$ ($O(\log n)$ по замени) плус сложеност формирања хипа. Јасно је да је хипсорт сортирање у месту; алгоритам је приказан на слици 10.

Алгоритам Hipsort(X, n);
Улаз: X (низ од n бројева).
Излаз: X (сортирани низ).
begin
  Преуредити X тако да буде хип; {видети наставак текста}
  for $i := n$ **downto** 2 **do**
    zameni $(X[1], X[n])$;
    Preuredi_Hip$(i-1)$
     {процедура слична алгоритму *Skini_max_sa_hipa* на слици 3.10 }
end

Рис. 10. Алгоритам хипсорт.

5.3.5.1. *Формирање хипа.* Сада ћемо се позабавити проблемом формирања хипа од произвољног низа.

Проблем. Дат је низ $A[1], A[2], \ldots, A[n]$ произвољно уређених бројева. Преуредити његове елементе тако да задовољавају услов хипа.

Постоје два природна приступа формирању хипа: одозго надоле и одоздо нагоре. Они одговарају проласку кроз вектор A слева удесно, односно здесна улево (видети слику 11). Пошто ови методи буду описани применом индукције. биће показано да између њих постоји знатна разлика у ефикасности.

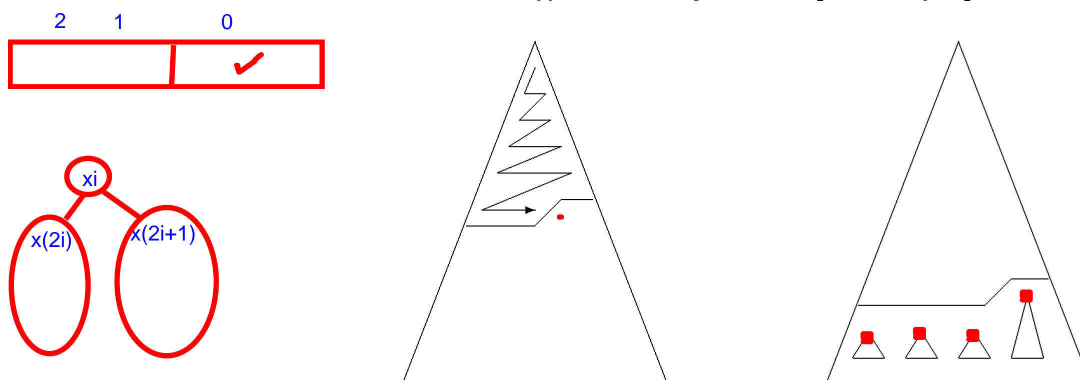


Рис. 11. Формирање хипа одозго надоле и одоздо нагоре.

Размотримо најпре пролазак кроз вектор слева удесно, тј. изградњу хипа одозго надоле.

Индуктивна хипотеза (одозго надоле)**.** Низ $A[1], A[2], \ldots, A[i]$ представља хип.

Базни случај је тривијалан, јер $A[1]$ јесте хип. Основни део алгоритма је уградња елемента $A[i+1]$ у хип $A[1], A[2], \ldots, A[i]$ (видети поглавље 3). Елеменат $A[i+1]$ упоређује се са својим оцем, и замењује се са њим ако је већи од њега; нови елеменат се премешта навише, све док не постане мањи или једнак од оца, или док не доспе у корен хипа. Број упоређивања је у најгорем случају $\lfloor \log_2(i+1) \rfloor$.

Размотримо сада пролазак низа здесна улево, што одговара проласку хипа одоздо нагоре. Волели бисмо да кажемо да је низ $A[i+1], A[i+2], \ldots, A[n]$ хип, у који треба убацити елеменат $A[i]$. Али низ $A[i+1], A[i+2], \ldots, A[n]$ не одговара једном, него колекцији хипова (низ $A[i+1], A[i+2], \ldots, A[n]$ посматрамо као део стабла представљеног низом $A[1], A[2], \ldots, A[n]$). Због тога је индуктивна хипотеза нешто компликованија.

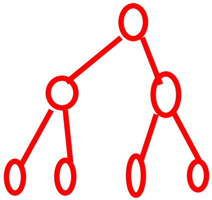
Индуктивна хипотеза (одоздо нагоре)**.** Сва стабла представљена низом $A[i+1], A[i+2], \ldots, A[n]$ задовољавају услов хипа.

Индукција је по i, али обрнутим редоследом, $i = n, n-1, \dots, 1$. Елеменат $A[n]$ очигледно представља хип, што представља базу индукције. Може се закључити и нешто више. Елементи вектора A са индексима од $\lfloor n/2 \rfloor + 1$ до n су листови стабла. Због тога се стабла која одговарају том низу састоје само од корена, па тривијално задовољавају услов хипа. Довољно је дакле да са индукцијом кренемо од $i = \lfloor n/2 \rfloor$. То већ указује да би приступ одоздо нагоре могао бити ефикаснији: половина посла је тривијална; ово је још један пример важности пажљивог избора базе индукције.

Размотримо сада елеменат $A[i]$. Он има највише два сина $A[2i+1]$ и $A[2i]$, који су, према индуктивној хипотези, корени исправних хипова. Због тога је јасан начин укључивања $A[i]$ у хип: $A[i]$ се упоређује са већим од синова, и замењује се са њим ако је потребно. Ово је слично брисању из хипа, видети поглавље 3. Са заменама се наставља наниже низ стабло, док претходна вредност елемента $A[i]$ не дође до места на коме је већа од оба сина. Конструкција хипа одоздо нагоре илустрована је примером у табели 4.

Сложеност (одозго надоле)**.** Корак са редним бројем i захтева највише $\lfloor \log_2 i \rfloor \leq \lfloor \log_2 n \rfloor$ упоређивања, па је временска сложеност $O(n \log n)$. Оцена сложености $O(n \log n)$ није груба, јер је

$$\sum_{i=1}^{n} \lfloor \log_2 i \rfloor \geq \sum_{i=n/2}^{n} \lfloor \log_2 i \rfloor \geq (n/2) \lfloor \log_2 (n/2) \rfloor = \Omega(n \log n).$$



Сложеност (одоздо нагоре)**.** Број упоређивања у сваком кораку мањи је или једнак од двоструке висине разматраног чвора (пошто се чвор мора упоредити са већим од синова, евентуално заменити, и тако даље низ стабло до листа). Према томе, сложеност је мања или једнака од двоструке суме висина свих чворова у стаблу. Израчунајмо дакле суму висина свих чворова, али најпре само за комплетна стабла. Нека је $H(i)$ сума висина свих чворова комплетног бинарног стабла висине i. Низ $H(i)$ задовољава диференцну једначину $H(i) = 2H(i-1) + i$, јер се комплетно бинарно стабло висине i састоји

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

ТАБЛИЦА 4. Пример формирања хипа одоздо нагоре. Бројеви у првој врсти су индекси. Бројеви који су управо учествовали у замени, подебљани су.

од два комплетна бинарна подстабла висине $i-1$ и корена, чија је висина i. Поред тога, $H(0) = 0$. Члан i из диференцне једначине може се уклонити њеним преписивањем у облику $H(i) + i = 2(H(i-1) + (i-1)) + 2$, односно $G(i) = 2G(i-1) + 2$, где је уведена ознака $G(i) = H(i) + i$; $G(0) = H(0) = 0$. Да би се уклонио преостали константни члан 2, обема странама треба додати одговарајућу константу, тако да буде корисна смена $F(i) = G(i) + a$; одатле се добија $a = 2$: $G(i)+2 = 2(G(i-1)+2)$, тј. $F(i) = 2F(i-1)$ и $F(0) = G(0)+2 = 2$. Решење последње диференцне једначине је $F(i) = 2^{i+1}$, па коначно добијамо $H(i) = 2^{i+1} - (i+2)$. Пошто је укупан број чворова комплетног бинарног стабла висине i једнак $2^{i+1} - 1$, закључујемо да је за хипове са $n = 2^i - 1$ чворова временска сложеност формирања хипа одоздо нагоре $O(n)$. Сложеност за хип са n чворова, при чему је $2^i \leq n \leq 2^{i+1} - 1$ мања је или једнака од сложености за хип се $2^{i+1} - 1$ чворова, која је и даље $O(n)$. Основни разлог због кога је приступ одоздо нагоре ефикаснији је у томе да је велики део чворова стабла на његовом дну, а врло мали део при врху, у близини корена. Зато је боље минимизирати обраду чворова на дну.

Ово је још један пример који показује да се добрим избором редоследа индукције може конструисати ефикасан алгоритам. Метод формирања хипа одозго надоле је директнији и очигледнији, али се испоставља да је приступ одоздо нагоре бољи.

5.3.6. Доња граница за сортирање. Полазећи од алгоритама за сортирање сложености $O(n^2)$, усавршавањем смо дошли до алгоритма сложености $O(n \log n)$. Да ли је могуће даље побољшање, односно да ли је сложеност сваког алгоритма за сортирање $\Omega(n \log n)$?

Функција $f(n)$ је доња граница сложености неког проблема ако за сложеност $T(n)$ произвољног алгоритма који решава тај проблем важи $T(n) = \Omega(f(n))$ (или: не постоји алгоритам сложености *мање од* $O(f(n))$, који решава проблем). Доказати да је нека функција доња граница сложености разматраног проблема није лако, јер се доказ односи на све могуће алгоритме, а не само на један конкретан приступ. Неопходно је најпре специфицирати модел који одговара произвољном алгоритму, а затим доказати да је временска сложеност произвољног алгоритма обухваћеног тим моделом већа или једнака од доње границе. У овом одељку размотрићемо један такав модел који се зове **стабло одлучивања**. Стабло одлучивања моделира израчунавања која се састоје највећим делом од упоређивања. Стабло одлучивања није општи модел израчунавања као што је нпр. Тјурингова машина, па су доње границе нешто слабије; али с друге стране, стабла одлучивања су једноставнија и са њима се лакше оперише.

Стабло одлучивања дефинише се као бинарно стабло са две врсте чворова, унутрашњим чворовима и листовима. Сваки унутрашњи чвор одговара питању са два могућа одговора, који су пак придружени двема гранама које излазе из чвора. Сваком листу придружен је један од могућих излаза. Ако се ради о

проблему сортирања, претпостављамо да је улазни низ $x_1, x_2, \ldots, x_n$. Израчунавање почиње од корена стабла. У сваком чвору поставља се једно питање у вези са улазом, и у зависности од добијеног одговора, наставља се левом или десном излазном граном. Кад се достигне лист, излаз придружен листу је излаз израчунавања. На слици 12 приказано је сортирање помоћу стабла одлучивања за $n = 3$. Временска сложеност (у најгорем случају) алгоритма дефинисаног стаблом T једнака је висини T, односно највећем могућем броју питања која треба поставити за неки улаз. Стабло одлучивања према томе одговара алгоритму. Иако стабло одлучивања не може да моделира сваки алгоритам (нпр. не може се израчунати квадратни корен из неког броја коришћењем стабла одлучивања), оно је прихватљив модел за алгоритме засноване на упоређивању. Доња граница добијена за модел стабла одлучивања односи се наравно сано на алгоритме *тог облика.* Сада ћемо искористити стабла одлучивања да бисмо одредили доњу границу за сортирање.

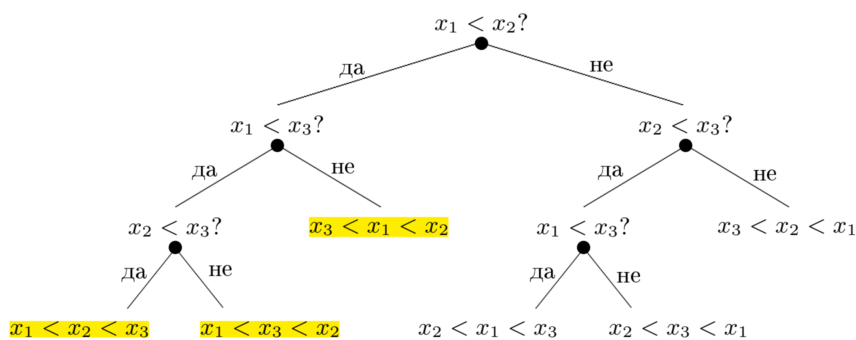


Рис. 12. Сортирање три елемента помоћу стабла одлучивања.

Теорема 5.1. *Произвољно стабло одлучивања за сортирање n елемената има висину $\Omega(n \log n)$.*

Доказательство. Улаз у алгоритам за сортирање је низ $x_1, x_2, \ldots, x_n$. Излаз је исти низ у сортираном редоследу. Другим речима, излаз је *пермутација* улаза: излаз уствари показује како треба испремештати елементе тако да они постану сортирани. На излазу се може појавити свака пермутација, јер улаз може бити задат у произвољном редоследу. Алгоритам за сортирање је коректан ако може да обради све могуће улазе. Према томе, свака пермутација скупа $\{1, 2, \ldots, n\}$ треба да се појави као могући излаз стабла одлучивања за сортирање. Излази стабла одлучивања придружени су његовим листовима. Пошто две различите пермутације одговарају различитим излазима, морају им бити придружени различити листови. Према томе, за сваку пермутацију мора да постоји бар један лист. Пермутација има $n!$, па пошто претпостављамо да је стабло бинарно, његова висина је најмање $\log_2(n!)$. Према Стирлинговој

формули је

$$n! = \sqrt{2\pi n}\Big(\frac{n}{e}\Big)^n \left(1 + O\left(\frac{1}{n}\right)\right),$$

односно $\log_2(n!) = \Omega(n \log n)$, чиме је доказ завршен. □

Оваква врста доње границе зове се **теоријско–информациона** доња граница, јер она не зависи од самог израчунавања (ми, на пример, нисмо дефинисали каква питања су дозвољена), него само од количине информација садржане у излазу. Значај доње границе је у томе да произвољан алгоритам сортирања захтева $\Omega(n \log n)$ упоређивања у најгорем случају, јер он мора да разликује $n!$ различитих излаза, а у једном кораку може да изабере само једну од две могућности. Могли смо да дефинишемо стабло одлучивања као стабло у коме сваки чвор има три сина (који, на пример, одговарају исходима ”$<$”, ”$=$” и ”$>$”). У том случају би висина била најмање $\log_3 n!$, што је и даље $\Omega(n \log n)$. Другим речима, доња граница $\Omega(n \log n)$ односи се на сва стабла одлучивања са константним бројем грана по чвору.

Овај доказ доње границе показује да сваки алгоритам за сортирање заснован на упоређивањима има сложеност $\Omega(n \log n)$. Сортирање је ипак могуће извршавати брже коришћењем специјалних особина бројева, или изводећи алгебарске манипулације са њима. На пример, ако имамо n природних бројева из опсега од 1 до $4n$, сортирање разврставањем ће их уредити за време $O(n)$. Ово не противречи доказаној доњој граници, јер сортирање разврставањем не користи упоређивања. Користи се чињеница да вредности бројева могу да се ефикасно користе као *адресе*.

Кад разматрамо стабла одлучивања, обично игноришемо њихову величину, а занима нас само висина. Због тога чак и једноставни алгоритми линеарне временске сложености могу да одговарају стаблима одлучивања са експоненцијалним бројем чворова. Величина није битна, јер се стабло уствари не формира експлицитно. Стабло се користи као средство за доказивање доњих граница. Игнорисање величине чини доказ јачим, јер се може применити на програме експоненцијалне дужине. С друге стране, техника може да буде превише моћна, чинећи је бескорисном за извођење доњих граница за проблеме који се не могу решити програмима разумне величине, али се могу решити програмима експоненцијалне дужине (на пример, програмом са таблицом која садржи сва могућа решења). Стабла одлучивања су **неуниформни** модели израчунавања. Стабло зависи од n, величине улаза. Испоставља се да се могу формирати стабла одлучивања полиномијалне висине — али експоненцијалне величине — за проблеме који вероватно захтевају експоненцијално време извршавања, па су стабла одлучивања понекад превише оптимистичка. Другим речима, доња граница по моделу стабла одлучивања може да буде далеко испод стварне сложености проблема. С друге стране, ако је доња граница једнака горњој граници за неки конкретан алгоритам, као што је случај са сортирањем, онда доња граница показује да се алгоритам не може убрзати ни употребом много већег простора.

5.4. Ранговске статистике

За елемеат x_i задатог низа елемената $S = x_1, x_2, \ldots, x_n$ кажемо да има **ранг** k у S ако је x_i k-ти најмањи елеменат у S. Рангови елемената у низу могу се лако одредити сортирањем. Ипак, постоји доста проблема у вези са ранговима који се могу решити и без сортирања. У овом одељку размотрићемо најпре проблем истовременог налажења најмањег и највећег елемента, а затим општи проблем налажења k-тог најмањег елемента.

5.4.1. Највећи и најмањи елеменат. Налажење највећег или најмањег елемента у низу је једноставно. Ако знамо највећи елеменат у низу дужине $n-1$, онда треба да га још упоредимо са n-тим елементом да бисмо пронашли највећи елеменат низа дужине n. Налажење највећег елемента низа дужине 1 је тривијално. Процес захтева једно упоређивање по елементу, почевши од другог елемента. Укупан број упоређивања је дакле $n-1$.

Претпоставимо сада да је циљ пронаћи истовремено најмањи и највећи елеменат низа.

Проблем. Пронаћи највећи и најмањи елеменат датог низа.

Најједноставније је решити ова два проблема независно. Укупан број упоређивања је $2n-3$: $n-1$ да се пронађе највећи, а затим $n-2$ да се пронађе најмањи елеменат (јер се највећи при томе не мора разматрати). Може ли се ово побољшати? Размотримо још једном индуктивни приступ. Претпоставимо да умемо да решимо проблем за датих $n-1$ елемената, и да је потребно решити проблем за n елемената (базни случај је тривијалан). Треба упоредити нови елеменат са највећим и најмањим до тада. То су два упоређивања, што значи да ће број упоређивања поново бити $2n-3$, јер за први елеменат није потребно ни једно упоређивање.

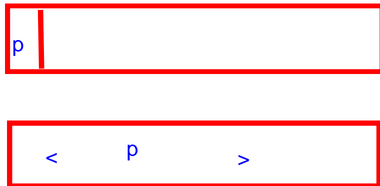
Следећи покушај могао би да буде да се решење прошири за више од једног елемента. Покушајмо да одједном додајемо по два елемента: претпоставимо да умемо да решимо проблем за $n-2$ елемента, и покушајмо да га решимо за n елемената. Да би конструкција била комплетна, морају се решити два базна случаја, $n = 1$ и $n = 2$. Нека су max и min највећи и најмањи елементи међу првих $n-2$ елемената (познати на основу индуктивне хипотезе) и нека су x_{n-1} и x_n новододати елементи. Лако је видети да су за налажење новог максимума и минимума довољна три упоређивања: најпре x_{n-1} са x_n, а затим већи од њих са max, а мањи са min. Укупно алгоритам захтева око $3n/2$ уместо $2n$ упоређивања! Може ли се алгоритам даље побољшати ако се одједном дода по три (или четири) елемента? Користећи исти приступ, долази се до истог броја упоређивања. Испоставља се да се број упоређивања не може смањити било којим методом. Занимљиво је да и приступ заснован на декомпозицији такође доводи до алгоритма са око $3n/2$ упоређивања.

5.4.2. Налажење k-тог најмањег елемента. Посматрајмо сада општи проблем.

Проблем. За задати низ елемената $S = x_1, x_2, \ldots, x_n$ и природни број k, $1 \leq k \leq n$, одредити k-ти најмањи елеменат у S.

Ово је проблем одређивања **ранговских статистика**, односно **селекције**. Ако је k врло близу 1 или n, онда се k-ти најмањи елеменат може одредити извршавањем алгоритма за налажење минимума (максимума) k пута. Овај приступ захтева око kn упоређивања. Сортирање је боље од овог једноставног алгоритма, сем ако је k реда величине $\log n$. Међутим, постоји други алгоритам који ефикасно проналази k-ти најмањи елеменат за произвољно k.

Идеја је применити декомпозицију на исти начин као и код алгоритма сортирање раздвајањем, сем што је овог пута довољно решити само један од два потпроблема. Код сортирања раздвајањем низ се раздваја помоћу пивота на два подниза. Два подниза се затим сортирају рекурзивно. Овде је довољно одредити који од поднизова садржи k-ти најмањи елеменат, а онда алгоритам применити рекурзивно *само на тај подниз*. Остали елементи могу се игнорисати. Алгоритам је приказан на слици 13.



Алгоритам Selekcija(X, n, k);
Улаз: X (низ од n бројева) и k (природни број).
Излаз: S (k-ти најмањи елеменат; низ X је промењен).
begin
    if $(k < 1)$ **or** $(k > n)$ **then** *print* "грешка"
    else
        $S := Sel(1, n, k)$
end
function Sel$(Levi, Desni, k)$;
begin
    if $Levi = Desni$ **then**
        $Sel := Levi$
    else
        $S = Razdvajanje(X, Levi, Desni)$; {видети слику 8}
        if $S - Levi + 1 \geq k$ **then**
            $Sel := Sel(Levi, S, k)$
        else
            $Sel := Sel(S + 1, Desni, k - (S - Levi + 1))$
end

Рис. 13. Алгоритам за одређивање k-тог најмањег елемента.

Сложеност. Као и код сортирања раздвајањем, лош избор пивота води квадратном алгоритму. Средња сложеност овог алгоритма може се оценити на

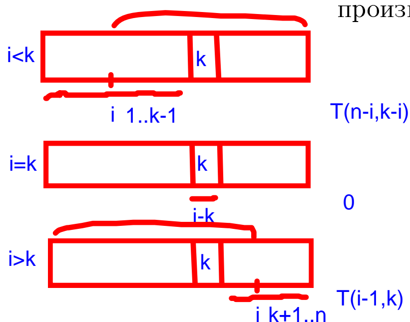
T(n)=T(n/2)+n-1

T(n)=O(n)

сличан начин као код сортирања раздвајањем. Нека је са $T(n,k)$ означен просечан број упоређивања у алгоритму на улазима дужине n кад се тражи k-ти најмањи елеменат. Претпоставимо да после раздвајања ($n-1$ упоређивање) пивот са једнаком вероватноћом $1/n$ може доћи на сваку позицију $i = 1, 2, \ldots, n$. Разматраћемо мало измењену варијанту алгоритма, у којој се даље извршавање одмах прекида ако је позиција пивота k (и слично у даљим рекурзивним позивима), јер је тада пронађен k-ти најмањи елеменат. За $i < k$ (односно $i > k$) долази се до рекурзивног позива алгоритма сложености $T(n-i, k-i)$ (односно $T(i, k)$), јер преостаје проналажење $(k-i)$-тог најмањег елемента од њих $n-i$ (односно k-тог најмањег елемента од њих i). Према томе, диференцна једначина за $T(n,k)$ је

$$T(n,k) = \frac{1}{n}\sum_{i=k}^{n-1} T(i,k) + \frac{1}{n}\sum_{i=n-k+1}^{n-1} T(i, n-k+1) + n - 1.$$

Индукцијом се може доказати да је $T(n,k) \le 4n$ за свако $k = 1, 2, \ldots, n$. За $n = 2$ ово се непосредно проверава, а из индуктивне хипотезе да за $i < n$ и произвољно m, $1 \le m \le i$, важи $T(i,m) \le 4i$ следи



$$\begin{aligned}
T(n,k) &\le n - 1 + \frac{1}{n}\sum_{i=k}^{n-1} 4i + \frac{1}{n}\sum_{i=n-k+1}^{n-1} 4i \\
&= n - 1 + \frac{4}{n}\left(\frac{n-1+k}{2}(n-k) + \frac{2n-k}{2}(k-1)\right) \\
&= n - 1 + \frac{2}{n}\left(n^2 + 2nk - 2k^2 - 3n + 2k\right) \\
&= 4n - \frac{1}{n}\left((n-2k)^2 + 3(n-k) + 4n\right) < 4n.
\end{aligned}$$

Према томе, за произвољно k је просечна сложеност овог алгоритма $O(n)$.

Коментар. Већина примена ранговских статистика захтева одређивање *медијане*, односно $n/2$-тог најмањег елемента. Алгоритам *Selekcija* је одличан алгоритам за ту сврху. Не постоји једноставнији алгоритам за налажење само медијане. Другим речима, уопштење проблема налажења медијане на налажење k-тог најмањег елемента чини алгоритам једноставнијим. Ово је такође пример појачавања индуктивне хипотезе, јер рекурзија захтева произвољне вредности за k.

5.5. Компресија података

Компресија података је поступак смањења простора који заузимају подаци. Компресија података је такође важна приликом њиховог преноса, кад је цена обраде података мања од цене њиховог преноса. Компресија података широко се примењује и представља област која се брзо развија. У овом одељку размотрићемо само један алгоритам погодан за компресију.

Због једноставности претпоставимо да су подаци које треба компримовати текст смештен у фајл. Сваки знак текста представљен је јединственим низом бита — **кодом** тог знака. Ако су дужине кодова свих знакова једнаке (што је управо случај са стандардним кодовима, као што је ASCII), број бита који представљају фајл зависи само од броја знакова у њему. Да би се постигла уштеда, морају се неки знаци кодирати мањим, а неки већим бројем бита. Нека се у фајлу појављује n различитих знакова, и нека се знак s_i појављује f_i пута, $1 \leq i \leq n$. Ако са l_i означимо број бита у коду знака s_i, у коду означеном са E, онда је укупан број бита употребљених за представљање кодираног фајла $L(E,F) = \sum_{i=1}^{n} l_i f_i$, где је са $F = f_1, f_2, \ldots, f_n$ означен низ учестаности (фреквенција) знакова у фајлу. Кодирање фајла је поступак његове трансформације, заменом знакова одговарајућим кодовима из E. Инверзна трансформација која кодове замењује знацима је декодирање.

Основни услов који код E мора да задовољи је да омогућује једнозначно декодирање, односно реконструкцију полазног фајла на основу његове кодиране верзије. Пре него што пређемо на тражење оптималног кода, размотримо услове које кô д треба да задовољи да би било могуће једнозначно декодирање. Један од начина да се једножначност обезбеди, је да се коду наметне услов да ни једна кодна реч није префикс некој другој. Кодови који задовољавају овај услов зову се *префиксни кодови*. Сваки бинарни префиксни (онај који знаке кодира низовима бита) кô д може се представити бинарним стаблом, видети пример на слици 14. Полазећи од бинарног стабла у коме су гране ка левим, односно десним синовима означене са 0, односно 1, на природан начин се свакој кодној речи једнозначно придружује чвор. Полази се од корена, а кретање се контролише кодном речи: ако је наредни бит 0, прелази се у левог, а ако је 1, у десног сина. Код префиксних кодова кодним речима очигледно одговарају листови кодног стабла. Важи и обрнуто, кô д задат кодним стаблом, у коме су кодне речи листови, префиксни је кô д. Декодирање префиксног кода је једноставно: из низа бита који представљају кодирани фајл редом се издваја једна по једна кодна реч. Процес је једнозначно одређен управо зато што је кô д префиксни. На пример, низ бита 011100010001, може се у коду представљеним стаблом на слици 14 декодирати само на један начин: $011 - 1 - 00 - 010 - 00 - 1$. Проблем ефикасног кодирања може се сада формулисати на следећи начин.

Проблем. Задат је текст са n различитих знакова, тако да су учестаности знакова задате низом $F = f_1, f_2, \ldots, f_n$. Одредити префиксни кô д E који минимизира број бита $L(E,F)$ употребљених за кодирање.

За префиксни кô д који минимизира вредност $L(E,F)$ каже се да је оптимални кô д; он очигледно зависи од низа учестаности знакова F. Циљ је за дати низ F одредити оптимални префиксни кô д (било који; у општем случају постоји више различитих оптималних кодова). Размотримо због тога мало детаљније особине оптималних кодова. Јасно је најпре да у оптималном коду чешћи знаци морају бити кодирани мањим бројем бита, односно, ако је

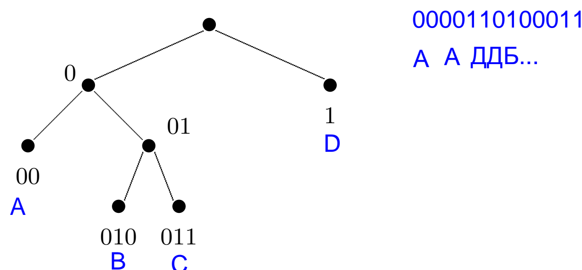


Рис. 14. Представљање префиксног кода стаблом.

за произвољна два знака c_i, c_j, знак c_i чешћи од знака c_j (тј. $f_i > f_j$), онда за одговарајуће дужине кодних речи l_i, l_j мора да важи $l_i \leq l_j$. Заиста, претпоставимо супротно, да је $f_i > f_j$ и $l_i > l_j$. Тада је $(f_i - f_j)(l_i - l_j) > 0$, или $f_i l_i + f_j l_j > f_j l_i + f_i l_j$. Ова неједнакост супротна је претпоставци да се ради о оптималном коду: заменом кодних речи за знакове c_i и c_j добија се кôд са мањом вредношћу $L(E, F)$ од полазне:

$$
\begin{aligned}
& f_1 l_1 + f_2 l_2 + \cdots + f_j l_i + \cdots + f_i l_j + \cdots + f_n l_n < \\
< \quad & f_1 l_1 + f_2 l_2 + \cdots + f_i l_i + \cdots + f_j l_j + \cdots + f_n l_n.
\end{aligned}
$$

У кодном стаблу оптималног префиксног кода сваки унутрашњи чвор мора да има оба сина: у противном би се кôд могао поједноставити скраћивањем кодних речи у листовима подстабла испод чвора са само једним сином, видети пример на слици 15. Одатле следи друга карактеристика оптималних префиксних кодова: постоји оптимални префиксни кôд у коме су кодне речи за два знака са најмањим фреквенцијама — листови са истим оцем, на највећем могућем растојању од корена (другим речима, те кодне речи се разликују само по последњем биту). Заиста, два знака c_i, c_j са најмањим учестаностима морају бити кодирана са највише бита, односно морају бити представљени листовима у најдаљем слоју (на највећем растојању од корена). Знаци c_i, c_j морају бити у истом слоју; ако ти знаци нису синови истог оца, онда се могу извршити замене кодова знакова у последњем слоју (та замена не мења број употребљених бита $L(E, F)$), тако да у новом оптималном коду знаци c_i, c_j буду синови истог оца (видети пример на слици 16).

Алгоритам кодирања (налажења оптималног кода) заснива се на свођењу проблема са n знакова на проблем са $n-1$ знаком; базни случај је тривијалан. Као и обично, проблем је како дефинисати индуктивну хипотезу и којим редоследом елиминисати знакове. Свођење је у овом случају нешто друкчије од досад виђених. Уместо да просто елиминишемо један знак, ми уводимо један нови знак уместо два постојећа. Ова техника је мало компликованија, али води циљу: величина улаза се смањује. Нека су c_i и c_j два произвољна знака са најмањим учестаностима. Према горе доказаном тврђењу, постоји оптимални кôд у коме знацима c_i и c_j одговарају листови на максималном растојању од корена. Два знака c_i и c_j замењујемо новим знаком који можемо да означимо са

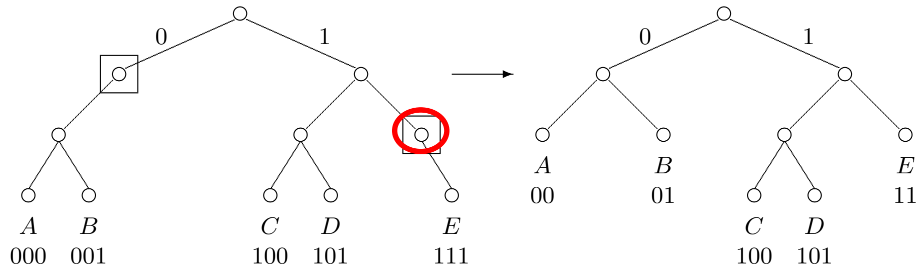


Рис. 15. Кодно стабло у коме неки чвор има само једног сина није кодно стабло оптималног кода.

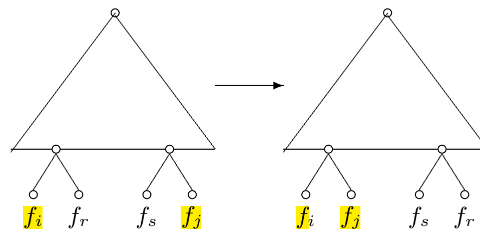


Рис. 16. Трансформација кодног стабла, после које два знака са најмањим фреквенцијама f_i и f_j постају синови истог оца.

c_ic_j и чија је фреквенција $f_i + f_j$. Алфабет сада има $n-1$ знак, при чему збир учестаности знакова није промењен, па се за њега према индуктивној хипотези може одредити оптимални кôд. Оптимални кôд за полазни алфабет добија се тако што се листу c_ic_j у коду за $n-1$ знак додају два сина, листа који одговарају знацима c_i и c_j. Оптимални кôд добијен описаном конструкцијом зове се Хофманов кôд (према имену аутора алгоритма, D. Huffman).

Реализација. Операције које се извршавају приликом формирања Хофмановог кода су

- уметање у структуру података,
- брисање два знака са најмањим фреквенцијама из структуре, и
- конструкција кодног стабла.

Хип је погодна структура података за прве две операције, јер се тада те операције у најгорем случају извршавају за $O(\log n)$ корака. Алгоритам за формирање Хофмановог кода приказан је на слици 17.

Пример 5.1. Нека је дат фајл у коме се појављују знаци A, B, C, D, E и F са фреквенцијама редом 5, 2, 3, 4, 10 и 1. Два знака са најмањим фреквенцијама су F и B; они се замењују новим знаком BF са фреквенцијом $1+2=3$. Даља

Алгоритам Hofman(S, F);
Улаз: S (стринг знакова који се појављују у тексту) и F (фреквенције).
Излаз: T (кодно стабло Хофмановог кода за S).
begin
  убацити све знаке у хип H према њиховим фреквенцијама;
  while H непразан **do**
    if H садржи само један знак X **then**
      формирај стабло T које има само корен, чвор X
    else
      скини са хипа H знакове X, Y са најмањим фреквенцијама f_X, f_Y;
      замени X и Y новим знаком Z са фреквенцијом $f_Z = f_X + f_Y$;
      убаци Z у хип H;
      повежи X и Y тако да буду синови Z у T; $\{Z$ још нема оца$\}$
end

Рис. 17. Алгоритам за формирање Хофмановог кода.

замењивања приказана су у следећој табели.

1	2	3	4	5	6	нови знак
A	B	C	D	E	F	BF
5	2	3	4	10	1	3
A	C	D	E	BF		CBF
5	3	4	10	3		6
A	D	E	CBF			AD
5	4	10	6			9
E	CBF	AD				$CBFAD$
10	6	9				15
E	$CBFAD$					$ECBFAD$
10	15					25

На слици 18 приказано је добијено кодно стабло.

Сложеност. Сложеност додавања новог чвора стаблу ограничена је константом. Уметања и брисања из хипа извршавају се у $O(\log n)$ корака. Сложеност алгоритма је дакле $O(n \log n)$.

5.6. Тражење узорка у тексту

Нека су $S = s_1 s_2 \dots s_n$ и $P = p_1 p_2 \dots p_m$ два стринга (низа знакова из коначног алфабета). За први од њих рећи ћемо да је *текст*, а за други да је *узорак*. Подстринг стринга S је стринг $s_i s_{i+1} \dots s_j$ од узастопних знакова S.

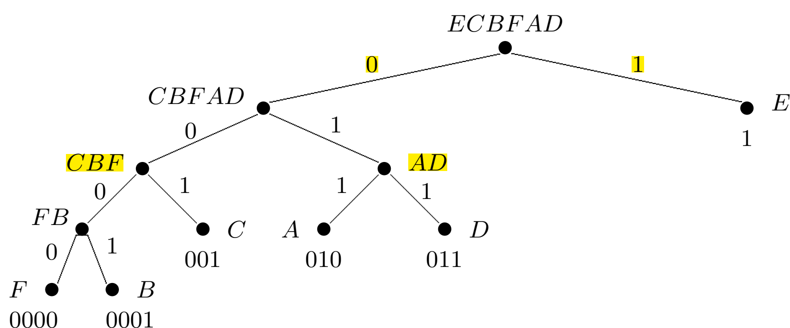


Рис. 18. Стабло Хофмановог кода из примера 5.1.

Проблем. За дати текст $S = s_1 s_2 \dots s_n$ и узорак $P = p_1 p_2 \dots p_m$ установити да ли постоји подстринг S једнак P, а ако постоји, пронаћи прву појаву узорка у тексту, односно најмањи индекс k такав да је $s_{k+i-1} = p_i$ за $i = 1, 2, \dots, m$.

Типична ситуација у којој се наилази на овај проблем је кад се тражи нека реч у текстуалном фајлу. Проблем има примену и у другим областима, на пример у молекуларној биологији, где је корисно пронаћи неке узорке у оквиру великих молекула РНК или ДНК.

На први поглед проблем изгледа једноставно. Довољно је упоредити узорак P са свим могућим подстринговима S облика $s_k s_{k+1} \dots s_{k+m-1}$ дужине m, при чему k узима вредности редом $1,\ 2, \dots, n-m+1$. Упоређивање узорка са подстрингом врши се знак по знак слева удесно, све док се не установи да су сви знаци узорка једнаки одговарајућим знацима подстринга (у том тренутку прекида се даље прегледање подстрингова), или док се не наиђе на неслагање $p_i \neq s_{k+i-1}$ за неко i, $1 \leq i \leq m$:

$$\begin{array}{ccccccccc} & & & & j = k+i-1 & & & & \\ & & & & \downarrow & & & & \\ s_1 & \dots & s_k & \dots & s_j & \dots & s_{k+m-1} & \dots & s_n \\ & & p_1 & \dots & p_i & \dots & p_m & & \\ & & & & \uparrow & & & & \\ & & & & i & & & & \end{array}$$

У другом случају узорак се "помера" за један знак удесно, односно наставља се са провером једнакости p_1 са s_{k+1}. Овај једноставан алгоритам за тражење узорка у стрингу приказан је на слици 19. Број упоређивања знакова мањи је од mn, па је сложеност овог алгоритма $O(mn)$ у најгорем случају.

Нерационалност овог алгоритма огледа се у враћању уназад показивача на текст после наиласка на неслагање. Претпоставимо да је до неслагања дошло на позицији i у узорку, тј. за неки индекс j у тексту је $p_i \neq s_j$ и $p_1 p_2 \dots p_{i-1} = s_{j-i+1} s_{j-i+2} \dots s_{j-1}$ (видети слику 20). Поставља се питање: колико узорак најмање треба померити удесно, односно који знак p_l узорка треба поставити испод знака s_j текста, тако да се (мањи) део узорка $p_1 p_2 \dots p_{l-1}$,

Алгоритам Nadji_uzorak(S, n, P, m);
Улаз: S (стринг дужине n) и P (узорак дужине m).
Излаз: $Start$ (индекс почетка првог подстринга S једнаког P, ако такав постоји, односно 0 у противном).
begin
  $Start := 0$;
  $i := 1$; {показивач на узорак}
  $j := 1$; {показивач на стринг}
  while $i \leq m$ **and** $j \leq n$ **do** {док има наде да се нађе подстринг једнак узорку}
    if $P[i] = S[j]$ **then**
      $i := i + 1$; $j := j + 1$ {поклапање: напредовање и у стрингу и у узорку}
    else
      $j := j - i + 2$; $i := 1$ {неслагање: узорак клизи за једно место удесно}
  if $i > m$ **then**
    $Start := j - i + 1$ {пронађен је подстринг једнак узорку}
end

Рис. 19. Најједноставнији алгоритам за тражење узорка у тексту.

$l < i$, и даље слаже са делом текста $s_{j-l+1}s_{j-l+2}\ldots s_{j-1}$, и да при томе буде $p_l \neq p_i$? Пошто је $l < i$, биће $p_{i-l+1}p_{i-l+2}\ldots p_{i-1} = s_{j-l+1}s_{j-l+2}\ldots s_{j-1}$. Због тога је једнакост $p_1p_2\ldots p_{l-1} = s_{j-l+1}s_{j-l+2}\ldots s_{j-1}$ еквивалентна са слагањем $p_1p_2\ldots p_{l-1} = p_{i-l+1}p_{i-l+2}\ldots p_{i-1}$ префикса узорка дужине $l-1$ са делом узорка који се завршава са p_{i-1} — што је услов који не зависи од текста, него само од узорка! Пошто и услов $p_l \neq p_i$ зависи само од узорка, јасно је да индекс l зависи само од индекса i и знакова узорка p_1, p_2,..., p_i. Могуће је дакле најпре обрадити узорак, тако да се у посебан вектор h на позицију $h[i]$ смести израчунати индекс l, $i = 1, 2, \ldots, m$. Ако ни за једно $l < i$ није истовремено $p_1p_2\ldots p_{l-1} = p_{i-l+1}p_{i-l+2}\ldots p_{i-1}$ и $p_l \neq p_i$, онда се ставља $h[i] = 0$: p_1 се поставља испод s_{j+1}.

Вратимо се на тражење узорка у тексту, и то на ситуацију на слици 20, кад је прво неслагање $p_i \neq s_j$. Тада се очитава вредност $l = h[i]$ и узорак помера удесно за $i - h[i]$, па се (ако је $l > 0$) проверава да ли је $p_l = s_j$. Ако јесте, онда се упоређују p_{l+1} и s_{j+1}, а ако није, онда се поступа на већ познат начин: очитава вредност $h[l]$ а узорак се помера даље удесно за $l - h[l]$, итд. Уколико је пак $l = 0$, онда се напредује у тексту, тј. p_1 се поставља испод s_{j+1}.

Описани алгоритам, који је добио име КМР по својим ауторима Кнуту, Морису и Прату (Knuth, Moris, Pratt), приказан је на слици 21. Алгоритам најпре учитава узорак и формира табелу h, која одређује колико знакова треба померити узорак удесно у случају неслагања. Затим се прелази на тражење узорка у тексту. Пошто нема враћања, улаз се може учитавати знак по знак. Оператор **cand** у унутрашњој петљи је "условна конјункција"

(скраћеница од conditional and), а његов ефекат је да се p_i и s_j упоређују само ако је $i > 0$. У унутрашњој **while**-петљи се, после откривања неслагања $p_i \neq s_j$, узорак помера удесно за $i - h[i]$ позиција, тј. показивач на узорак уместо i добија нову вредност $h[i]$; при томе се претпоставља да су елементи вектора већ израчунати. Овај корак се понавља све док не буде $i = 0$ (што значи да ни један почетак узорка не може бити једнак префиксу улазног текста који се завршава са s_j) или $p_i = s_j$ (што значи да је $p_1 \ldots p_{i-1} p_i = s_{j-i+1} \ldots s_{j-1} s_j$ за нову вредност i). Спољашња **while**-петља помера за по једно место показиваче на узорак и текст.

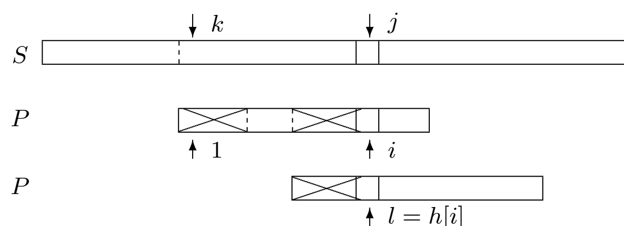


Рис. 20. Илустрација идеје на којој се заснива алгоритам KMP.

Алгоритам KMP(S, n, P, m);
Улаз: S (текст — стринг дужине n) и P (узорак дужине m).
  {Претпоставља се да је низ h израчунат}
Излаз: $Start$ (индекс почетка првог подстринга S једнаког P, ако такав постоји, односно 0 у противном).
begin
  $Start := 0$;
  $i := 1$; {показивач на узорак}
  $j := 1$; {показивач на текст}
  while $i \leq m$ **and** $j \leq n$ **do** {док има наде да се нађе подстринг једнак узорку}
    while $i > 0$ **cand** $P[i] \neq S[j]$ **do**
      $i := h[i]$; {померање узорка удесно за $i - h[i]$}
    $i := i + 1$; $j := j + 1$; {напредовање у тексту и у узорку}
  if $i > m$ **then**
    $Start := j - i + 1$ {пронађен је подстринг једнак узорку}
end

Рис. 21. Алгоритам KMP за налажење узорка у тексту.

Размотримо сада како се ефективно могу израчунати елементи вектора h. Као што је речено, ако је $p_i \neq s_j$, онда је $l = h[i]$ индекс знака узорка кога

следећег треба упоредити са знаком текста s_j. Другим речима, l је највећи индекс за који је $p_1p_2\dots p_{l-1} = p_{i-l+1}p_{i-l+2}\dots p_{i-1}$ и $p_l \neq p_i$. Уз услов $h[1] = 0$ овим су одређени сви чланови вектора h. Да бисмо поједноставили проблем, уведимо вектор g исте дужине, тако да буде $g[1] = 0$, а за $i > 1$ је $g[i]$ највећи индекс l, такав да је $p_1p_2\dots p_{l-1} = p_{i-l+1}p_{i-l+2}\dots p_{i-1}$. Који су следећи индекси $l_2 > l_3 > \cdots$, такви да се p_{l_r} може поставити наспрам p_i, односно такви да је $p_1p_2\dots p_{l_r-1} = p_{i-l_r+1}p_{i-l_r+2}\dots p_{i-1}$ (тј. суфикс дужине $l_r - 1$ стринга $p_1p_2\dots p_{i-1}$ једнак је префиксу тог стринга исте дужине), $r = 2, 3, \dots$? Јасно је да су то индекси $l_2 = g[g[l]] = g^2[l]$, $l_3 = g^3[l]$, ... који чине опадајући, па дакле коначан низ природних бројева. Користећи ову чињеницу, лако је израчунати елементе вектора h кад се зна g. Нека је $g[i] = l$; тада је

$$h[i] = \begin{cases} l, & \text{ако} \quad p_i \neq p_l \\ h[l], & \text{ако} \quad p_i = p_l \end{cases},$$

тј. $h[i]$ је једнако $g[i]$, или се изражава преко већ израчунатог елемента $h[l]$, $l < i$.

Елементи вектора g се такође могу израчунавати рекурзивно, полазећи од једнакости $g[1] = 0$. Нека су већ израчунати $g[1]$, $g[2]$,..., $g[i]$. Тада је $g[i+1]$ највећи индекс $l + 1$, такав да је $p_1p_2\dots p_l = p_{i-l+1}p_{i-l+2}\dots p_i$. Ова једнакост за последицу има $p_1p_2\dots p_{l-1} = p_{i-l+1}p_{i-l+2}\dots p_{i-1}$, па је $l \in \{h[i], h^2[i], \dots\}$; подразумева се да овај скуп не обухвата нулу. Према томе, за вредности l из овог скупа редом се проверава да ли је $p_l = p_i$, па се за $g[i+1]$ узима се прво l за које је $p_l = p_i$. Ако ни за једно l из скупа није $p_l = p_i$, онда је $g[i+1] = 1$.

Алгоритам *Pomaci* за израчунавање вектора h (и успут вектора g), који је практично идентичан са алгоритмом за тражење узорка у тексту, приказан је на слици 22. При томе се показивач j замењује са $h[j]$ уместо са $g[j]$, што представља побољшање, јер се смањује број итерација. Тиме се постиже "аутономија" израчунавања h, тј. наредбе у којима се појављује вектор g постају непотребне. На слици 23 је шематски приказана унутрашња петља алгоритма.

Сложеност. Број извршавања наредбе $j := h[j]$ у унутрашњој петљи (друга наредба **while** на слици 22) која смањује j, мањи је или једнак од броја извршавања наредбе $j := j + 1$ у спољашњој петљи, која за по један повећава j: j је увек позитивно. Међутим, број инкрементирања j једнак је броју инкрементирања i, па је дакле мањи од m. Према томе, израчунавање h обавља се за време $O(m)$. На сличан начин наредба $i := h[i]$ у унутрашњој петљи на слици 21 (која смањује i бар за 1) не може се извршити више пута од наредбе $j := j + 1$ (а тиме и наредбе $i := i + 1$ која повећава i за 1) у спољашњој петљи. Према томе, у унутрашњој петљи се узорак помера удесно највише n пута, па је временска сложеност овог алгоритма $O(n)$. Приметимо да временска сложеност алгоритма KMP не зависи од величине алфабета.

Алгоритам Pomaci(P, m);
Улаз: P (узорак, стринг дужине m).
Излаз: h (низ дужине m).
begin
    $i := 1$; {индекс елемента h који се израчунава}
    $j := 0$; {показивач на смакнути узорак; ускакање у петљу}
    $h[1] := 0; g[1] := 0$; {неслагање на првом знаку у KMP: померање удесно}
    while $i < m$ **do**
        {на овом месту је $P[i-j+k] = P[k]$ за $k = 12, \dots j-1]$ — инваријанта петље}
        while $j > 0$ **cand** $P[i] \neq P[j]$ **do**
            $j := h[j]$; {смањивање индекса j; може и $j := g[j]$;, али је овако боље!}
        $i := i+1$; $j := j+1$; {паралелно напредовање показивача}
        $g[i] := j$;
        {нека је s симбол у тексту, са којим се $P[i]$ упоређује у алгоритму KMP}
        if $P[i] = P[j]$ **then**
            $h[i] := h[j]$ {ако $P[i] \neq s$, онда је и $P[j] \neq s$; удесно на $h[j]$}
        else
            $h[i] := j$ {ако $P[i] \neq s$, онда проверити да ли је $P[j] \neq s$; удесно на j}
end

Рис. 22. Алгоритам за израчунавање табеле h.

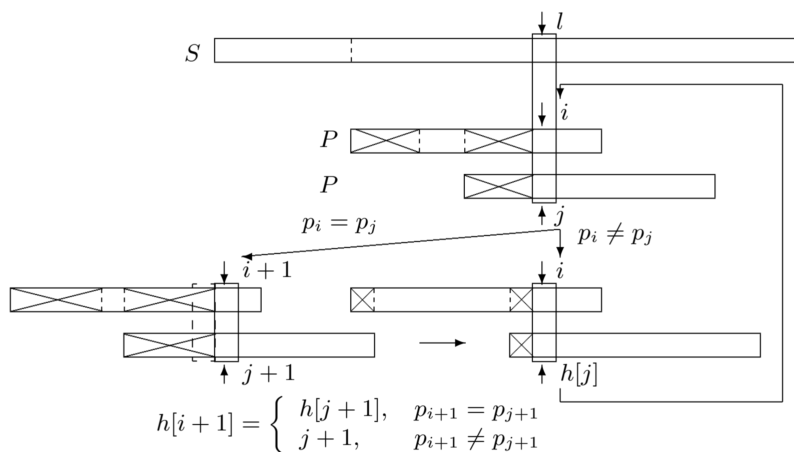


Рис. 23. Унутрашња петља и израчунавање једног елемента табеле h у алгоритму *Pomaci*.

Пример 5.2. Нека је узорак $P = abaababaabaab$. Алгоритам са слике 22 даје следеће векторе h и g:

i	1	2		3	4		5	6	7		8	9	10	11	12		13
p_i		b		a	a		b	a	b		a	a	b	a	a		b
j	0	1	0	1	2	1	2	3	4	2	3	4	5	6	7	4	5
p_j		a		a	b	a	b	a	a	b	a	a	b	a	b	a	b
$p_i = p_j$?		$\neq$		$=$	$\neq$	$=$	$=$	$=$	$\neq$	$=$	$=$	$=$	$=$	$=$	$\neq$	$=$	$=$
$h[i]$	0	1		0	2		1	0	4		0	2	1	0	7		1
$g[i]$	0	1		1	2		2	3	4		3	4	5	6	7		5

Применимо алгоритам КМР да бисмо пронашли овај узорак у тексту

$$S = abaababaabacabaababaabaab.$$

После слагања првих 11 знакова, долази се до неслагања на знаку c у S. У том тренутку је $i = j = 12$:

```
                                 j = 12
                                   ↓
a  b  a  a  b  a  b  a  a  b  a    c    a  b  a  a  b  a  b  a  ...
a  b  a  a  b  a  b  a  a  b  a    a    b
                                   ↑
                                 i = 12
```

Прва итерација у унутрашњој петљи на слици 21 поставља $i = h_{12} = 7$, што одговара померању узорка пет знакова удесно; тиме се седми знак узорка поставља испод 12-тог знака текста:

```
                                 j = 12
                                   ↓
a  b  a  a  b  a  b  a  a  b  a    c    a  b  a  a  b  a  b  a  ...
               a  b  a  a  b  a    b    a  a  b  a  a  b
                                   ↑
                                 i = 7
```

Сада је и даље $p_i \neq s_j$, па се i поставља на нову вредност $h_7 = 4$. Неслагања се на овој позицији настављају и за $i = 4, 2, 1, 0$. Унутрашња петља завршена је без налажења узорка у тексту, па се у спољашњој петљи поставља $i = 1$, $j = 13$:

```
                         j = 13
                           ↓
...  a  b  a  a  b  a  c   a    b  a  a  b  a  b  a  a  b  a  a  b
                           a    b  a  a  b  a  b  a  a  b  a  a  b
                           ↑
                         i = 1
```

На овом месту алгоритам проналази узорак у тексту.

Други ефикасан алгоритам за налажење узорка у стрингу развили су Бојер и Мур 1977. (Boyer, Moore). Приказаћемо га укратко. Разлика између овог и алгоритма КМР је у томе што се узорак и текст пролазе у различитим смеровима. Најпре се упоређује p_m са s_m. Ако су та два знака једнака, наставља се са упоређивањем p_{m-1} са s_{m-1}, итд. У случају неслагања, прикупљене информације се користе као и у алгоритму КМР да се узорак помери удесно.

Ако је, на пример, $s_m =$"Z", а знак "Z" се не појављује у узорку, онда се узорак помера удесно за m позиција и следеће упоређивање је s_{2m} са p_m. Ако се "Z" појављује у P као нпр. p_i, унда се узорак помера $m - i$ позиција удесно. Одлука о томе колико узорак треба померити удесно постаје компликованија ако је претходно дошло до неколико слагања знакова узорка са одговарајућим знацима текста; при томе се узима у обзир и евентуални суфикс пронађеног дела узорка једнак префиксу узорка, као код алгоритма KMP. На овом месту нећемо се упуштати у детаље алгоритма. Занимљиво је да у тексту са великим алфабетом овај алгоритам најчешће извршава много мање од n упоређивања знакова, у просеку чак n/m упоређивања, што значи да су честа померања узорка за m позиција удесно.

5.7. Упоређивање низова

Проблему упоређивања низова тек се од недавно посвећује већа пажња. Основни разлог су примене у молекуларној биологији. Овде ћемо се бавити само једним проблемом — налажењем минималног броја едит операција које преводе један стринг у други. Основна техника која се у вези са тим користи је **динамичко програмирање** (видети одељак 4.10).

Нека су $A = a_1a_2 \dots a_n$ и $B = b_1b_2 \dots b_m$ два стринга, низа чији су елементи знакови из коначног алфабета. Циљ је трансформисати A у B, знак по знак. Дозвољене су три врсте елементарних трансформација (или **едит операција**), које имају цену 1:

- *уметање* — уметање знака;
- *брисање* — брисање знака, и
- *замена* — замена једног знака другим знаком.

Проблем. За задате стрингове $A = a_1a_2 \dots a_n$ и $B = b_1b_2 \dots b_m$ пронаћи низ едит операција најмање цене који A трансформише у B.

На пример, да се стринг *abbc* трансформише у стринг *babb*, може се прво обрисати *a* (добија се *bbc*), уметнути *a* између два *b* (добија се *babc*), и на крају заменити *c* са *b* — укупно три промене. Исти резултат може се постићи са само две промене: уметањем новог *b* на почетку (добија се *babbc*) и брисањем последњег *c*. Циљ је пронаћи трансформацију која се састоји од минималног броја едит операција (таква трансформација у општем случају није јединствена).

Најмањи број едит операција који трансформише стринг A у стринг B зваћемо едит растојањем $d(A, B)$ стрингова A и B. Функција $d(A, B)$ је метрика: очигледно је $d(A, A) = 0$ и $d(A, B) = d(B, A)$, за произвољне стрингове A и B (ако постоји низ едит операција дужине k који трансформише A у B, онда инверзних k операција, примењених обрнутим редоследом, трансформишу B у A). Ако су A, B и C произвољни стрингови, тада важи неједнакост троугла: $d(A, B) + d(B, C) \geq d(A, C)$. Заиста, постоји низ од $d(A, B)$ едит операција које преводе A у B, и постоји низ од $d(B, C)$ едит операција које преводе B у C. Посматране заједно, ових $d(A, B) + d(B, C)$ едит операција преводе A у C;

из чињенице да је $d(A, C)$ најмањи број едит операција које A преводе у C следи тражена неједнакост. Растојање $d(A, B)$ познато је као Левенштајново растојање (Levenstein).

Размотримо како може да изгледа минимални низ едит операција који A трансформише у B. На неке знакове $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ из A примењује се операција замене знацима $b_{j_1}, b_{j_2}, \ldots, b_{j_k}$ из B. Искључивањем ових знакова, стрингови A и B су подељени на $k+1$ (евентуално празних) група знакова, тако да се i-та (празна) група у B добија брисањем знакова из i-те групе у A, или се од i-те (празне) групе у A уметањем добија i-та група у B, $i = 1, 2, \ldots, k+1$. Због минималности су све едит операције у оквиру једне групе истог типа — или брисање или уметање, јер се брисање једног знака, па уметање другог знака на исто место (две едит операције са укопном ценом 2), може заменити само једном заменом, цене 1.

Узимајући у обзир ове чињенице, низ едит операција који је кандидат за оптимални, може се представити на следећи прегледан начин. Уведимо нови знак ϕ. Наспрам обрисаних знакова из A у B се умеће знак ϕ, а наспрам знакова у B, добијених уметањем, у A се умеће знак ϕ. Тиме су A, B продужени до низова $\bar{A}$, $\bar{B}$ једнаке дужине, а све едит операције сведене су на замену: брисању знака a из A одговара замена a из $\bar{A}$ са ϕ у $\bar{B}$, а уметању знака b у B одговара замена ϕ у $\bar{A}$ знаком b у $\bar{B}$. Знак ϕ сматра се различитим од свих осталих знакова, па је цена замене истим, односно различитим знаком и даље 0, односно 1.

Два горе наведена примера низова едит операција могу се дакле представити са

$$\begin{array}{cccccc} a & b & \phi & b & c & \\ \phi & b & a & b & b & \\ \hline 1+ & 0+ & 1+ & 0+ & 1 & =3, \end{array}$$

односно

$$\begin{array}{cccccc} \phi & a & b & b & c & \\ b & a & b & b & \phi & \\ \hline 1+ & 0+ & 0+ & 0+ & 1 & =2. \end{array}$$

Проблем трансформисања стрингова има такође примену у вези са упоређивањем фајлова и чувањем различитих верзија фајла. Претпоставимо да имамо текстуални фајл (нпр. програм) и други фајл који је модификација првог. Згодно је издвојити разлике два фајла. То могу бити различите верзије истог програма, па ако су верзије сличне и треба да буду архивиране, ефикасније је чувати само једну верзију и разлике уместо обе верзије програма. У таквим случајевима (овде знацима стринга одговарају линије у фајлу) дозвољене операције су само уметања и брисања. У другим ситуацијама могу се различитим едит операцијама доделити различите цене.

Могућих низова едит операција које A преводе у B, има много, па је на први поглед тешко међу њима наћи најбољи. За почетак, ограничићемо се на налажење едит растојања $d(A, B)$, односно најмањег броја едит операција које преводе A у B (а не и самог низа едит операција). Као и обично, покушаћемо

да проблем решимо индукцијом. За задати стринг A означимо са $A(i)$ његов префикс дужине i. Проблем је одредити растојање $d(A(n), B(m))$. За $n = 0$ потребно је празан префикс $A(0)$ трансформисати у стринг $B(m)$ дужине m. Очигледно је за то потребно најмање m операција уметања знакова; према томе, $d(A(0), B(m)) = m$ за $m \geq 0$. Слично, $d(A(n), B(0)) = n$ за $n \geq 0$, јер се најкраћи низ едит операција које стринг $A(n)$ дужине n преводи у празан стринг $B(0)$ састоји од n брисања. Да бисмо проблем решили индукцијом, потребно је да израчунавање растојања $d(A(i), B(j))$ између неких префикса сведемо на израчунавање растојања између неких "краћих" префикса. Овај циљ може се постићи ако размотримо која едит операција може бити последња у низу едит операција које преводе $A(i)$ у $B(j)$. Постоје три могућности:

(а) замена знака a_i (истим или различитим) знаком b_j, пошто је претходно префикс $A(i-1)$ трансформисан у $B(j-1)$,
(б) уметање знака b_j пошто је претходно префикс $A(i)$ трансформисан у $B(j-1)$;
(ц) брисање знака a_i, пошто је претходно префикс $A(i-1)$ трансформисан у $B(j)$,

што је илустровано на слици 24. Нека је

$$c(i,j) = \begin{cases} 0, & \text{за} \quad a_i = b_j \\ 1, & \text{за} \quad a_i \neq b_j \end{cases} .$$

Најмањи број едит операција које $A(i)$ преводе у $B(j)$ једнак најмањем од три израза $d(A(i-1), B(j-1)) + c(i,j)$, $d(A(i), B(j-1)) + 1$ и $d(A(i-1), B(j)) + 1$. Нека је C матрица чији су елементи растојања свих подстрингова A од свих подстрингова B, тј. $C[i,j] = d(A(i), B(j))$, $0 \leq i \leq n$, $0 \leq j \leq m$. Елементи ове матрице задовољавају диференцну једначину

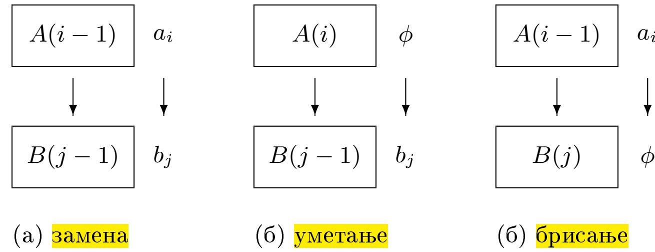
$$C[i,j] = \min\{C[i-1,j-1] + c(i,j), C[i,j-1] + 1, C[i-1,j] + 1\},$$

која њен произвољан елеменат изражава преко три друга: изнад, лево и горе–лево:

$$\begin{array}{cccccc} & & & j & \\ & & & \downarrow & \\ & & & \vdots & \\ & \cdots & C[i-1,j-1] & C[i-1,j] & \ldots \\ i\!\rightarrow & \cdots & C[i,j-1] & C[i,j] & \ldots \end{array}$$

Ова једнакост омогућује израчунавање свих елемената матрице C, пошто се знају њена 0-та колона и 0-та врста. Тиме је решен проблем израчунавања едит растојања стрингова A и B: $d(A,B) = C[n,m]$. Описани алгоритам је пример **динамичког програмирања**.

Реализација. За израчунавања се користи матрица C димензије $(n+1) \times (m+1)$, при чему први, односно други индекс узимају вредности из опсега $[0,n]$, односно $[0,m]$. Нека $M[i,j]$ означава последњу едит операцију која доводи до минималне вредности $C[i,j]$. Довољно је запамтити само ту последњу



(а) замена (б) уметање (б) брисање

Рис. 24. Низови едит операција које трансформишу A у B разврстани према последњој операцији.

едит операцију (брисање $A[i]$, уметање $B[j]$ или замена $A[i]$ са $B[j]$), јер се до комплетног низа едит операција долази пролазећи елементе матрице "уназад", полазећи од $M[n, m]$. За израчунавање $C[i, j]$, потребно је знати $C[i-1, j]$, $C[i, j-1]$ и $C[i-1, j-1]$. Последња промена $M[i, j]$ одређена је тиме која од три могућности даје минималну вредност за $C[i, j]$; избор није увек једнозначан! Алгоритам је приказан на слици 25.

```
Алгоритам Edit_rastojanje(A, n, B, m);
Улаз: A (стринг дужине n), и B (стринг дужине м).
Излаз: C (матрица растојања подстрингова A и B).
begin
    for i := 0 to n do C[i, 0] := i;
    for j := 0 to m do C[0, j] := j;
    for i := 1 to n do
        for j := 1 to m do
            x := C[i − 1, j] + 1;
            y := C[i, j − 1] + 1;
            if a_i = b_j then
                z := C[i − 1, j − 1]
            else
                z := C[i − 1, j − 1] + 1;
            C[i, j] := min(x, y, z)
             {изабрана едит операција памти се у M[i, j]}
end
```

Рис. 25. Алгоритам за налажење едит растојања два низа.

Пример 5.3. Размотримо рад алгоритма на горе наведеном примеру стрингова $A = abbc$ и $B = babb$. Матрица растојања подстрингова A и B, заједно са стрелицама које приказују могуће путеве добијања $C[i, j]$ ($\searrow$: замена $A[i]$ са

$B[j]$, $\rightarrow$: уметање $B[j]$ или $\downarrow$: брисање $A[i]$), дата је следећом табелом:

A \ B	0	1 *b*	2 *a*	3 *b*	4 *b*
0	**0**	$\rightarrow$ **1**	$\rightarrow$ 2	$\rightarrow$ 3	$\rightarrow$ 4
1 *a*	$\downarrow$ 1	$\searrow$ 1	$\searrow$ **1**	$\rightarrow$ 2	$\rightarrow$ 3
2 *b*	$\downarrow$ 2	$\searrow$ 1	$\searrow$ $\downarrow$ $\rightarrow$ 2	$\searrow$ **1**	$\searrow$ $\rightarrow$ 2
3 *b*	$\downarrow$ 3	$\searrow$ $\downarrow$ 2	$\searrow$ 2	$\searrow$ $\downarrow$ 2	$\searrow$ **1**
4 *c*	$\downarrow$ 4	3	$\searrow$ $\downarrow$ 3	$\searrow$ $\downarrow$ 3	$\downarrow$ **2**

У доњем десном углу табеле проналазимо $d(A, B) = 2$. До доњег десног угла долази се само само из поља изнад њега операцијом $\downarrow$. На сличан начин идући уназад, видимо да се до горњег левог угла табеле уназад може доћи само на један начин, низом операција (уназад) $\downarrow\searrow\searrow\searrow\rightarrow$. Према томе, у овом примеру постоји само један низ од две едит трансформације који преводи $A = abbc$ у $B = babb$.

Сложеност. Сваки елеменат матрице C израчунава се за константно време, па је временска сложеност алгоритма $O(nm)$. Проблем са овом верзијом алгоритма је у томе што је и његова просторна сложеност $O(nm)$. Ако се матрица C попуњава врсту по врсту, пошто врста зависи само од претходне, овај алгоритам може се прерадити тако да му просторна сложеност буде $O(m)$.

Примедба. Динамичко програмирање је корисно у ситуацији кад се решење проблема изражава преко неколико решења нешто мањих проблема. Коришћење табеле за смештање претходних резултата је уобичајено код динамичког програмирања. Табела се обично пролази неким редоследом (обично по врстама), па је сложеност алгоритма најмање квадратна. Због тога је приступ са динамичким програмирањем обично мање ефикасан од, на пример, приступа заснованог на декомпозицији.

5.8. Пробабилистички алгоритми

Алгоритми које смо до сада разматрали били су **детерминистички** — сваки наредни корак је унапред одређен. Кад се детерминистички алгоритам извршава два пута са истим улазом, начин извршавања је оба пута исти, као и добијени излази. Пробабилистички алгоритми су другачији. Они садрже кораке који зависе не само од улаза, него и од неких **случајних догађаја**. Постоји много варијација пробабилистичких алгоритама. Овде ћемо размотрити две од њих. Размотрићемо најпре један једноставан пример.

Претпоставимо да је дат скуп бројева $x_1, x_2, \ldots, x_n$ и да међу њима треба изабрати неки број из "горње половине", односно број који је већи или једнак од бар $n/2$ осталих. На пример, потребно је изабрати "доброг" студента, при

чему је критеријум просечна оцена. Једна могућност је узети највећи број (који је увек у горњој половини). Већ смо видели да је за одређивање максимума потребно $n-1$ упоређивање. Друга могућност је започети са извршавањем алгоритма за налажење максимума, и зауставити се кад се прође половина бројева. Број који је већи или једнак од једне половине бројева је сигурно у горњој половини. Алгоритам захтева око $n/2$ упоређивања. Може ли се овај посао обавити ефикасније? Није тешко показати да је немогуће гарантовати да број припада горњој половини ако је извршено мање од $n/2$ упоређивања. Према томе, описани алгоритам је оптималан.

Овај алгоритам је, међутим, оптималан само ако инсистирамо на *гаранцији*. У много случајева гаранција није неопходна, довољна је пристојна вероватноћа да је решење довољно тачно. На пример, код хеш таблица није могуће гарантовати да до колизија неће доћи, али постоји начин за решавање проблема изазваних појавом колизија (операције са хеш таблицама се такође могу сматрати пробабилистичким алгоритмима). Ако одустанемо од гаранције, онда постоји бољи алгоритам за налажење елемента из горње половине. Изаберимо на случајан начин два броја x_i и x_j из скупа, тако да је $i \neq j$. Претпоставимо да је $x_i \geq x_j$. Вероватноћа да случајно изабрани број из скупа припада горњој половини је бар $1/2$ (она ће бити већа од $1/2$ ако је више бројева једнако медијани). Вероватноћа да ни један од бројева x_i, x_j не припада горњој половини је највише $1/4$. Због $x_i \geq x_j$ ова вероватноћа једнака је вероватноћи да x_i не припада горњој половини. Према томе, вероватноћа да x_i припада горњој половини је бар $3/4$.

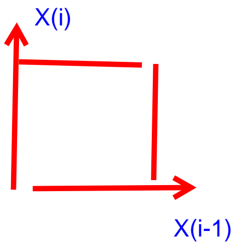
Вероватноћа $3/4$ да је добијени резултат тачан обично није довољна. Међутим, описани приступ може се уопштити. На случајан начин бирамо k бројева из скупа и бирамо највећи од њих. На исти начин као у специјалном случају, закључујемо да највећи од k елемената припада горњој половини са вероватноћом најмање $1-2^{-k}$ (он не припада горњој половини акко горњој половини не припада ни један од изабраних бројева, односно са вероватноћом највише 2^{-k}). На пример, ако је $k=10$, вероватноћа успеха је приближно 0.999, а за $k=20$ та вероватноћа је око 0.999999. Ако је пак $k=100$, онда је вероватноћа грешке за све практичне сврхе занемарљива. Вероватноћа грешке у програму, хардверске грешке, или чак земљотреса за време извршавања програма, већа је од $2^{-100} \simeq 10^{-30}$. Имамо дакле алгоритам који бира број из горње половине са произвољно великом вероватноћом, а који извршава највише 100 упоређивања независно од величине улаза. При томе се претпоставља да се случајни избор једног броја може извршити за константно време; генерисање случајних бројева биће размотрено у одељку 5.8.1.

За овакав алгоритам обично се каже да је **Монте Карло** алгоритам. Нетачан резултат може се добити са јако малом вероватноћом, али је време извршавања овог Монте Карло алгоритма боље него за најбољи детерминистички алгоритам. Други тип пробабилистичког алгоритма је онај који никад не даје погрешан резултат, али му време извршавања није гарантовано. Алгоритам се може извршити брзо, али се може извршавати и произвољно дуго.

Овај тип алгоритма, који са обично зове **Лас Вегас**, користан је ако му је *очекивано* време извршавања мало. У одељку 5.8.2 биће размотрен Лас Вегас алгоритам који решава један проблем бојења елемената скупа.

Идеја пробабилистичких алгоритама тесно је повезана са извођењем доказа. Коришћење вероватноће за доказивање комбинаторних тврђења је моћна техника. У основи, ако се докаже да неки објекат из скупа објеката има вероватноћу већу од нуле, онда је то индиректан доказ да постоји објекат са тим особинама. Ова идеја може се искористити за конструкцију пробабилистичког алгоритма. Претпоставимо да тражимо објекат са неким особинама, а знамо да ако генеришемо случајни објекат, он ће задовољавати жељени услов са вероватноћом већом од нуле (што је пробабилистички доказ да тражени објекат постоји). Ми затим пратимо пробабилистички доказ, генеришући случајне догађаје кад је потребно, и на крају налазимо жељени објекат се неком позитивном вероватноћом. Овај поступак може се понављати више пута, све до успешног проналажења жељеног објекта.

5.8.1. Случајни бројеви. Битан елеменат пробабилистичких алгоритама је генерисање случајних бројева. Потребно је имати ефикасне методе за решавање овог проблема. Детерминистичка процедура генерише бројеве на фиксирани начин, па тако добијени бројеви не могу бити *случајни* у правом смислу те речи. Ти бројеви су међусобно повезани на сасвим одређени начин. На срећу, то у пракси није велики проблем: довољно је користити тзв. **псеудослучајне бројеве**. Ти бројеви генеришу се детерминистичком процедуром (па дакле нису прави случајни бројеви), али процедура генерисања је довољно сложена, да друге апликације не "осећају" међузависности између тих бројева.



Овде нећемо детаљније разматрати добијање случајних бројева. Један од начина за добијање псеудослучајних бројева је **линеарни конгруентни метод**. Први корак је избор целог броја X_0 као првог члана низа, случајног броја изабраног на неки независан начин (на пример, текуће време у микросекундама). Остали бројеви израчунавају се на основу диференцне једначине $X_n = aX_{n-1} + b \pmod{m}$, где су a, b и m константе, које се морају опрезно изабрати. И поред пажљивог избора, овакви низови нису довољно квалитетни ("случајни"). Алтернатива су диференцне једначине вишег степена, односно вишег реда (са зависношћу од више претходних чланова). Добија се низ бројева из опсега од 0 до $m-1$. Ако су потребни случајни бројеви из опсега од 0 до 1, онда се чланови овог низа могу поделити са m.

5.8.2. Један проблем са бојењем. Нека је S скуп од n елемената, и нека је $S_1, S_2, \ldots S_k$ колекција његових различитих подскупова, од којих сваки садржи тачно r елемената, при чему је $k \leq 2^{r-2}$.

Проблем. Обојити сваки елеменат скупа S једном од две боје, црвеном или плавом, тако да сваки подскуп S_i садржи бар један плави и бар један црвени елеменат.

Бојење које задовољава тај услов зваћемо **исправним бојењем**. Испоставља се да под наведеним условима исправно бојење увек постоји. Једноставан пробабилистички алгоритам добија се преправком пробабилистичког доказа постојања таквог бојења:

Обојити сваки елеменат S случајно изабраном бојом, плавом или црвеном, независно од бојења осталих елемената.

Јасно је да овај алгоритам не даје увек исправно бојење. Израчунаћемо вероватноћу неуспеха. Вероватноћа да су сви елементи S_i обојени црвено је 2^{-r}, а вероватноћа да су сви обојени истом бојом (црвеном или плавом) је $2 \cdot 2^{-r} = 2^{1-r}$. Случајни догађај (подскуп скупа свих 2^n случајних бојења — елементарних догађаја) A: "неки од скупова S_i је неисправно обојен" је унија свих случајних догађаја A_i: "скуп S_i је неисправно обојен", за $i = 1, 2, \ldots, k$, па важи неједнакост

$$P(A) \leq \sum_{i=1}^{k} P(A_i) = \sum_{i=1}^{k} 2^{1-r} = k2^{1-r} \leq 2^{r-2}2^{1-r} = \frac{1}{2}.$$

Тиме је доказано да исправно бојење постоји (у противном би вероватноћа неисправног бојења била тачно 1). Поред тога, види се да је овај пробабилистички алгоритам добар. Исправност задатог бојења лако се проверава: проверавају се елементи сваког подскупа док се не пронађу два различито обојена елемента. Вероватноћа успешног бојења $p = 1 - P(A)$ је бар $1/2$. Ако се у једном покушају не добије неисправно бојење, поступак (бојење) се понавља. Очекивани број покушаја бојења је мањи или једнак од два. Заиста, вероватноћа да се исправно бојење пронађе у j-том покушају је $(1-p)^{j-1}p$, па је математичко очекивање броја покушаја

$$\sum_{j=1}^{\infty} j(1-p)^{j-1}p = \frac{1}{p} \leq 2.$$

Описани алгоритам бојења је очигледно Лас Вегас алгоритам, јер се бојења проверавају једно за другим, а се тражењем се завршава кад се наиђе на исправно бојење. Не постоји гаранција успешног бојења у било ком фиксираном броју покушаја, али је овај алгоритам у пракси ипак врло ефикасан.

5.9. Резиме

У овом поглављу разматрано је доста различитих области — претрага, сортирање, ранговске статистике, компресија података, упоређивање стрингова и пробабилистички алгоритми. У свакој области приказано је само неколико основних алгоритама. У пракси проблеми најчешће нису тако једноставни и није их тако лако формулисати. Због тога је неопходно издвојити само главне делове задатог проблема. Технике искоришћене у овом поглављу сличне су онима уведеним у поглављу 4. Индукција поново игра најважнију улогу.

Многи проблеми разматрани у овом поглављу имају директно решење које се може добити уз мало напора; примери су линеарна претрага и сортирање

избором. За мале величине улаза ова решења су најчешће не само довољно добра, него су и боља од савршенијих, компликованијих решења. Међутим, кад је величина улаза велика, важно је покушати са проналажењем бољих решења. Коришћење линеарне претраге и квадратних алгоритама за сортирање је уобичајено. На несрећу, ти и други неефикасни алгоритми сувише често се користе и за велике улазе.

Задаци

5.1. Играчи A и B играју игру погађања бројева: A замишља произвољан природни број n, а B покушава да погоди n постављањем што мањег броја питања облика "да ли је број x већи (мањи) од n?". Предложите ефикасну стратегију за играча B.

5.2. Играч A изабрао је на случајан начин, са равномерном расподелом вероватноћа, природан број k, $1 \le k \le 97$. Играч B погађа број k примењујући следећи алгоритам:
(1) на почетку је $a = 1$, $b = 97$;
(2) ако је $a = b$, тражени број је $k = a$;
(3) израчунати $m := \lfloor (a+b)/2 \rfloor$ и поставити питање "Да ли је $k > m$?"; ако јесте, $a := m+1$; ако није, $b := m$; у оба случаја наставити од (2).
Колики је очекивани број питања играча B?

5.3. Књига има 1999 страна. Задатак је бинарном претрагом пронаћи страницу x. Међутим, бинарна претрага није идеална: кад се претражује опсег страна од m до n, онда се због грешке само зна да је модуо разлике непарне стране на којој је књига отворена и $\lfloor (m+n)/2 \rfloor$ мањи или једнак од десетине броја страна у интервалу, односно од $(n-m+1)/10$. Колики број отварања књиге је довољан?

5.4. Претпоставимо да радимо са програмом који обрађује велике текстове. Програм обрађује текст са улаза као низ знакова, и даје неки излаз. У једном тренутку програм наилази на грешку, после које не може да настави са радом; при томе програм не саопштава каква је грешка у питању, ни где је до ње дошло. Другим речима, једино се види да се програм зауставио и одштампао поруку "Грешка". Претпоставимо да је грешка локална, тј. да до ње долази због неког кратког дела текста који се програму не свиђа, из непознатог разлога — грешка не зависи од контекста у коме се тај део текста налази. На који начи се може установити у ком делу текста долази до грешке?

5.5. Навести пример интерполационе претраге (тражење датог природног броја z у уређеном низу природних бројева $x_1 < x_2 < \cdots < x_n$) која користи $\Omega(n)$ упоређивања.

5.6. За свако n облика $n = 2^k$ навести пример низа $x_1, x_2, \ldots, x_n$ код кога се при сортирању обједињавањем (слика 7) извршава максимални број упоређивања.

5.7. Упоредите сортирање обједињавањем са проблемом одређивања максимума скупа правоугаоних функција из одељка 4.6. Може ли се решење једног од тих проблема искористити као "црна кутија" за решавање другог?

5.8. Формулисати инваријанту петље у основној петљи алгоритма *Razdvajanje* (слика 8), и доказати коректност алгоритма.

5.9. Конструисати улаз на коме сортирање раздвајањем користи $\Omega(n)$ упоређивања, ако се за пивот увек бира медијана првог, последњег и средњег елемента низа.

5.10. Конструисати алгоритам за налажење максималног и минималног елемента у скупу, заснован на декомпозицији. За $n = 2^k$ алгоритам треба да користи највише $3n/2$ упоређивања.

5.11. Кодирати следећу реченицу оптималним бинарним префиксним кодом: "На врх брда врба мрда".

5.12. Одредити едит растојање стрингова *aabccbba* и *baacbabacc* и наћи бар један низ едит операција минималне дужине који прву реч трансформише у другу.

5.13. Дат је низ природних бројева $A[1], A[2], \ldots, A[n]$ такав да је $|A[i] - A[i+1]| \leq 1$ за свако i, $1 \leq i \leq n$. Нека је $A[1] = x$, $A[n] = y$, $x < y$ и нека је дат број z, $x \leq z \leq y$. Конструисати ефикасан алгоритам за налажење индекса j таквог да је $A[j] = z$. Колики максимални број упоређивања са z се извршава?

5.14. Коришћењем стабла одлучивања извести доњу границу за број корака (у најгорем случају) у алгоритму који решава проблем из задатка 5.13, и упоредити га са сложеношћу конструисаног алгоритма.

5.15. Дат је скуп S од n реалних бројева. Конструисати алгоритам сложености $O(n)$ за налажење неког броја који није у S. Доказати да је $\Omega(n)$ доња граница за број корака потребних да се реши овај проблем.

5.16. Дат је скуп S од n реалних бројева, и реални број x. (а) Конструисати алгоритам сложености $O(n \log n)$ за утврђивање да ли у S постоје два елемента којима је збир једнак x. (б) Ако су елементи S дати у растућем редоследу, конструисати алгоритам који решава исти проблем, а има сложеност $O(n)$.

5.17. Дата су два скупа реалних бројева S_1 и S_2 и реални број x. Установити да ли постоје реални бројеви $y_1 \in S_1$ и $y_2 \in S_2$ такви да им је збир једнак x. Временска сложеност алгоритма треба да буде $O(n \log n)$, где је n укупан број елемената у оба скупа.

5.18. Конструисати алгоритам сложености $O(n \log n)$ за налажење уније два скупа од по највише n елемената. Излаз треба да буде низ различитих елемената уније (ни један елеменат се не сме у низу појавити више него једном).

5.19. Задат је низ реалних бројева $x_1, x_2, \ldots, x_n$, при чему је n парно. Конструисати алгоритам за разлагање низа у $n/2$ парова на следећи начин. Нека су са $s_1, s_2, \ldots, s_{n/2}$ означене суме бројева у паровима, и нека је $s_{max} = \max\{s_1, s_2, \ldots, s_{n/2}\}$. Парове треба формирати тако да s_{max} има најмању могућу вредност.

5.20. Дат је низ бројева $x_1, x_2, \ldots, x_n$ и низ различитих бројева $a_1, a_2, \ldots, a_n$ из скупа $\{1, 2, \ldots, n\}$ (пермутација тог скупа). Конструисати алгоритам сложености $O(n \log n)$ који уређује први низ у складу са редоследом одређеним другим низом, тј. на позицији a_i у првом низу треба да се нађе број x_i, $i = 1, 2, \ldots, n$. Просторна сложеност алгоритма треба да буде $O(1)$.

5.21. Дато је d растуће уређених низова са укупно n елемената. Конструисати алгоритам сложености $O(n \log d)$ за обједињавање свих ових низова у један растуће уређени низ.

5.22. Дат је низ од n природних бројева са више понављања елемената, тако да је број различитих елемената у низу $O(\log n)$. (а) Конструисати алгоритам за сортирање оваквих низова, у коме се извршава највише $O(n \log \log n)$ упоређивања бројева. (б) Због чега је сложеност овог алгоритма мања од доње границе $\Omega(n \log n)$ за сортирање?

5.23. Бинарно стабло са n чворова је уравнотежено ако одговара имплицитно представљеном стаблу помоћу низа дужине n. Доказати да је збир $f(n)$ висина свих чворова уравнотеженог бинарног стабла са n чворова мањи или једнак од $n - 1$. За која стабла се у овој неједнакости достиже једнакост?

5.24. Сума висина свих чворова у хипу представљеном имплицитно (видети одељак 5.3.5) може се израчунати директно, користећи чињеницу да је висина чвора на позицији i у вектору дужине n једнака $\lfloor \log_2(n/i) \rfloor$. Одредити суму висина чворова на овај начин.

5.25. Дат је хип са n елемената (са највећим елементом у корену), и реални број x. Конструисати алгоритам који утврђује да ли је k-ти највећи елеменат хипа мањи или једнак од x. Временска сложеност алгоритма треба да буде $O(k)$ (независно од величине хипа), а

може се користити меморијски простор величине $O(k)$ (није потребно *одредити* k-ти највећи елеменат, него само установити његов однос са x).

5.26. Нека је A алгоритам који одређује k-ти највећи од n елемената $x_1, x_2, \ldots, x_n$ низом упоређивања. Показати да A прикупља довољно информација да се може установити који су елементи већи, а који мањи од k-тог највећег (другим речима, скуп се може разложити на веће и мање од k-тог највећег, без нових упоређивања)

5.27. Циљ је одредити k-ти најмањи елеменат слично као у задатку 5.26, али овог пута желимо да минимизирамо време извршавања, и да при томе користимо врло мали простор (не обавезно минимални). Улаз је низ бројева $x_1, x_2, \ldots, x_n$ који се добијају један по један. Конструисати алгоритам средње временске сложености $O(n)$ за израчунавање k-тог најмањег елемента, који користи само $O(k)$ меморијских локација. Вредност k унапред је позната (па се може резервисати довољан меморијски простор), али се вредност n не зна до појаве последњег елемента низа.

5.28. Нека су A и B два скупа од по n елемената, при чему се A чува у рачунару P, а B у рачунару Q. Рачунари могу да комуницирају размењујући поруке, а могу и да изврше било какво локално израчунавање. Конструисати алгоритам који одређује n-ти најмањи елеменат скупа $A \cup B$ (тј. медијану). Може се, због једноставности, претпоставити да је $A \cap B = \emptyset$. Циљ је минимизирати број порука, при чему порука може да садржи један елеменат или један цели број. Колики је број порука потребан у најгорем случају?

5.29. Коришћењем теоријско-информационе границе показати да је $\Omega(\log n)$ доња граница броја упоређивања при проналажењу вредности i такве да је $x_i = i$ у низу $x_1, x_2, \ldots, x_n$, или утврђивању да таква вредност не постоји (проблем је разматран у одељку 5.2).

5.30. Тражење узорка у тексту у реалном времену. Претпоставимо да се узорак уноси (укуцава) знак по знак малом брзином, а текст је већ дат. Очекује се да тражење у сваком тренутку напредује колико је највише могуће, не чекајући да буде унесен комплетан узорак. Другим речима, желимо да се при уношењу k-тог знака узорка већ налазимо на првој позицији у тексту одакле се текст поклапа са првих $k-1$ знакова узорка. Како треба променити алгоритам КМР да би се постигао овај циљ?

5.31. Модификовати алгоритам КМР тако да пронађе највећи префикс узорка P који као подстринг постоји у тексту S (другим речима, у тексту S треба пронаћи највећи могући префикс узорка P).

5.32. Највећи заједнички подниз (LCS, скраћеница од largest common subsequence) два низа T и P је најдужи низ L који је подниз оба низа T и P. **Најмањи заједнички надниз** (SCS, скраћеница од smallest common supersequence) два низа T и P је најкраћи низ L такав да су T и P његови поднизови.

(а) Конструисати ефикасне алгоритме за одређивање LCS и SCS два дата низа.

(б) Нека је $d(T, P)$ едит растојање између T и P, при чему замене нису дозвољене (дозвољена су само уметања и брисања знакова). Доказати да је $d(T, P) = |SCS(T, P)| - |LCS(T, P)|$, где су $|SCS(T, P)|$, односно $|LCS(T, P)|$ дужине SCS, односно LCS низова T и P.

5.33. Уопштити алгоритам за налажење едит растојања из одељка 5.6 на случај кад се уметања знакова на почетку или на крају краћег низа не рачунају. Другим речима, рачуна се само едит растојање B и било ког ("најближег") подниза A.

5.34. Проблем упоређивања низова може се уопштити на три (или више) низова на следећи начин. Дата су три низа $A = a_1a_2 \ldots a_m$, $B = b_1b_2 \ldots b_n$ и $C = c_1c_2 \ldots c_p$. Потребно је одредити најмањи скуп едит операција који сва три низа трансформише у један исти низ.

5.35. Нека су $A = a_1a_2 \ldots a_n$ и $B = b_1b_2 \ldots b_m$ два низа знакова. Означимо са $A\{i\}$ низ $a_ia_{i+1} \ldots a_n$ (тј. i-ти суфикс A). Нека је d_i едит растојање низова B и $A\{i\}$. Конструисати алгоритам сложености $O(n^2)$ за налажење најмање вредности d_i, $i = 1, 2, \ldots, n$.

5.36. На располагању нам је процедура, која за произвољно k, $1 \leq k \leq n$, генерише случајне бројеве из опсега од 1 до k са униформном расподелом вероватноће (РВ). Задатак је конструисати алгоритам за генерисање случајне пермутације бројева $\{1, 2, \ldots, n\}$ са униформном РВ.

5.37. Нека је E задати низ бројева $x_1, x_2, \ldots, x_n$. **Вишеструкост** броја x у E је број појављивања x у E. Одредити елеменат E вишеструкости веће од $n/2$ ("преовлађујући елеменат") или установити да такав елеменат не постоји.

5.38. Нека је дат низ од n елемената. Конструисати алгоритам сложености $O(n)$ за проналажење елемента који се у низу појављује више од $n/4$ пута.

Глава 6

Графовски алгоритми

6.1. Увод

У претходном поглављу разматрали смо алгоритме за рад са низовима и скуповима објеката. Анализирани су односи уређења, вишеструкости, преклапања и слични. У овом поглављу разматраћемо компликованије односе између објеката. За моделирање тих односа користићемо графове. Графови могу да моделирају много ситуација, и користе се у различитим областима, од археологије до социјалне психологије. Приказаћемо више важних основних алгоритама за обраду графова и за израчунавање неких њихових карактеристика.

Погледајмо најпре неке примере моделирања помоћу графова.

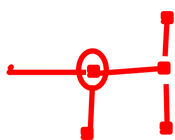
(1) Налажење доброг пута до неког одредишта у граду је графовски проблем. Улице одговарају гранама (усмерене гране у случају једносмерних улица), а раскрснице чворовима. Сваком чвору и свакој грани (делу улице) може се придружити време потребно за пролазак, па се проблем своди на налажење "најбржег" пута између два чвора.

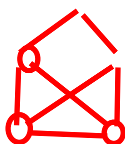
(2) Процес извршавања неког програма може се поделити на стања. Из сваког стања може се наставити на неколико начина. Нека стања сматрају се непожељним. Проблем утврђивања која стања могу довести у непожељна стања је графовски проблем у коме стањима одговарају чворови, а гране указују на могуће преласке из једног у друго стање.

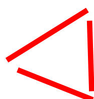
(3) Прављење распореда часова на факултету може се посматрати као графовски проблем. Чворови одговарају групама, а две групе су повезане ако постоји студент који намерава да их обе похађа, или ако имају заједничког предавача. Проблем је како направити распоред по групама тако да се минимизирају конфликти. То је тежак проблем, за који није лако наћи добра решења.

Представљање графова разматрано је у одељку 3.8. Најчешће ћемо користити представљање листом повезаности, која је ефикаснија за ретке графове (односно графове са релативно малим бројем грана). Увешћемо најпре неопходне појмове. Граф $G = (V, E)$ састоји се од скупа V **чворова**, и скупа E **грана**. Свака грана одговара пару различитих чворова (понекад су дозвољене петље, односно гране које воде од чвора ка њему самом; ми ћемо претпостављати да оне нису дозвољене). Граф може бити **усмерен** или **усмерен**. Гране

усмереног графа су уређени парови чворова (који се зову крајеви гране); редослед два чвора које повезује грана је битан. Гране усмереног графа цртају се као стрелице усмерене од једног чвора (почетка) ка другом чвору (крају). Гране неусмереног графа су неуређени парови: оне се цртају као обичне дужи (линије). **Степен** $d(v)$ чвора v је број грана суседних чвору v (односно број грана које v повезују са неким другим чвором). У усмереном графу разликујемо **улазни степен**, број грана за које је чвор v крај, и **излазни степен**, број грана за које је чвор v почетак.







Пут од v_1 до v_k је низ чворова $v_1, v_2, \ldots, v_k$ повезаних гранама (v_1, v_2), (v_2, v_3), ..., (v_{k-1}, v_k); ове гране се обично такође сматрају делом пута. Пут је **прост**, ако се сваки чвор у њему појављује само једном. За чвор u се каже да је **достижан** из чвора v ако постоји пут (усмерен, односно неусмерен, зависно од графа) од v до u. По дефиницији је чвор v достижан из v. **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 појављује два пута. **Неусмерени облик** усмереног графа $G = (V, E)$ је исти граф, без смерова на гранама. За граф се каже да је **повезан** ако (у његовом неусмереном облику) постоји пут између произвољна два чвора. **Шума** је граф који (у свом неусмереном облику) не садржи циклусе. **Стабло** или **дрво** је повезана шума. **Коренско стабло** је усмерено стабло са једним посебно издвојеним чвором, који се зове **корен** при чему су све гране усмерене од корена.

Граф $H = (U, F)$ је **подграф** графа $G = (V, E)$ ако је $U \subset V$ и $F \subset E$. **Повезујуће стабло** неусмереног графа G је његов подграф који је стабло и садржи све чворове G. **Повезујућа шума** неусмереног графа G је његов подграф који је шума и садржи све чворове G. **Индуковани подграф** графа $G = (V, E)$ је његов подграф $H = (U, F)$ такав да је $U \subset V$ и F садржи све гране из E чија су оба краја у U. Ако граф $G = (V, E)$ није повезан, онда се он може на јединствен начин разложити у скуп повезаних подграфова, који се зову **компоненте повезаности** графа G. Скуп чворова компоненте повезаности је класа еквиваленције у односу на релацију достижности за чворове: компоненту повезаности G којој припада чвор v чине сви чворови достижни из v. Компонента повезаности графа G је максимални повезани подграф G (односно такав подграф који није прави подграф ни једног другог повезаног подграфа G). **Бипартитни граф** је граф чији се чворови могу поделити на два дисјунктна подскупа тако да у графу постоје само гране између чворова из различитих подскупова. **Тежински граф** је граф чијим су гранама придружени реални бројеви (тежине, цене, дужине, енг. weight, cost, length).

Многе дефиниције за усмерене и неусмерене графове су сличне, изузев неких очигледних разлика. На пример, усмерени и неусмерени путеви дефинишу се на исти начин, сем што се код усмерених графова специфицирају и смерови грана. Значење таквих термина зависи од контекста; ако се, на пример, ради о путевима у усмереном графу, онда се мисли на усмерене путеве.

Размотрићемо најпре једноставан проблем, који се сматра првим проблемом теорије графова — обилазак Кенингсбершких мостова. Затим ћемо се

бавити обиласцима графова, уређењем чворова графа, налажењем најкраћих путева у графу и другим проблемима. У поглављу 8 разматрају се везе графовских и алгоритама са матрицама, па је приказано и неколико графовских алгоритама.

6.2. Ојлерови графови

Појам Ојлерових графова у вези је са, како се сматра, првим решеним проблемом теорије графова. Швајцарски математичар Леонард Ојлер наишао је на следећи задатак 1736. године. Град Кенигсберг, данас Калињинград, лежи на обалама и на два острва на реци Прегел, као што је приказано на слици 1. Град је био повезан са 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. Решење је добијено уопштавањем проблема. ”Граф” на истој слици, бар што се овог проблема тиче, еквивалентан је са планом града (наводници су употребљени јер овај ”граф” има вишеструке гране, па строго гледано по дефиницији и није граф; то је тзв. мултиграф). Проблем се може еквивалентно формулисати као следећи проблем из теорије графова: да ли је могуће у повезаном графу пронаћи циклус, који сваку грану садржи тачно једном (*Ојлеров циклус*). Или: да ли је могуће нацртати граф са слике 1 не дижући оловку са папира, тако да оловка свој пут заврши на месту са кога је и кренула. Ојлер је решио проблем, доказавши да је овакав обилазак могућ ако и само ако је граф повезан и сви његови чворови имају паран степен. Такви графови зову се **Ојлерови графови**. Пошто ”граф” на слици 1 има чворове непарног степена, закључујемо да проблем Кенигсбершких мостова нема решење. Доказ теореме индукцијом, који следи, даје ефикасан алгоритам за налажење Ојлеровог циклуса у графу.

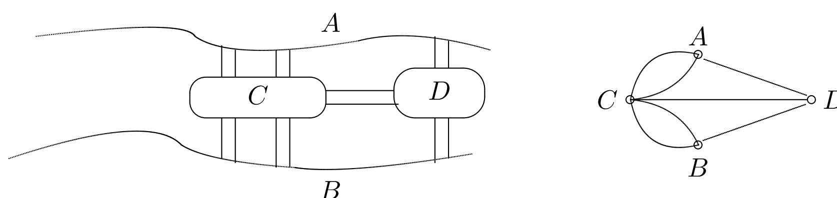


Рис. 1. Проблем Кенигсбершких мостова, и одговарајући граф.

Проблем. У задатом неусмереном повезаном графу $G = (V, E)$ чији сви чворови имају паран степен, пронаћи затворени пут P, такав да се у њему свака грана из E појављује тачно једном.

Лако је показати да ако у графу постоји Ојлеров циклус, онда сви чворови графа морају имати паран степен. За време обиласка затвореног 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 услов довољан, морамо најпре да изаберемо параметар по коме ће бити изведена индукција. Тај избор треба да омогући смањивање проблема, без његове промене. Ако уклонимо чвор из графа, степени чворова у добијеном графу нису више сви парни. Требало би да уклонимо такав скуп грана S, да за сваки чвор v графа број грана из S суседних са v буде паран (макар и 0). Произвољан циклус задовољава овај услов, па се поставља питање да ли Ојлеров граф увек садржи циклус. Претпоставимо да смо започели обилазак графа из произвољног чвора v произвољним редоследом. Сигурно је да ће се обилазак раније или касније завршити у чвору v, 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.

Сада можемо да формулишемо индуктивну хипотезу и докажемо теорему.

Индуктивна хипотеза. Повезани граф са $< m$ грана чији сви чворови имају паран степен, садржи Ојлеров циклус, који се може пронаћи.

Посматрајмо граф $G = (V, E)$ са m грана. Нека је P неки циклус у G, и нека је G' граф добијен уклањањем грана које чине P из графа G. Степени свих чворова у 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, видети пример на слици 2. Нека су G'_1, G'_2, ..., G'_k компоненте повезаности графа G'. У свакој компоненти степени свих чворова су парни. Поред тога, број грана у свакој компоненти је мањи од m (њихов укупан број грана мањи је од m). Према томе, индуктивна хипотеза може се применити на све компоненте: у свакој компоненти G'_i постоји Ојлеров циклус P'_i, и ми знамо да га пронађемо. Потребно је сада све ове циклусе објединити у један Ојлеров циклус за граф G. Полазимо из било ког чвора циклуса P ("магистралног пута") све док не дођемо до неког чвора v_j који припада компоненти G'_j. Тада обилазимо компоненту G'_j циклусом P_j ("локалним путем") и враћамо се у чвор v_j. Настављајући на тај начин, обилазећи циклусе компоненти у тренутку наиласка на њих, на крају ћемо се вратити у полазну тачку. У том тренуку све гране графа G пројђене су тачно једном, што значи да је конструисан Ојлеров циклус. Алгоритам ипак није сасвим комплетан: потребан нам је ефикасан метод за издвајање компоненти повезаности графа, односно за обилазак графа. Оба ова проблема биће разматрана у следећем одељку.

6.3. Обиласци графова

Први проблем на који се наилази при конструкцији било ког алгоритма за обраду графа је како прегледати улаз. У претходном поглављу тај проблем био је тривијалан због једнодимензионалности улаза — низови и скупови могу се лако прегледати линеарним редоследом. Прегледање графа, односно његов

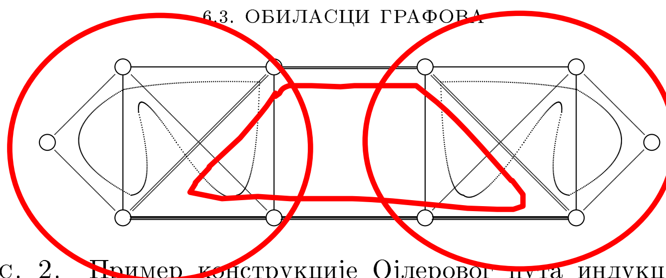


Рис. 2. Пример конструкције Ојлеровог пута индукцијом. Пуном линијом извучене су гране помоћног циклуса. Избацивањем грана овог циклуса из графа, добија се граф са две компоненте повезаности.

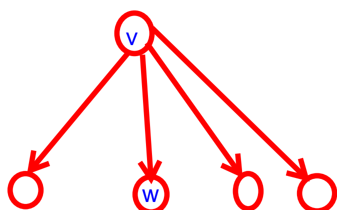
обилазак, није тривијалан проблем. Постоје два основна алгоритма за обилазак графа: **претрага у дубину** и **претрага у ширину**. Више алгоритама из овог поглавља заснива се на једној од ове две технике.

6.3.1. Претрага у дубину. Претрага у дубину је практично иста за неусмерене и усмерене графове. Међутим, пошто желимо да испитамо неке особине графова, које нису исте за неусмерене и усмерене графове, разматрање ће бити подељено на два дела.

6.3.1.1. *Неусмерени графови*. Претпоставимо да граф $G = (V, E)$ одговара уметничкој галерији, која се састоји од низа ходника са сликама на зидовима. Гране G одговарају ходницима, а чворови одговарају пресецима ходника. Ми хоћемо да обиђемо галерију и видимо све слике. Претпоставка је да у току проласка кроз ходник у било ком смеру видимо слике на оба његова зида. Ако је граф Ојлеров, могуће је обићи галерију посећујући сваки ходник тачно једном. Овде се не претпоставља да је граф Ојлеров, а дозвољава се пролазак исте гране више него једном (као што ће се видети, свака грана биће прегледана тачно два пута). Идеја на којој се заснива претрага у дубину је следећа. Пролазимо кроз галерију трудећи се да уђемо у нови ходник увек кад је то могуће. Када први пут уђемо у неку раскрсницу, остављамо каменчић и настављамо кроз неки други ходник (изузев ако је то ходник – слепа улица). Кад дођемо до раскрснице у којој већ постоји каменчић, враћамо се истим путем назад, и покушавамо кроз неки други ходник. Ако су сви ходници који воде из раскрснице прегледани, онда уклањамо каменчић из раскрснице и враћамо се ходником кроз који смо први пут ушли у раскрсницу. Ову раскрсницу више нећемо посећивати (каменчић се уклања због одржавања реда у галерији; уклањање није суштински део алгоритма). Увек се трудимо да испитујемо нове ходнике; кроз ходник којим смо први пут ушли у раскрсницу враћамо се тек кад смо прошли све ходнике из раскрснице. Овај приступ зове се **претрага у дубину** (DFS, скраћеница од depth–first–search), што је у вези са тенденцијом да се посећују увек нови ходници (идући све дубље у галерију). Основни разлог употребљивости претраге у дубину лежи у њеној једноставности и прилагодљивости рекурзивним алгоритмима.

После описа претраге у дубину заснованог на обиласку галерије и остављању каменчића, размотримо исти проблем за неусмерени граф задат листом повезаности. Обилазак започиње из произвољног задатог чвора r, **корена претраге у дубину**. Корен се **означава** као посећен. Затим се бира произвољни неозначени чвор r_1, суседан са r, па се из чвора r_1 рекурзивно стартује претрага у дубину. Из неког нивоа рекурзије излази се кад се наиђе на чвор v коме су сви суседи (ако их има) већ означени. Ако су у тренутку завршетка претраге из r_1, сви суседи чвора r означени, онда се претрага за чвор r завршава. У противном, бира се следећи произвољни неозначени сусед r_2 чвора r, извршава се претрага полазећи од r_2, итд.

Претрага графа увек се врши са неким циљем. Да би се различите апликације уклопиле у претрагу у дубину, посети чвора или гране придружују се две врсте обраде, **улазна обрада** и **излазну обраду**. Улазна обрада врши се у тренутку означавања чвора. Излазна обрада врши се после повратка неком граном, или кад се открије да нека грана води већ означеном чвору. Улазна и излазна обрада зависе од конкретне примене DFS. Ови појмови омогућиће нам решавање неколико проблема једноставним дефинисањем улазне и излазне обраде. Алгоритам претраге у дубину дат је на слици 3. Полазни чвор за рекурзивну претрагу је v. Због једноставности се за почетак претпоставља да је граф повезан. Пример претраге графа у дубину дат је на слици 4, при чему бројеви уз чворове показују редослед обиласка.

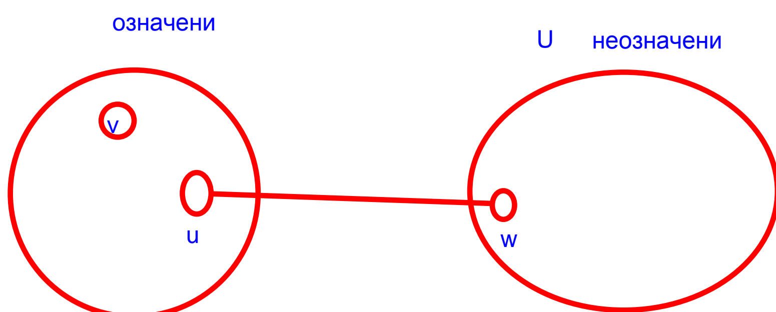


Алгоритам DFS(G, v);
Улаз: $G = (V, E)$ (неусмерени повезани граф) и v (чвор графа G).
Излаз: зависи од примене.
begin
  означи v;
  изврши улазну обраду на v; {улазна обрада зависи од примене DFS}
  for све гране (v, w) **do**
    if w је неозначен **then** $DFS(G, w)$;
    изврши излазну обраду за (v, w)
     {излазна обрада зависи од примене DFS}
     {она се понекад врши само за гране ка новоозначеним чворовима}
end

Рис. 3. Претрага у дубину.

Лема 6.1. *Ако је граф G повезан, онда ће алгоритмом претраге у дубину сви његови чворови бити означени, а све његове гране биће у току извршавања алгоритма прегледане бар по једном.*

Доказательство. Претпоставимо супротно, и означимо са U скуп неозначених чворова заосталих после извршавања алгоритма. Пошто је G повезан,



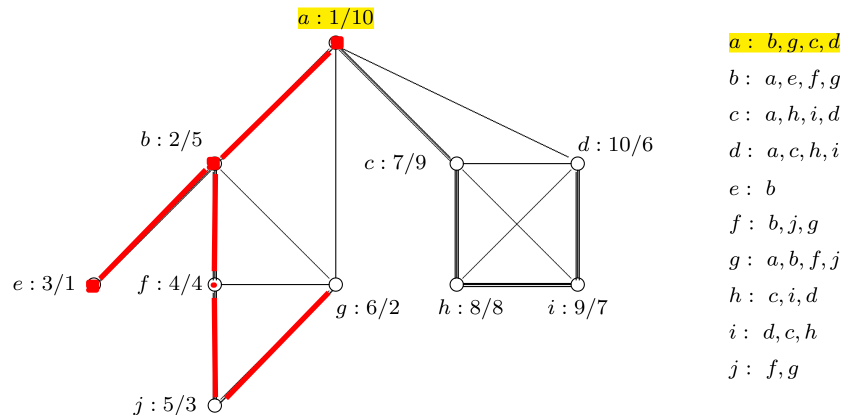


Рис. 4. Пример претраге у дубину. Два броја уз чвор једнака су његовим редним бројевима при долазној, односно одлазној DFS нумерацији.

бар један чвор из U мора бити повезан граном са неким означеним чвором. Међутим, овако нешто је немогуће, јер кад год се посети чвор, морају бити посећени (па дакле и означени) сви његови неозначени суседи. Пошто су сви чворови посећени, а кад се чвор посети, онда се прегледају све гране које воде из њега, закључујемо да су и све гране графа прегледане. □

За неповезане графове се алгоритам DFS мора променити. Ако су сви чворови означени после првог покретања описаног алгоритма, онда је граф повезан, и обилазак је завршен. У противном, може се покренути нова претрага у дубину полазећи од произвољног неозначеног чвора, итд. Према томе, DFS се може искористити да би се установило да ли је граф повезан, односно за проналажење свих његових компоненти повезаности. Одговарајући алгоритам дат је на слици 5. Ми ћемо најчешће разматрати само повезане графове, јер се у општем случају проблем своди на посебну обраду сваке компоненте повезаности. Дакле, користићемо DFS у облику са слике 3, не спомињући експлицитно да се претрага мора више пута покретати као на слици 5.

Сложеност. Лако је уверити се да се свака грана прегледа тачно два пута, по једном са сваког краја. Према томе, временска сложеност је пропорционална броју грана. С друге стране, број рекурзивних покретања алгоритма је $|V|$, па се сложеност алгоритма може описати изразом $O(|V| + |E|)$.

6.3.1.2. *Конструкција* DFS *стабла.* Приказаћемо сада две једноставне примене DFS — нумерацију чворова графа **DFS бројевима**, и формирање специјалног повезујућег стабла, такозваног **DFS стабла**. DFS бројеви и DFS стабло имају посебне особине, које су корисне у многим алгоритмима. Чак и ако се стабло не формира експлицитно, многе алгоритме је лакше разумети разматрајући DFS стабло. Да би се описали ови алгоритми, довољно је задати било

```
Алгоритам Komponente_povezanosti(G);
Улаз: G = (V, E) (неусмерени граф).
Излаз: v.Komp за сваки чвор v добија вредност једнаку редном броју компоненте
    повезаности која садржи v.
begin
    Rb_komp := 1;
    while постоји неозначен чвор v do
        DFS(G, v);
        {са следећом улазном обрадом:}
            {v.Komp := Rb_komp;}
        Rb_komp := Rb_komp + 1;
end
```

Рис. 5. Алгоритам за налажење компоненти повезаности графа.

улазну, било излазну обраду. Постоје две варијанте DFS нумерације: чворови се могу нумерисати према редоследу означавања (**долазна** DFS **нумерација**, или према редоследу напуштања (**одлазна** DFS **нумерација**. Алгоритам за добијање обе DFS нумерације чворова графа дат је на слици 6, а пример графа са чворовима нумерисаним на два начина приказан је на слици 4. Алгоритам за формирање DFS стабла приказан је на слици 7. Наравно, ова два алгоритма не морају се извршавати одвојено.

```
Алгоритам DFS_numeracija(G, v);
Улаз: G = (V, E) (неусмерени граф) и v (чвор G).
Излаз: за сваки чвор v рачунају се његови редни бројеви v.Pre и v.Post
    при долазној, односно одлазној нумерацији.
Иницијализација DFS_pre := 1; DFS_post := 1;
Покренути DFS са следећом улазном и излазном обрадом:
улазна обрада:
    v.Pre := DFS_pre;
    DFS_pre := DFS_pre + 1;
излазна обрада:
    if w је последњи на листи суседа v then
        v.Post := DFS_post;
        DFS_post := DFS_post + 1;
```

Рис. 6. DFS нумерација чворова графа.

Чвор v зове се **предак** чвора w у стаблу T са кореном r ако је v на јединственом путу од w до r у T. Ако је v предак w, онда је w **потомак** v. DFS стабло обухвата све чворове графа G. Редослед синова сваког чвора у стаблу

Алгоритам DFS_stablo(G, v);
Улаз: $G = (V, E)$ (неусмерени граф) и v (чвор G).
Излаз: T (DFS стабло графа G; на почетку је T празно).
Покренути DFS са следећом излазном обрадом:
излазна обрада:
  if w је био неозначен **then** додати грану (v, w) у T;
  {горњу наредбу треба додати у **if** наредбу алгоритма DFS}

Рис. 7. Формирање DFS стабла графа.

задат је листом повезаности која специфицира граф G, па се за свака два међу њима 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 један испод другог (односно у релацији предак–потомак), видети слику 8. За њих тада постоји јединствени заједнички предак w у DFS стаблу, као и синови u' и v' чвора w такви да је u' предак u и v' предак v. Кажемо да је u лево од v ако и само ако је u' лево од v'. Јасна је геометријска интерпретација ове релације: можемо да замислимо да се DFS стабло исцртава наниже док се прелази у нове – неозначене чворове (кораци у дубину), односно слева удесно приликом додавања нових грана после повратка у већ означене чворове. Долазна DFS нумерација чворова непосредно је повезана са уведеном релацијом. Нека су $u.Pre$, $v.Pre$ редни бројеви u, v према долазној DFS нумерацији. Ако је чвор u лево од чвора v, онда је он приликом претраге достигнут пре чвора v, па је $u.Pre < v.Pre$. Обрнуто не важи увек: ако је $u.Pre < v.Pre$, онда је u лево од v, или је u предак v у DFS стаблу (тј. изнад v).

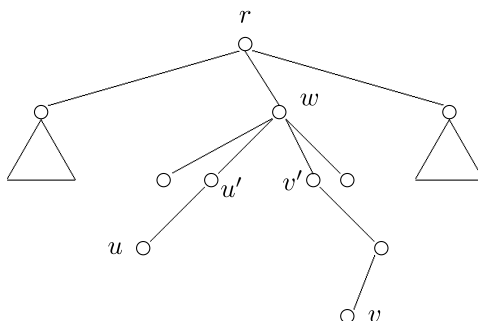


Рис. 8. Уз дефиницију релације "лево–десно" на скупу чворова DFS стабла.

Лема 6.2 (Основна особина DFS стабла неусмереног графа)**.** *Нека је* $G = (V, E)$ *повезан неусмерен граф, и нека је* $T = (V, F)$ DFS *стабло графа* G *добијено алгоритмом* DFS_stablo. *Свака грана* $e \in E$ *припада* T *(тј.* $e \in F$*) или спаја два чвора графа* G*, од којих је један предак другог у* T.

Доказательство. Нека је (v, u) грана у G, и претпоставимо да је у току DFS v посећен пре u. После означавања v, у петљи се рекурзивно покреће DFS из сваког неозначеног суседа v. У тренутку кад дође ред на суседа u, ако је u означен, онда је u потомак v у T, а у противном, се из u стартује DFS, па u постаје син v. □

Тврђење леме може се преформулисати на следећи начин: гране графа не могу бити **попречне гране** за DFS стабло, односно гране које повезују чворове на различитим путевима од корена (или: таква два чвора u и v да је нпр. u лево од v).

Пошто је DFS врло важан алгоритам, дајемо и нерекурзивну варијанту његове реализације. Основна алатка за реализацију рекурзивног програма је стек, на коме се чувају информације потребне за "размотавање" рекурзивних позива. Преводилац обезбеђује простор на стеку за локалне податке придружене свакој копији рекурзивне процедуре. Због тога, кад се заврши са једним рекурзивним позивом процедуре, могућ је повратак на тачно одређено место у позивајућој процедури (која може да буде друга копија исте рекурзивне процедуре). Често није неопходно све локалне податке чувати на стеку, па се испоставља да су нерекурзивне процедуре ефикасније.

Основна потешкоћа при превођењу рекурзивне у нерекурзивну верзију потиче од неопходности експлицитног памћења података, који се иначе чувају на стеку. DFS процедуру позивали смо рекурзивно унутар **for** петље, а од програма смо очекивали да запамти тачно место у петљи, где треба наставити после рекурзивног позива. У нерекурзивној верзији се ти подаци морају експлицитно памтити. Претпостављамо да сваки чвор v има повезану листу суседних грана (DFS прати редослед грана у листи). Показивач на почетак те листе је $v.Prvi$. Сваки елеменат листе је слог који садржи две променљиве: $\check{C}vor$ и $Naredni$; $\check{C}vor$ је име чвора на другом крају гране, а $Naredni$ је показивач на наредни елеменат листе. Последњи елеменат листе $Naredni$ има вредност **nil**. DFS се извршава као и раније, силазећи низ стабло све до доласка у "слепу улицу". За регистровање достигнутог нивоа претраге користи се стек. У току обиласка на стеку се чувају сви чворови на путу од корена до текућег чвора, оним редом којим су на путу. Између свака два чвора $Otac$ и Sin стек садржи показивач на грану из $Otac$ која ће бити следећа прегледана после повратка из Sin. Нерекурзивна верзија DFS приказана је на слици 9.

6.3.1.3. *Усмерени графови*. Процедура претраге у дубину усмерених графова иста је као за неусмерене графове. Међутим, усмерена DFS стабла имају нешто другачије особине. За њих, на пример, није тачно да немају попречне гране, што се може видети из примера на слици 10. Гране графа могу се поделити на четири врсте: **гране стабла**, **повратне**, **директне** и **попречне**. Прве три врсте грана повезују два чвора од којих је један потомак другог у стаблу: грана стабла повезује оца са сином, повратна грана потомка са претком, а директна грана претка са потомком. Једино попречне гране повезују чворове

```
Алгоритам Nerek_DFS(G, v);
Улаз: G = (V, E) (неусмерени граф) и v (чвор графа G).
Излаз: зависи од примене.
begin
    while постоји неозначени чвор w do
        if v неозначен then w := v; {претрага почиње од v}
        означи w;
        изврши улазну обраду на w;
        Grana := w.Prvi; {показивач на прву грану из w}
        упиши на стек w и Grana;
        Otac := w;
        {почетак основне петље рекурзије}
        while стек није празан do
            скини Grana са врха стека;
            while Grana ≠ nil do {док има непрегледаних грана из чвора}
                Sin := Grana^.Čvor;
                if Sin није означен then {корак у дубину}
                    означи Sin;
                    изврши улазну обраду на Sin;
                    упиши Grana^.Naredni на врх стека; {следећа, кад се обради Sin}
                    {за наставак по завршетку обраде Sin}
                    Grana := Sin.Prvi;
                    Otac := Sin;
                    упиши Otac на врх стека {новоозначени чвор};
                else {Grana је повратна грана};
                    изврши излазну обраду за (Otac, Sin);
                    {ово се прескаче ако се излазна обрада ради само за гране стабла}
                    Grana := Grana^.Naredni;
                {на врху стека су последњи означени чвор и следећа грана из њега}
            скини Sin са врха стека; {корак назад}
            if стек није празан then {стек постаје празан кад је Sin корен}
                нека су Grana и Otac на врху стека; {не уклањати их са стека}
                изврши излазну обраду за (Otac, Sin)
end
```

Рис. 9. Нерекурзивни алгоритам претраге у дубину.

који нису "сродници" у стаблу. Попречне гране, међутим, морају бити усмерене "здесна улево", као што показује следећа лема.

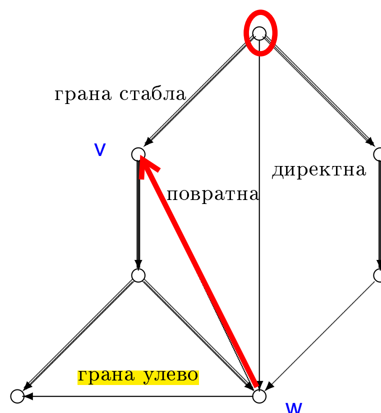


Рис. 10. DFS стабло усмереног графа.

Лема 6.3 (Основна особина усмерених DFS стабала)**.** *Нека је* $G = (V, E)$ *усмерени граф, и нека је* $T = (V, F)$ DFS *стабло графа* G. *Ако је* $(v, w) \in E$ *грана графа* G *за коју важи* $v.Pre < w.Pre$, *онда је* w *потомак* v *у стаблу* T.

Доказательство. Пошто је према долазној DFS нумерацији v испред w, w је означен после v. Грана графа (v, w) мора бити разматрана у току рекурзивног позива DFS из чвора v. Ако у том тренутку чвор w није био означен, онда се грана (v, w) мора укључити у стабло, тј. $(v, w) \in F$, па је тврђење леме тачно. У противном, w је означен у току извођења рекурзивног позива DFS из v, па w мора бити потомак v у стаблу T. □

DFS за повезан неусмерени граф, стартована из произвољног чвора, обилази цео граф. Аналогно тврђење не мора бити тачно за усмерене графове. Посматрајмо усмерени граф на слици 11. Ако се DFS започне из чвора v, онда ће бити достигнути само чворови у десној колони. DFS може да достигне све чворове графа само ако се започне из чвора a. Ако се чвор a уклони из графа заједно са две гране које излазе из њега, онда више неће постојати чвор из кога DFS обилази цео граф. Према томе, увек кад говоримо о DFS усмерног графа, сматраћемо да је DFS покренута толико пута колико је потребно да би сви чворови били означени и све гране биле размотрене. Дакле, у општем случају граф уместо DFS стабла има DFS шуму.

Непостојање грана графа које иду слева удесно говори нешто занимљиво и корисно о одлазној нумерацији чворова графа и о четири врсте грана у односу на DFS стабло. На слици 12(а) приказана су три чвора графа u, v и w у DFS обиласку графа. Чворови v и w су потомци чвора u, а чвор w је десно од чвора v. На слици 12(б) приказани су временски интервали трајања рекурзивних позива DFS за сваки од ових чворова. Запажамо да DFS за потомка, као што је v, активна само у подинтервалу времена за које је DFS активна за претка, као што је u. Специјално, DFS из v завршава се пре DFS из u. Према томе, из чињенице да је v потомак u следи да је $v.Post < u.Post$. Поред тога, ако

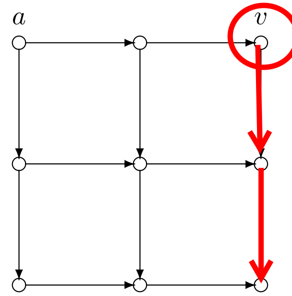
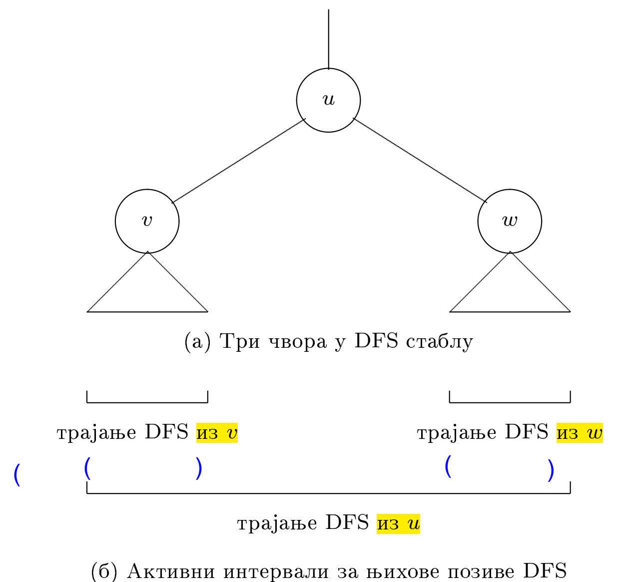


Рис. 11. Пример кад DFS усмереног графа не обилази комплетан граф.

је w десно од v, онда позив DFS из w не може бити активиран пре него што се заврши DFS из v. Према томе, ако је v лево од w, онда је $v.Post < w.Post$. Иако то није показано на слици 12, исти закључак је тачан и ако су v и w у различитим стаблима DFS шуме, при чему је стабло чвора v лево од стабла чвора w.



(а) Три чвора у DFS стаблу

(б) Активни интервали за њихове позиве DFS

Рис. 12. Однос између положаја чвора у DFS стаблу и трајања рекурзивних позива.

Размотримо сада за сваку грану (u, v) однос одлазних DFS бројева чворова u и v.

(1) Ако је (u, v) грана стабла или директна грана, онда је v потомак u, па је $v.Post < u.Post$.

(1)' директна грана: v.Post < u.Post

(2) Ако је (u, v) попречна грана, онда је због тога што је v лево од u, поново $v.Post < u.Post$.
(3) Ако је (u, v) повратна грана и $v \neq u$, онда је v прави предак u и $v.Post > u.Post$. Међутим, $v = u$ је могуће за повратну грану, јер је и петља повратна грана. Према томе, за повратну грану (u, v) знамо да је $v.Post \geq u.Post$.

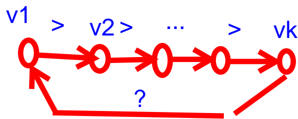
Према томе, доказано је следеће тврђење.

Лема 6.4. *Грана (u, v) усмереног графа $G = (V, E)$ је повратна ако и само ако према одлазној нумерацији чвор u претходи чвору v, односно $u.Post \leq v.Post$.*

Показаћемо сада како се DFS може искористити за утврђивање да ли је задати граф ациклички.

Проблем. За задати усмерени граф $G = (V, E)$ установити да ли садржи усмерени циклус.

Лема 6.5. *Нека је $G = (V, E)$ усмерени граф, и нека је T DFS стабло графа G. Тада G садржи усмерени циклус ако и само ако G садржи повратну грану (у односу на T).*



Доказательство. Ако је грана (u, v) повратна, онда она заједно са гранама стабла на путу од v до u затвара циклус. Супротно тврђење је такође тачно: ако у графу постоји циклус, тада у њему мора да постоји повратна грана. Заиста, претпоставимо да у графу постоји циклус који чине гране (v_1, v_2), (v_2, v_3),$\ldots(v_k, v_1)$. Ако је $k = 1$, односно циклус је петља, онда је грана (v, v) повратна грана. Ако је пак $k > 1$, претпоставимо да ни једна од грана (v_1, v_2), (v_2, v_3),$\ldots(v_{k-1}, v_k)$ није повратна. Према претходној леми је тада $v_1.Post > v_2.Post > \cdots > v_k.Post$, односно $v_k.Post < v_1.Post$, па је грана (v_k, v_1) повратна. Тиме је доказано да у сваком циклусу постоји повратна грана. □

Алгоритам за проверу да ли граф садржи циклус своди се на DFS нумерацију и проверу постојања повратне гране на основу леме, видети слику 13.

6.3.2. Претрага у ширину. Претрага у ширину (или BFS, што је скраћеница од breadth–first–search) је обилазак графа на систематичан начин, ниво по ниво. Ако полазимо од чвора v, онда се најпре посећују сви суседи v (деца у стаблу претраге, ниво један). Затим се долази до свих "унука" (ниво два), и тако даље (видети слику 14). Обилазак се реализује слично као у нерекурзивној верзији DFS, сем што је стек замењен листом FIFO (скраћеница од first–in–first–out queue). Приликом обиласка чворови се могу нумерисати BFS бројевима, слично као при DFS. Прецизније, чвор w има BFS број k ако је он k-ти чвор означен у току BFS. BFS стабло графа може се формирати укључивањем само грана ка новоозначеним чворовима. Алгоритам BFS дат је на слици 15. Запажа се да овде нема смисла излазна обрада

Алгоритам Acikl(G);
Улаз: $G = (V, E)$ (усмерени граф).
Излаз: *Postoji_ciklus* (тачно ако G садржи циклус и нетачно у противном).
Иницијализација: $v.Na_putu := false$ за све чворове и $Postoji_ciklus := false$;
Покренути ДФС из произвољног чвора, са следећом улазном и излазном обрадом:
улазна обрада:
  if из v излази бар једна грана **then**
    $v.Na_putu := true$;
    $\{x.Na_putu$ је тачно ако је x на путу од корена до текућег чвора$\}$
излазна обрада:
  if $w.Na_putu$ **then** $Postoji_ciklus := true$; $halt$;
  if w последњи чвор у списку суседа v **then** $v.Na_putu := false$;

Рис. 13. Алгоритам за проверу да ли граф садржи циклус.

као код DFS; претрага нема повратак "навише", већ се креће само наниже од корена.

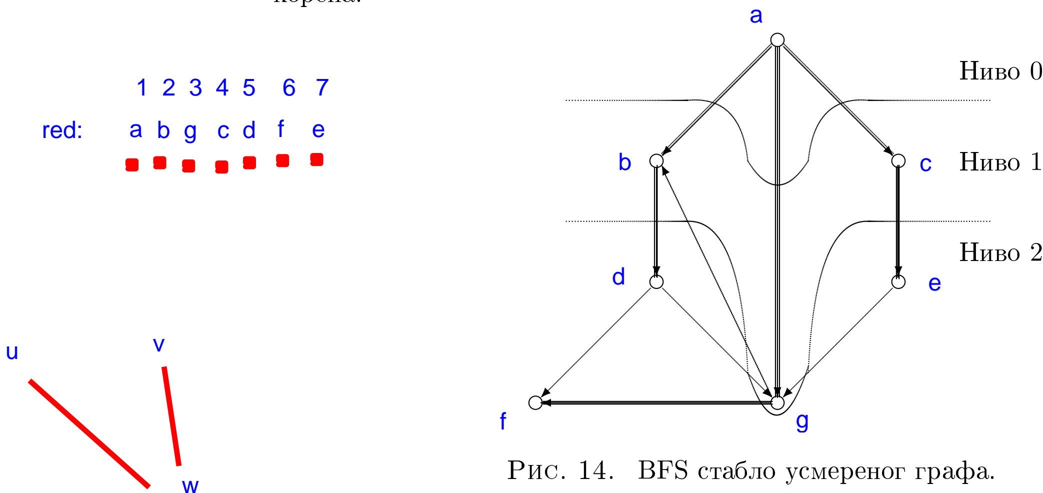


Рис. 14. BFS стабло усмереног графа.

Лема 6.6. *Ако грана* (u, w) *припада* BFS *стаблу и чвор* u *је отац чвора* w*, онда чвор* u *има минимални* BFS *број међу чворовима из којих постоји грана ка* w*.*

Доказательство. Ако би у графу постојала грана (v, w), при чему v има мањи BFS број од u, онда би у тренутку обраде v на листу морао бити стављен чвор w, па би грана (v, w) морала бити укључена у стабло, супротно претпоставци. □

Лема 6.7. *Пут од корена* r BFS *стабла до произвољног чвора* w *кроз* BFS *стабло најкраћи је пут од* r *до* w *у графу* G.

Алгоритам BFS(G, v);

Улаз: $G = (V, E)$ (неусмерени повезани граф) и v (чвор графа G).

Излаз: зависи од примене.

begin

  означи v;

  упиши v у листу; {FIFO листа, први унутра – први напоље}

  while листа је непразна **do**

    скини први чвор w са листе;

    изврши улазну обраду на w; {улазна обрада зависи од примене BFS}

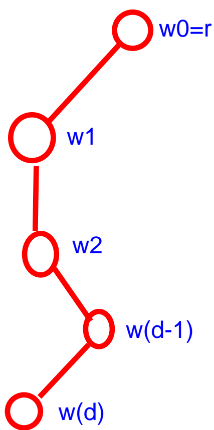
    for све гране (w, x) за које x није означен **do**

      означи x;

      додај (w, x) у стабло T;

      упиши x у листу;

end



nivo cvora w = minimalno rastojanje cvora w od korena r

dva smera, <= i >=

Рис. 15. Претрага у ширину (BFS).

indukcijom po velicini minimalnog rastojanja cvora w od korena r

Доказательство. Индукцијом по d доказаћемо да до сваког чвора w на растојању d од корена r (јединствени) пут кроз стабло од r до v има дужину d. За $d = 1$ тврђење је тачно: грана (r, w) је обавезно део стабла, па између r и w постоји пут кроз стабло дужине 1. Претпоставимо да је тврђење тачно за чворове који су на растојању мањем од d од корена, и нека је w неки чвор на растојању d од корена, тј. постоји низ чворова $w_0 = r$, w_1, $w_2, \ldots, w_d = w$ који чине пут дужине d од r до w. Према индуктивној хипотези пут од r до w_{d-1} кроз стабло има дужину $d - 1$. У тренутку обраде чвора w_{d-1}, пошто у G постоји грана (w_{d-1}, w_d), та грана се укључује у BFS стабло, па до чвора w_d постоји пут дужине d кроз стабло. □

Ниво чвора w је дужина пута у стаблу од корена до w. Претрага у ширину обилази граф ниво по ниво.

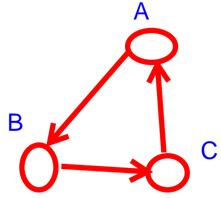
Лема 6.8. *Ако је (v, w) грана из E која не припада T, онда она спаја два чвора чији се нивои разликују највише за један.*

Доказательство. Нека је нпр. чвор v први достигнут претрагом и нека је његов ниво d. Тада је ниво чвора w већи или једнак од d. Тај ниво једнак је растојању w од корена, па је мањи или једнак од $d + 1$, јер до w постоји пут дужине $d + 1$ (пут од корена до v продужен граном (v, w)). □

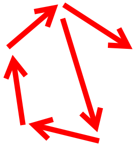
Размотрићемо сада неколико алгоритама за обраду графова.

6.4. Тополошко сортирање

Претпоставимо да је задат скуп послова на чији редослед извршавања су задата 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 такав распоред, да се по њему сваки посао започиње тек кад буду завршени сви послови од којих он зависи. Потребно је конструисати ефикасни алгоритам за формирање таквог распореда. Проблем се зове **тополошко сортирање**. Задатим пословима и њиховим међузависностима може се на природан начин придружити граф. Сваком послу придружује се чвор, а усмерена грана од посла x до посла y постоји ако се посао y не може започети пре завршетка посла x. Јасно је да граф мора бити ациклички, (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 од 1 до n, тако да ако је произвољан чвор v нумерисан са k, онда сви чворови до којих постоји усмерени пут из v имају број већи од k.

Природна је следећа индуктивна хипотеза.

Индуктивна хипотеза. Умемо да нумеришемо на захтевани начин чворове свих усмерених ацикличких графова са мање од n чворова.

Базни случај једног чвора, односно посла, је тривијалан. Као и обично, посматрајмо произвољни граф са n чворова, уклонимо један чвор, применимо индуктивну хипотезу и покушајмо да проширимо нумерацију на полазни граф. Имамо слободу избора n-тог чвора. Требало би га изабрати тако да остатак посла буде што једноставнији. Потребно је нумерисати чворове. Који чвор је најлакше нумерисати? То је очигледно чвор (посао) који не зависи од других послова, односно чвор са улазним степеном нула; њему се може доделити број 1. Да ли се увек може пронаћи чвор са улазним степеном нула? Интуитивно се намеће потврдан одговор, јер се са означавањем негде мора започети. Следећа лема потврђује ову чињеницу.

Лема 6.9. *Усмерени ациклички граф увек има чвор са улазним степеном нула.*

Доказательство. 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 циклус. Ово је међутим супротно претпоставци да се ради о ацикличком графу. На сличан начин закључује се да у графу мора постојати и чвор са излазним степеном нула. □

Претпоставимо да смо пронашли чвор са улазним степеном 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 од 1 до $n-1$, а затим се сваки редни број

може повећати за један). Види се да је после избора чвора са улазним степеном нула, остатак посла једноставан.

Реализација. Једини проблем при реализацији је како пронаћи чвор са улазним степеном нула и како поправити улазне степене чворова после уклањања гране. Сваком чвору може се придружити променљива (поље) *UlStepen*, тако да је на почетку *v.UlStepen* једнако улазном степену чвора v. Почетне вредности промењивих *UlStepen* могу се добити проласком кроз скуп свих грана произвољним редоследом (коришћењем DFS, на пример) и повећавањем за један *w.UlStepen* сваки пут кад се наиђе на грану (v, w). Чворови са улазним степеном нула стављају се у листу (или стек, што је једнако добро). Према леми 6.9 у графу постоји бар један чвор v са улазним степеном нула. Чвор v се као први у листи лако проналази; он се уклања из листе. Затим се за сваку грану (v, w) која излази из v бројач *w.UlStepen* смањује за један. Ако бројач при томе добије вредност нула, чвор w ставља се у листу. После уклањања чвора v граф остаје ациклички, па у њему према леми 6.9 поново постоји чвор са улазним степеном нула. Алгоритам завршава са радом кад листа са чворовима степена нула постане празна, јер су у том тренутку сви чворови нумерисани. Алгоритам је приказан на слици 16.

```
Алгоритам Top_sort(G);
Улаз: G = (V, E) (усмерени ациклички граф).
Излаз: v.Rb за сваки чвор v добија вредност у складу са тополошким сортирањем G.
begin
   Иницијализовати v.UlStepen за све чворове; {нпр. помоћу DFS}
   G_rb := 0;
   for i := 1 to n do
      if v_i.UlStepen = 0 then стави v_i у листу;
   while листа није празна
      скини чвор v из листе;
      G_rb := G_rb + 1;
      v.Rb := G_rb;
      for све гране (v, w) do
         w.UlStepen := w.UlStepen − 1;
         if w.UlStepen = 0 then стави w у листу;
end
```

Рис. 16. Алгоритам за тополошко сортирање ацикличког усмереног графа.

Сложеност. Сложеност израчунавања почетних вредности променљивих *UlStepen* је $O(|V|+|E|)$. За налажење чвора са улазним степеном нула потребно је константно време (приступ листи). Свака грана (v, w) разматра се тачно

једном, у тренутку кад се v уклања из листе. Према томе, број промена вредности $UlStepen$ једнак је броју грана у графу. Временска сложеност алгоритма је дакле $O(|V| + |E|)$, односно линеарна је функција од величине улаза.

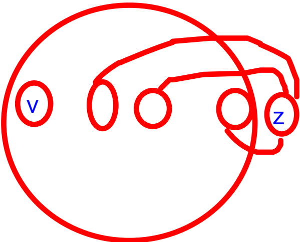
6.5. Најкраћи путеви из задатог чвора

У овом одељку бавићемо се **тежинским графовима**. Нека је $G = (V, E)$ усмерени граф са ненегативним **тежинама** придруженим гранама. Овде ћемо тежине звати *дужинама*, пошто оперишемо **дужинама путева** као збировима дужина грана (не бројевима грана). Ако је граф неусмерен, можемо га сматрати усмереним, при чему свакој његовој неусмереној грани одговарају две усмерене гране исте дужине, у оба смера. Према томе, разматрање у овом одељку односи се и на неусмерене графове.

Проблем. За дати усмерени граф $G = (V, E)$ и задати његов чвор v пронаћи најкраће путеве од v до свих осталих чорова G.

Због једноставности ћемо се бавити само налажењем дужина најкраћих путева. Алгоритми се могу проширити тако да проналазе и саме најкраће путеве. Постоји много ситуација у којима се појављује овај проблем. На пример, граф може одговарати ауто–карти: чворови су градови, а дужине грана су дужине директних путева између градова (или време потребно да се тај пут пређе, или изгради, итд, зависно од проблема).

6.5.1. Ациклички случај. Претпоставимо најпре да је граф G ацикличан. У том случају проблем је лакши, и његово решење помоћи ће нам да га решимо у општем случају. Покушаћемо индукцијом по броју чворова. Базни случај је тривијалан. Нека је $|V| = n$. Можемо да искористимо тополошко сортирање графа из претходног одељка. Ако је редни број чвора v једнак k, онда се чворови са редним бројевима мањим од k не морају разматрати: не постоји начин да се до њих дође из v. Поред тога, редослед добијен тополошким сортирањем је погодан за примену индукције. Посматрајмо последњи чвор, односно чвор z са редним бројем n. Претпоставимо (индуктивна хипотеза) да знамо најкраће путеве од v до свих осталих чворова, сем до z. Означимо дужину најкраћег пута од v до w са $w.SP$. Да бисмо одредили $z.SP$, довољно је да проверимо само оне чворове w из којих постоји грана до z. Пошто се најкраћи путеви до осталих чворова већ знају, $z.SP$ једнако је минимуму збира $w.SP + dužina(w, z)$, по свим чворовима w из којих води грана до z. Да ли је тиме проблем решен? Питање је да ли додавање чвора z може да скрати пут до неког другог чвора. Међутим, пошто је z последњи чвор у тополошком редоследу, ни један други чвор није достижан из z, па се дужине осталих најкраћих путева не мењају. Дакле, уклањање z, налажење најкраћих путева без њега, и враћање z назад су основни делови алгоритма. Другим речима, следећа индуктивна хипотеза решава проблем.



Индуктивна хипотеза. Ако се зна тополошки редослед чворова, умемо да израчунамо дужине најкраћих путева од v до првих $n - 1$ чворова.

Кад је дат ациклички граф са n чворова (тополошки уређених), уклањамо n-ти чвор, индукцијом решавамо смањени проблем, налазимо најмању међу вредностима $w.SP + dužina(w, z)$, за све чворове w такве да $(w, z) \in E$. Алгоритам је приказан на слици 17. Сада ћемо покушати да усавршимо алгоритам тако да се тополошко сортирање обавља истовремено са налажењем најкраћих путева. Другим речима, циљ је објединити два пролаза (за тополошко сортирање и налажење најкраћих путева) у један.

Алгоритам Acikl_najkr_putevi(G, v, n);
Улаз: $G = (V, E)$ (тежински ациклички граф), v (чвор) и n (број чворова).
Излаз: за сваки чвор $w \in V$, $w.SP$ је дужина најкраћег пута од v до w.
{Претпостављамо да је већ извршено тополошко сортирање.}
{Побољшани алгоритам који обухвата и тополошко сортирање дат је на слици 18.}
{Пре позива ове процедуре ставља се $z.SP := \infty$ за све чворове $z \neq v$.}
begin
  нека је z чвор са редним бројем n у тополошком редоследу;
  if $z \neq v$ **then**
    $Acikl_najkr_putevi(G - z, v, n - 1)$;
    {$G - z$ добија се уклањањем z са свим суседним гранама из G}
    for све чворове w такве да је $(w, z) \in E$ **do**
      if $w.SP + dužina(w, z) < z.SP$ **then**
        $z.SP := w.SP + dužina(w, z)$;
  else $v.SP := 0$;
end

Рис. 17. Алгоритам за налажење дужина најкраћих путева од задатог чвора у ацикличком графу.

Размотримо начин на који се алгоритам рекурзивно извршава (после налажења тополошког редоследа). Претпоставимо, због једноставности, да је редни број чвора v у тополошком редоследу 1 (чворови са редним бројем мањим од редног броја v ионако нису достижни из v). Први корак је позив рекурзивне процедуре. Процедура затим позива рекурзивно саму себе, све док се не дође до чвора v. У том тренутку се дужина најкраћег пута до v поставља на 0, и рекурзија почиње да се "размотава". Затим се разматра чвор u са редним бројем 2; дужина најкраћег пута до њега изједначује се са дужином гране (v, u), ако она постоји; у противном, не постоји пут од v до u. Следећи корак је провера чвора x са редним бројем 3. У овом случају у x улазе највише две гране (од v или u), па се упоређују дужине одговарајућих путева. Уместо оваквог извршавања рекурзије уназад, покушаћемо да исте кораке извршимо преко низа чворова са растућим редним бројевима.

Индукција се примењује према растућим редним бројевима почевши од v. Овај редослед ослобађа нас потребе да редне бројеве унапред знамо, па

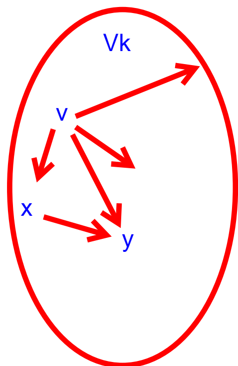
ћемо бити у стању да извршавамо истовремено оба алгоритма. Претпоставимо да су дужине најкраћих путева до чворова са редним бројевима од 1 до m познати, и размотримо чвор са редним бројем $m+1$, који ћемо означавати са z. Да бисмо пронашли најкраћи пут до z, морамо да проверимо све гране које воде у z. Тополошки редослед гарантује да све такве гране полазе из чворова са мањим редним бројевима. Према индуктивној хипотези ти чворови су већ разматрани, па се дужине најкраћих путева до њих знају. За сваку грану (w, z) знамо дужину $w.SP$ најкраћег пута до w, па је дужина најкраћег пута до z преко w једнака $w.SP + dužina(w, z)$. Поред тога, као и раније, не морамо да водимо рачуна о евентуалним променама најкраћих путева ка чворовима са мањим редним бројевима, јер се до њих не може доћи из z. Побољшани алгоритам приказан је на слици 18.

Алгоритам Acikl_najkr_putevi2(G, v);
Улаз: $G = (V, E)$ (тежински ациклички граф), v (чвор G).
Излаз: за сваки чвор $w \in V$, $w.SP$ је дужина најкраћег пута од v до w.
{Нерекурзивна верзија претходног алгоритма која обавља и тополошко сортирање.}

```
begin
    for све чворове w do
        w.SP := ∞;
    Иницијализуј v.UlStepen за све чворове; {улазни степени чворова; нпр. DFS}
    for i := 1 to n do
        if v_i.UlStepen = 0 then стави v_i у листу;
    v.SP := 0;
    while листа није празна
        скини чвор w из листе;
        for све гране (w, z) do
            if w.SP + dužina(w, z) < z.SP then
                z.SP := w.SP + dužina(w, z);
            z.UlStepen := z.UlStepen − 1;
            if z.UlStepen = 0 then стави z у листу;
end
```

Рис. 18. Побољшани алгоритам за налажење дужина најкраћих путева од задатог чвора у ацикличком графу.

Сложеност. Свака грана се по једном разматра у току иницијализације улазних степенова, и по једном у тренутку кад се њен полазни чвор уклања из листе. Приступ листи захтева константно време. Сваки чвор се разматра тачно једном. Према томе, временска сложеност алгоритма у најгорем случају је $O(|V| + |E|)$.



6.5.2. Општи случај. Кад граф није ациклички, не постоји тополошки редослед, па се разматрани алгоритми не могу директно применити. Међутим, основне идеје се могу искористити и у општем случају. Једноставност размотрених алгоритама последица је следеће особине тополошког редоследа:

> Ако је z чвор са редним бројем k, онда (1) не постоје путеви од z до чворова са редним бројевима мањим од k, и (2) не постоје путеви од чворова са редним бројевима већим од k до z.

Ова особина омогућује нам да нађемо најкраћи пут од v до z, не водећи рачуна о чворовима који су после z у тополошком редоследу. Може ли се некако дефинисати редослед чворова произвољног графа који би омогућио нешто слично?

Идеја је разматрати чворове графа редом према дужинама најкраћих путева до њих од v. Те дужине се на почетку, наравно, не знају; оне се израчунавају у току извршавања алгоритма. Најпре проверавамо све гране које излазе из v. Нека је (v, x) најкраћа међу њима. Пошто су по претпоставци све дужине грана позитивне, најкраћи пут од v до x је грана (v, x). Дужине свих других путева до x су веће или једнаке од дужине ове гране. Према томе, знамо најкраћи пут до x, и то може да послужи као база индукције. Покушајмо да направимо следећи корак. Како можемо да пронађемо најкраћи пут до неког другог чвора? Бирамо чвор који је други најближи до v (x је први најближи). Једини путеви које треба узети у обзир су друге гране из v или путеви који се састоје од две гране: прва је (v, x), а друга је грана из чвора x. Нека је са $dužina(u, w)$ означена дужина гране (u, w). Бирамо најмањи од израза $dužina(v, y)$ $(y \neq x)$ или $dužina(v, x) + dužina(x, z)$ $(z \neq v)$. Још једном закључујемо да се други путеви не морају разматрати, јер је ово најкраћи пут за одлазак из v (изузев до x). Може се формулисати следећа индуктивна хипотеза.

Индуктивна хипотеза. За задати граф и његов чвор v, умемо да пронађемо k чворова најближих чвору v, као и дужине најкраћих путева до њих.

Запазимо да је индукција 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 v, и да умемо да их пронађемо. Ми умемо да пронађемо први најближи чвор, па је база (случај $k = 1$) решена. Кад k добије вредност $|V| - 1$, решен је комплетан проблем.

Означимо са V_k скуп који се састоји од v и k њему најближих чворова. Проблем је пронаћи чвор w који је најближи чвору v међу чворовима ван V_k, и пронаћи најкраћи пут од v до w. Најкраћи пут од v до w може да саржи само чворове из V_k. Он не може да садржи неки чвор y ван V_k, јер би y био ближи чвору v од w. Према томе, да бисмо пронашли чвор w, 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 (u, z) грана таква да је $u \in V_k$ и $z \notin V_k$. Таква грана одређује пут од v до z који се састоји од најкраћег пута од v до u (према индуктивној хипотези већ познат) и гране (u, z). Довољно је упоредити све такве путеве и изабрати најкраћи међу њима, видети слику 19.

ИХ: до свих чворова ван Vk знамо
дужину најкраћег пута
једном граном из Vk;
у Vk је к најближих чворова од v

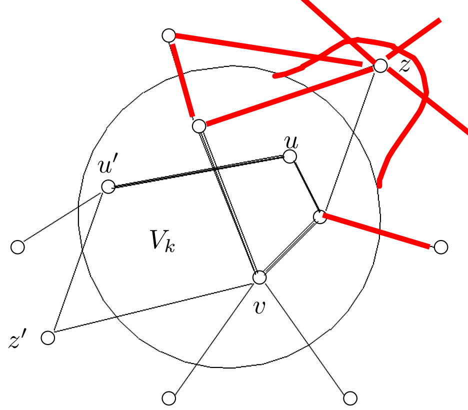


Рис. 19. Налажење следећег најближег чвора задатом чвору v.

Алгоритам одређен овом индуктивном хипотезом извршава се на следећи начин. У свакој итерацији додаје се нови чвор. То је чвор w за који је најмања дужина

$$\min\{u.SP + dužina(u, w) |\, u \in V_k\} \tag{6.1}$$

међу свим чворовима $w \notin V_k$. Из већ изнетих разлога, w је заиста $(k + 1)$-ви (следећи) најближи чвор чвору v. Према томе, његово додавање продужује индуктивну хипотезу.

Алгоритам је сада комплетно специфициран, али му се ефикасност може побољшати. Основни корак алгоритма је проналажење следећег најближег чвора. То се остварује израчунавањем најкраћег пута према (6.1). Међутим, није неопходно у сваком кораку проверавати све вредности $u.SP + dužina(u, w)$. Већина тих вредности не мења се при додавању новог чвора: могу се променити само оне вредности које одговарају путевима кроз новододати чвор. Ми можемо да памтимо дужине познатих најкраћих путева до свих чворова ван V_k, и да им поправљамо вредности само при проширивању V_k. Једини начин да се добије нови најкраћи пут после додавања w у V_k је да тај пут пролази кроз w. Према томе, треба проверити све гране од w ка чворовима ван V_k. За сваку такву грану (w, z) упоређујемо дужину $w.SP + dužina(w, z)$ са вредношћу $z.SP$, и по потреби поправљамо $z.SP$. Свака итерација обухвата налажење чвора са најмањом вредношћу SP, и поправку вредности SP за неке од преосталих чворова. Овај алгоритам познат је као Дијкстрин алгоритам (Dijkstra).

Реализација. Потребно је да проналазимо најмању вредност у скупу дужина путева и да често поправљамо дужине путева. Добра структура података за налажење минималних елемената и за поправке дужина елемената је хип. Пошто је потребно да пронађемо чвор са најмањом дужином пута до њега, све чворове ван скупа V_k чувамо у хипу, са кључевима једнаким дужинама тренутно најкраћих путева од v до њих. На почетку су све дужине путева сем једне једнаке ∞, па редослед елемената у хипу није битан, сем што v мора бити

на врху. Налажење чвора w је једноставно: он се узима са врха хипа. Могу се проверити све гране (w, u), и затим без тешкоћа поправити дужине путева. Међутим, кад се промени дужина пута до неког чвора z, може се променити положај z у хипу. Према томе, потребно је на одговарајући начин поправљати хип (проблем је у томе што хип као структура података не омогућује ефикасно проналажење задатог елемента). Лоцирање чвора z у хипу може се извести помоћу друге структуре података повезане са хипом. Подаци о свим чворовима могу се сместити у вектор са показивачима на њихове позиције у хипу. Налажење чвора у хипу је због тога еквивалентно приступању елементу вектора. Пошто су елементи хипа чворови графа, просторна сложеност је $O(|V|)$, што је прихватљиво. Дужине путева могу само да опадају. Ако неки елеменат хипа постане мањи од оца, он се може замењивањем померати навише док му се не пронађе одговарајући положај. То је исти поступак као и при обичном поступку поправке хипа (при уметању елемента у хип, на пример). Алгоритам за налажење најкраћих путева од задатог чвора приказан је на слици 20.

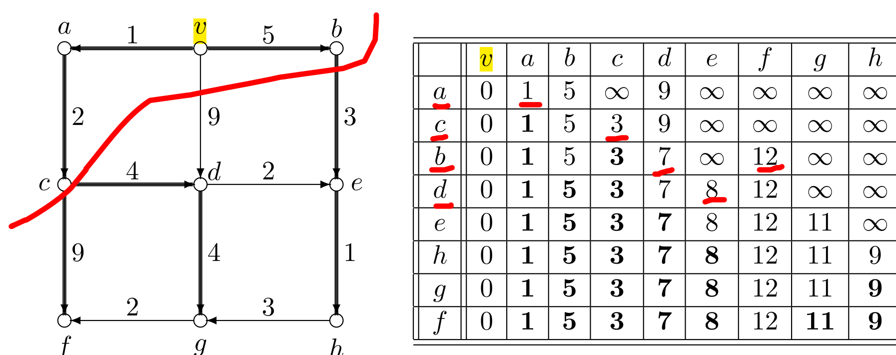
```
Алгоритам Najkr_putevi(G, v);
Улаз: G = (V, E) (тежински усмерени граф), v (полазни чвор).
Излаз: за сваки чвор w, w.SP је дужина најкраћег пута од v до w.
{Претпоставка је да су све дужина грана ненегативне.}
begin
    for све чворове w do
        w.Oznaka := false; {означени чворови су у V_k}
        w.SP := ∞;
    v.SP := 0;
    while постоји неозначен чвор do
        нека је w неозначени чвор са најмањом вредношћу w.SP;
        w.Oznaka := true; {укључивање w у V_k}
        for све гране (w, z) такве да је z неозначен do
            if w.SP + dužina(w, z) < z.SP then
                z.SP := w.SP + dužina(w, z);
end
```

Рис. 20. Дијкстрин алгоритам за налажење дужина најкраћих путева од задатог до свих осталих чворова у графу.

Сложеност. Поправка дужине пута захтева $O(\log m)$ упоређивања, где је m величина хипа. Укупно има $|V|$ итерација, и због тога $|V|$ брисања из хипа. Поправке треба извршити највише $|E|$ пута (јер свака грана може да проузрокује највише једну поправку), па је потребно извршити највише $O(|E| \log |V|)$ упоређивања у хипу. Према томе, временска сложеност алгоритма је $O((|E| +$

$|V|)\log|V|)$. Запажа се да је алгоритам спорији него алгоритам који исти проблем решава за ацикличке графове: у другом случају следећи чвор се узима из (произвољно уређене) листе, и никакве поправке нису потребне.

Пример 6.1. Пример извршавања алгоритма *Najkr_putevi* за налажење најкраћих путева од задатог чвора у графу дат је на слици 21. Прва врста односи се само на путеве од једне гране из v. Бира се најкраћи пут, у овом случају он води ка чвору a. Друга врста показује поправке дужина путева укључујући сада све путеве од једне гране из v или a, и најкраћи пут сада води до c. У свакој линији бира се нови чвор, и приказују се дужине тренутних најкраћих путева од v до свих чворова. Подебљана су растојања за која се поуздано зна да су најкраћа.

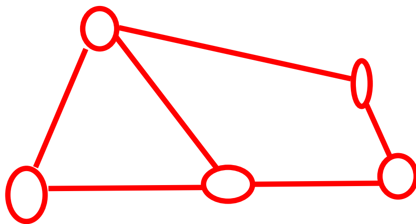


	v	a	b	c	d	e	f	g	h
a	0	1	5	∞	9	∞	∞	∞	∞
c	0	**1**	5	3	9	∞	∞	∞	∞
b	0	**1**	5	**3**	7	∞	12	∞	∞
d	0	**1**	**5**	**3**	7	8	12	∞	∞
e	0	**1**	**5**	**3**	**7**	8	12	11	∞
h	0	**1**	**5**	**3**	**7**	**8**	12	11	9
g	0	**1**	**5**	**3**	**7**	**8**	12	11	**9**
f	0	**1**	**5**	**3**	**7**	**8**	12	**11**	**9**

Рис. 21. Пример извршавања алгоритма за налажење најкраћих путева од задатог чвора.

Коментар. Овакав тип алгоритма се понекад зове **претрага са приоритетом** — сваком чвору додељује се приоритет (у овом случају тренутно најмање познато растојање од v), па се чворови обилазе редоследом одређеним приоритетом. Кад се заврши разматрање чвора, проверавају се све њему суседне гране. Та провера може да доведе до промене неких приоритета. Начин извођења тих промена је детаљ по коме се једна претрага са приоритетом разликује од друге. Претрага са приоритетом сложенија је од обичне претраге. Она је корисна код проблема са тежинским графовима.

Најкраће путеве од v до свих осталих чворова нашли смо тако што смо путеве проналазили један по један. Сваки нови пут је одређен једном граном, која продужује претходно познати најкраћи пут до новог чвора. Све те гране формирају стабло са кореном v. Ово стабло зове се **стабло најкраћих путева**, и важно је за решавање многих проблема са путевима. Ако су тежине свих грана једнаке, онда је стабло најкраћих путева уствари BFS стабло са кореном у чвору v. У примеру на слици 21 подебљане су гране које припадају стаблу најкраћих путева.



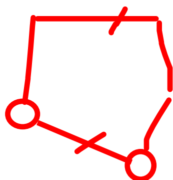
6.6. Минимално повезујуће стабло

Размотримо систем рачунара које треба повезати оптичким кабловима. Потребно је обезбедити да постоји веза између свака два рачунара. Познати су трошкови постављања кабла између свака два рачунара. Циљ је пројектовати мрежу оптичких каблова тако да цена мреже буде минимална. Систем рачунара може бити представљен графом чији чворови одговарају рачунарима, а гране – потенцијалним везама између рачунара, са одговарајућом (позитивном) ценом. Проблем је пронаћи повезани подграф (са гранама које одговарају постављеним оптичким кабловима), који садржи све чворове, такав да му сума цена грана буде минимална. Није тешко видети да тај подграф мора да буде стабло. Ако би подграф имао циклус, онда би се из циклуса могла уклонити једна грана — тиме се добија подграф који је и даље повезан, а има мању цену, јер су цене грана позитивне. Тражени подграф зове се **минимално повезујуће стабло** (MCST, скраћеница од minimum–cost spanning tree) и има много примена. Наш циљ је конструкција ефикасног алгоритма за налажење MCST. Због једноставности, претпоставимо да су цене грана различите. Ова претпоставка има за последицу да је MCST јединствено, што олакшава решавање проблема. Без ове претпоставке алгоритам остаје непромењен, изузев што, кад се наиђе на гране једнаке тежине, произвољно се бира једна од њих.

Проблем. За задати неусмерени повезани тежински граф $G = (V, E)$ конструисати повезујуће стабло T минималне цене.

У овом контексту тежине грана тежинског графа G су уствари њихове цене. Природно је користити следећу индуктивну хипотезу.

Индуктивна хипотеза (1). Умемо да конструишемо MCST за повезани граф са мање од m грана.



Базни случај је тривијалан. Ако је задат проблем MCST са m грана, како се он може свести на проблем са мање од m грана? Тврдимо да грана најмање тежине мора бити укључена у MCST. Ако она не би била укључена, онда би њено додавање стаблу MCST затворило неки циклус; уклањањем произвољне друге гране из тог циклуса поново се добија стабло, али мање тежине — што је у супротности са претпоставком о минималности MCST. Дакле, ми знамо једну грану која мора да припада MCST. Можемо да је уклонимо из графа и применимо индуктивну хипотезу на остатак графа, који сада има мање од m грана. Да ли је ово регуларна примена индукције?

Проблем је у томе што после уклањања гране, преостали проблем више није еквивалентан полазном. Прво, избор једне гране ограничава могућности избора других грана. Друго, после уклањања гране граф не мора да остане повезан.

Решење насталог проблема је у прецизирању индуктивне хипотезе. Ми знамо како да изаберемо прву грану, али не можемо да је уклонимо и просто

заборавимо на њу, јер остали избори зависе од ње. Дакле, уместо да грану уклонимо, треба да је означимо, и да ту чињеницу, њен избор, користимо даље у алгоритму. Алгоритам се извршава тако што се једна по једна грана бира и додаје у MCST. Према томе, индукција је не према величини графа, него према броју изабраних грана у задатом (фиксираном) графу.

Индуктивна хипотеза (2)**.** За задати повезан граф $G = (V, E)$ умемо да пронађемо подграф T са k грана ($k < |V| - 1$), тако да је стабло T подграф MCST графа G.

Базни случај за ову хипотезу смо већ размотрили — он се односи на избор прве гране. Претпоставимо да смо пронашли стабло T које задовољава индуктивну хипотезу и да је потребно да T проширимо наредном граном. Како да пронађемо нову грану за коју ћемо бити сигурни да припада MCST? Применићемо сличан приступ као и при избору прве гране. За T се већ зна да је део коначног MCST. Због тога у MCST мора да постоји бар једна грана која повезује неки чвор из T са неким чвором у остатку графа. Покушаћемо да пронађемо такву грану. Нека је E_k скуп свих грана које повезују T са чворовима ван T. Тврдимо да грана са најмањом ценом из E_k припада MCST. Означимо ту грану са (u, v) (видети слику 22; гране стабла T су подебљане). Пошто је MCST повезујуће стабло, оно садржи тачно један пут од u до v (између свака два чвора у стаблу постоји тачно један пут). Ако грана (u, v) не припада MCST, онда она не припада ни том путу од u до v. Међутим, пошто u припада, а v не припада T, на том путу мора да постоји бар једна грана (x, y) таква да $x \in T$ и $y \notin T$. Цена ове гране већа је од цене (u, v), јер је цена (u, v) најмања међу ценама грана које повезују T са остатком графа. Сада можемо да применимо слично закључивање као при избору прве гране. Ако додамо (u, v) стаблу MCST, а избацимо (x, y), добијамо повезујуће стабло мање цене, што је контрадикција.

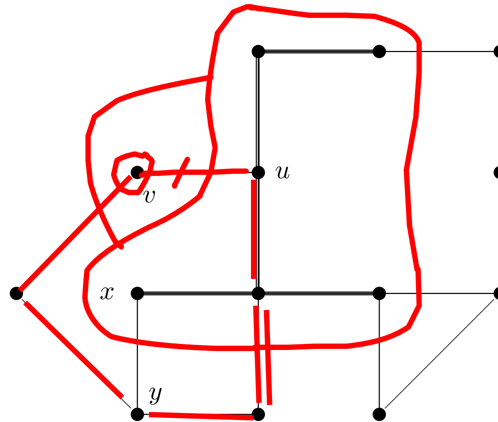


Рис. 22. Налажење следеће гране минималног повезујућег стабла (MCST).

Реализација. Описани алгоритам сличан је алгоритму за налажење најкраћих путева од задатог чвора из претходног одељка. Прва изабрана грана је грана са најмањом ценом. T се дефинише као стабло са само том једном граном. У свакој итерацији проналази се грана која повезује T са неким чвором ван T, а има најмању цену. У алгоритму за налажење најкраћих путева од задатог чвора тражили смо најкраћи *пут* до чвора ван T. Према томе, једина разлика између MCST алгоритма и алгоритма за налажење најкраћих путева је у томе што се минимум тражи не по дужини пута, него по цени гране. Остатак алгоритма преноси се практично без промене. За сваки чвор w ван T памтимо цену гране минималне цене до w од неког чвора из T, односно ∞ ако таква грана не постоји. У свакој итерацији ми на тај начин бирамо грану најмање цене и повезујемо одговарајући чвор w са стаблом T. Затим проверавамо све гране суседне чвору w. Ако је цена неке такве гране (w, z) (за $z \notin T$) мања од цене тренутно најјефтиније познате гране до z, онда поправљамо цену чвора z. Алгоритам је приказан на слици 23.

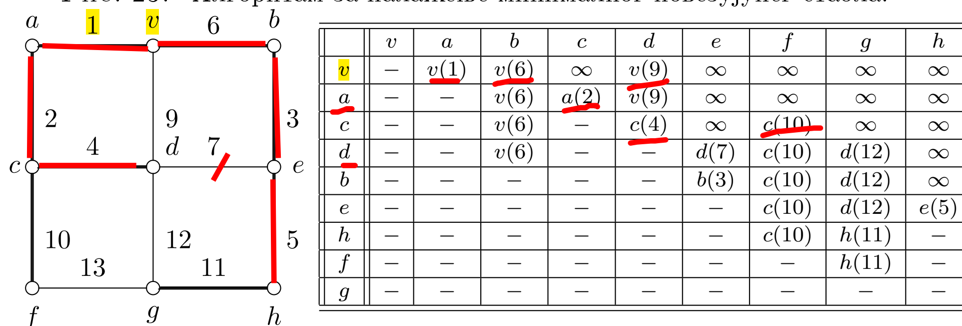
Сложеност. Сложеност овог алгоритма идентична је сложености алгоритма за налажење најкраћих растојања од задатог чвора из претходног одељка, $O((|E| + |V|) \log |V|)$.

Пример 6.2. Алгоритам за конструкцију MCST илустроваћемо примером на слици 24. Чвор у првој колони табеле је онај који је додат у одговарајућем кораку. Први додати чвор је v, и у првој врсти наведене су све гране из v са својим ценама. У свакој врсти бира се грана са најмањом ценом. Списак тренутно најбољих грана и њихових цена поправља се у сваком кораку (приказани су само крајеви грана). На слици су гране графа које припадају MCST подебљане.

Коментар. Алгоритам за конструкцију MCST је пример, иако не сасвим чист, **похлепног метода**. Претпоставимо да радимо са скупом елемената којима су придружене цене, и да је циљ пронаћи скуп елемената се максималном (или минималном) ценом који задовољава нека ограничења. У проблему MCST елементи су гране графа, а ограничење је да гране одговарају повезујућем стаблу. Суштина похлепног метода је у томе да се у сваком кораку узимају елементи са највећом ценом. У алгоритму MCST увели смо нека додатна ограничења при избору грана: разматране су само гране повезане са текућим стаблом. Због тога алгоритам није чисти пример похлепног алгоритма. MCST се може конструисати и на други начин, избором гране са најмањом ценом у сваком кораку, уз ограничење да та грана не затвара циклус. Прецизније, изабрана грана треба да повезује два чвора из различитих стабала тренутне шуме; полази се од шуме са $|V|$ чворова без грана; за обједињавање стабала у већа користи се структура података за налажење унија из одељка 3.7). Похлепни метод не води увек оптималном решењу. Он је обично само *хеуристика* за налажење субоптималних решења. Понекад, међутим, као у примеру налажења MCST, похлепни метод проналази најбоље решење.

```
Алгоритам MCST(G);
Улаз: G (тежински неусмерени граф).
Излаз: T (минимално повезујуће стабло графа G).
begin
    На почетку је T празан скуп;
    for све чворове w do
        w.Oznaka := false; {w.Oznaka је труе ако је w у T}
        w.Cena := ∞;
    нека је (x, y) грана са најмањом ценом у G;
    x.Oznaka := true; {y ће бити означено у главној петљи}
    for све гране (x, z) do
        z.Grana := (x, z); {грана најмање цене од T до z}
        z.Cena := cena(x, z); {цена гране z.Grana}
    while постоји неозначен чвор do
        нека је w неозначени чвор са најмањом вредношћу w.Cena;
        if w.Cena = ∞ then
            print "G није повезан"; halt;
        else
            w.Oznaka := true;
            додај w.Grana у T;
            {сада поправљамо цене неозначених чворова повезаних са w}
            for све гране (w, z) do
                if not z.Oznaka then
                    if cena(w, z) < z.Cena then
                        z.Grana := (w, z);
                        z.Cena := cena(w, z);
end
```

Рис. 23. Алгоритам за налажење минималног повезујућег стабла.



	v	a	b	c	d	e	f	g	h
v	–	$v(1)$	$v(6)$	∞	$v(9)$	∞	∞	∞	∞
a	–	–	$v(6)$	$a(2)$	$v(9)$	∞	∞	∞	∞
c	–	–	$v(6)$	–	$c(4)$	∞	$c(10)$	∞	∞
d	–	–	$v(6)$	–	–	$d(7)$	$c(10)$	$d(12)$	∞
b	–	–	–	–	–	$b(3)$	$c(10)$	$d(12)$	∞
e	–	–	–	–	–	–	$c(10)$	$d(12)$	$e(5)$
h	–	–	–	–	–	–	$c(10)$	$h(11)$	–
f	–	–	–	–	–	–	–	$h(11)$	–
g	–	–	–	–	–	–	–	–	–

Рис. 24. Пример извршавања алгоритма за налажење MCST.

1 2 3 4 5 7 9 10 11 12 13

av ac be cd eh ...

6.7. Сви најкраћи путеви

Сада ћемо размотрити проблем израчунавања најкраћих путева између *свака два* чвора у графу.

Проблем. Дат је тежински граф $G = (V, E)$ (усмерени или неусмерени) са ненегативним тежинама (дужинама) грана. Пронаћи путеве минималне дужине између свака два чвора.

Поново, пошто говоримо о најкраћим путевима, тежине грана зовемо дужинама. Ово је проблем налажења **свих најкраћих путева**. Због једноставности ћемо се задовољити налажењем дужина свих најкраћих путева, уместо самих путева. Претпостављамо да је граф усмерен; све што ће бити речено важи и за неусмерене графове. Поред тога, претпоставља се да су дужине грана ненегативне.

Као и обично, покушајмо са директним индуктивним приступом. Може се користити индукција по броју грана или чворова. Како се мењају најкраћи путеви у графу после додавања нове гране (u, w)? Нова грана може пре свега да представља краћи пут између чворова u и w. Поред тога, може се променити најкраћи пут између произвољна друга два чвора v_1 и v_2. Да би се установило има ли промене, треба са претходно познатом најмањом дужином пута од v_1 до v_2 упоредити збир дужина најкраћег пута од v_1 до u, гране (u, w) и дужине најкраћег пута од w до v_2. Укупно, за сваку нову грану потребно је извршити $O(|V|^2)$ провера, па је сложеност оваквог алгоритма у најгорем случају $O(|E||V|^2)$. Пошто је број грана највише $O(|V|^2)$, сложеност овог алгоритма је $O(|V|^4)$.

Како се мењају најкраћи путеви ако се у граф дода нови чвор u? Потребно је да најпре пронаћи дужине најкраћих путева од u до свих осталих чворова, и од свих осталих чворова до u. Пошто су дужине најкраћих путева који не садрже u већ познате, најкраћи пут од u до w можемо да пронађемо на следећи начин. Потребно је да одредимо само прву грану на том путу. Ако је то грана (u, v), онда је дужина најкраћег пута од u до w једнака збиру дужине гране (u, v) и дужине најкраћег пута од v до w, која је већ позната. Потребно је дакле да упоредимо ове дужине за све гране суседне са u, и да међу њима изаберемо најмању. Најкраћи пут од w до u може се пронаћи на сличан начин. Али то све није довољно. Поново је потребно да за сваки пар чворова проверимо да ли између њих постоји нови краћи пут кроз нови чвор u. За свака два чвора v_1 и v_2, да би се установило има ли промене, треба са претходно познатом најмањом дужином пута од v_1 до v_2 упоредити збир дужина најкраћег пута од v_1 до u и дужине најкраћег пута од u до v_2. То је укупно $O(|V|^2)$ провера и сабирања после додавања сваког новог чвора, па је сложеност оваквог алгоритма у најгорем случају $O(|V|^3)$. Испоставља се да је ефикаснија индукција по броју чворова него по броју грана. Међутим, постоји још боља индуктивна конструкција за решавање овог проблема.

Идеја је да се не мења број чворова или грана, него да се уведу ограничења на тип дозвољених путева. Индукција се изводи по опадајућем броју таквих

ограничења, тако да на крају долазе у обзир сви могући путеви. Нумеришимо чворове од 1 до $|V|$. Пут од u до w зове се *k-пут* ако су редни бројеви свих чворова на путу (изузев u и w) мањи или једнаки од k. Специјално, 0-пут се састоји само од једне гране (пошто се ни један други чвор не може појавити на путу).

Индуктивна хипотеза. Умемо да одредимо дужине најкраћих путева између свака два чвора, при чему су дозвољени само k-путеви, за $k < m$.

База индукције је случај $m = 1$, кад се разматрају само директне гране и решење је очигледно. Претпоставимо да је индуктивна хипотеза тачна и да хоћемо да је проширимо на $k \leq m$. Једини нови путеви које треба да размотримо су m-путеви. Треба да пронађемо најкраће m-путеве између свака два чвора и да проверимо да ли они побољшавају k-путеве за $k < m$. Нека је v_m чвор са редним бројем m. Произвољан најкраћи m-пут садржи v_m највише једном. Најкраћи m-пут између u и v састоји се од најкраћег $(m-1)$-пута од u до v_m, и најкраћег $(m-1)$-пута од v_m до v. Према индуктивној хипотези ми већ знамо дужине најкраћих k-путева за $k < m$, па је довољно да саберемо ове две дужине (и збир упоредимо са дужином најкраћег $(m-1)$-пута од u до v) да бисмо пронашли дужину најкраћег m-пута од u до v. Овај алгоритам је нешто бржи од претходног (за константни фактор), а лакше га је и реализовати, видети слику 25.

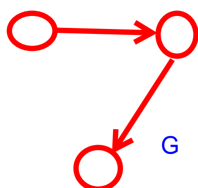
Алгоритам Svi_Najkr_putevi(W);
Улаз: W ($n \times n$ матрица повезаности која представља тежински граф).
{$W[x, y]$ је тежина гране (x, y), ако она постоји, односно ∞ у противном.}
{$W[x, x]$ је 0 за све чворове x}
Излаз: На крају матрица W садржи дужине најкраћих путева.

```
begin
  for m := 1 to n do  {индукција је по параметру m}
    for x := 1 to n do
      for y := 1 to n do
        if W[x,m] + W[m,y] < W[x,y] then
          W[x,y] := W[x,m] + W[m,y]
end
```

Рис. 25. Алгоритам за налажење дужина свих најкраћих путева у графу.

Унутрашње две петље користе се за проверу свих *парова* чворова. Запажа се да се ова провера може извршавати са паровима чворова произвољним редоследом, јер је свака провера независна од осталих. Оваква флексибилност је важна, на пример, код паралелних алгоритама.

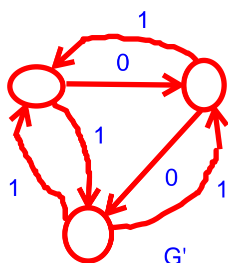
Сложеност. За свако m алгоритам извршава једно сабирање и једно упоређивање за сваки пар чворова. Број корака индукције је $|V|$, па је укупан број сабирања, односно упоређивања, највише $|V|^3$. Присетимо се да је временска сложеност алгоритма за налажење дужина најкраћих путева од једног чвора $O(|E| \log |V|)$. Ако је граф густ, па је број грана $\Omega(|V|^2)$, онда је описани алгоритам ефикаснији од извршавања за сваки чвор алгоритма за најкраће путеве од датог чвора. Иако је могуће реализовати алгоритам за најкраће путеве од једног чвора сложености $O(|V|^2)$ (видети задатак 6.31), а тиме и алгоритам сложености $O(|V|^3)$ за налажење свих најкраћих растојања, алгоритам из овог одељка бољи је за густе графове због своје једноставне реализације. С друге стране, ако граф није густ (па има на пример $O(|V|)$ грана), онда је боља временска сложеност $O(|E||V| \log |V|)$ више ($|V|$) пута употребљеног алгоритма за најкраће путеве од једног чвора.



6.8. Транзитивно затворење

За задати усмерени граф $G = (V, E)$ његово **транзитивно затворење** $C = (V, F)$ је усмерени граф у коме грана (u, w) између чворова u и w постоји ако и само ако у G постоји усмерени пут од u до w. Постоји много примена транзитивног затворења, па је важно имати ефикасни алгоритам за његово налажење.

Проблем. Пронаћи транзитивно затворење задатог усмереног графа $G = (V, E)$.



Овај проблем решићемо **редукцијом** (свођењем) на други проблем. Другим речима показаћемо како се може произвољни улаз за проблем транзитивно затворење свести на улаз за други проблем, који умемо да решимо. После тога решење другог проблема трансформишемо у решење проблема транзитивног затворења. Проблем о коме је реч је налажење свих најкраћих путева.

Нека је $G' = (V, E')$ комплетни усмерени граф (граф код кога за сваки пар чворова постоје обе гране, у оба смера). Грани $e \in E'$ додељује се дужина 0 ако је $e \in E$, односно 1 у противном. Сада за граф G' решавамо проблем налажења свих најкраћих путева. Ако у G постоји пут између v и w, онда је у G' његова дужина 0. Према томе, пут између v и w у G постоји ако и само ако је дужина најкраћег пута између v и w у G' једнака 0. Другим речима, решење проблема свих најкраћих путева непосредно се трансформише у решење проблема транзитивног затворења.

Идеја коришћења редукције једног проблема на други детаљније је разрађена у поглављу 10. Овде редукцију користимо са циљем да је као технику илуструјемо на једноставном примеру. Није тешко преправити алгоритам за све најкраће путеве, тако да директно решава проблем транзитивног затворења, видети слику 26.

Чињеница да можемо да сведемо један проблем на други значи да је први проблем општији од другог, односно да је други проблем специјални случај

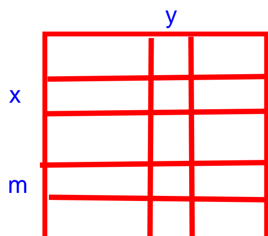
Алгоритам Tranzit_zatvorenje(A);
Улаз: A ($n \times n$ матрица повезаности која представља усмерени граф).
{$A[x, y]$ је *true* ако грана (x, y) припада графу, односно *false* у противном.}
{$A[x, x]$ је *true* за све чворове x}
Излаз: На крају матрица A представља транзитивно затворење графа.
begin
    for $m := 1$ **to** n **do** {индукција је по параметру m}
        for $x := 1$ **to** n **do**
            for $y := 1$ **to** n **do**
                if $A[x, m]$ **and** $A[m, y]$ **then** $A[x, y] := true$
                {овај корак је поправљен у следећем алгоритму}
end

Рис. 26. Алгоритам за налажење транзитивног затворења графа.

првог. Обично су општија решења скупља, сложенија. Ми смо видели много примера где је општији проблем лакше решити; ипак, да бисмо више добили, морамо више и да платимо. После коришћења редукције препоручљиво је покушати са поправком добијеног решења, користећи специјалне особине проблема.

Размотримо основни корак алгоритма, наредбу **if**. Она се састоји од две провере, $A[x, m]$ и $A[m, y]$. Нешто се предузима само ако су оба услова испуњена. Ова **if** наредба извршава се n 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 x и m, а друга зависи само од m и y. Због тога се прва провера може за фиксиране x и m извршити само једном (уместо n пута). Ако први услов није испуњен, онда се други не мора проверавати ни за једну вредност y. Ако је пак први услов испуњен, онда се његова испуњеност не мора поново проверавати. Ова промена је уграђена у побољшани алгоритам на слици 27. Асимптотска сложеност остаје непромењена, али се алгоритам извршава приближно два пута брже.

Реализација. Реализација алгоритма је директна. Резултат унутрашње петље је да је на врсту x матрице A примењена **or** (или) операција са врстом m (елеменат по елеменат). Многи рачунари могу да изврше истовремено више битских операција **or**. Због тога се **or** операција са врстама може извршити брже него више битских **or** операција. У пракси је дакле број **or** операција n^3/w, где је w величина речи на рачунару (број бита на које се може истовремено применити **or** операција). Ово је врло једноставан пример паралелног алгоритма.



Алгоритам Tranzit_zatvorenje2(A);
Улаз: A ($n \times n$ матрица повезаности која представља усмерени граф).
{$A[x,y]$ је *true* ако грана (x,y) припада графу, односно *false* у противном.}
{$A[x,x]$ је *true* за све чворове x}
Излаз: На крају матрица A представља транзитивно затворење графа.

```
begin
    for m := 1 to n do  {индукција је по параметру m}
        for x := 1 to n do
            if A[x,m] then
                for y := 1 to n do
                    if A[m,y] then A[x,y] := true        A[x,y]=A[x,y] or A[m,y]
end
```

Рис. 27. Побољшани алгоритам за налажење транзитивног затворења графа.

6.9. Упаривање

За задати неусмерени граф $G = (V, E)$ **упаривање** је скуп дисјунктних грана (грана без заједничких чворова). Ово име потиче од чињенице да се гране могу схватити као парови чворова. Битно је да сваки чвор припада највише једној грани — то је моногамно упаривање. Чвор који није суседан ни једној грани из упаривања зове се **неупарени** чвор; каже се такође да чвор не припада упаривању. **Савршено упаривање** је упаривање у коме су сви чворови упарени. **Оптимално упаривање** је упаривање са максималним бројем грана. **Максимално упаривање** је пак упаривање које се не може проширити додавањем нове гране. Проблеми који се своде на упаривање појављују се у многим ситуацијама, не само социјалним. Могу се упаривати радници са радним местима, машине са деловима, групе студената са учионицама, итд.

Проблем налажења оптималног упаривања за произвољан граф је тежак проблем. У овом одељку ограничићемо се на два специјална случаја. Први од њих није тако важан — ради се о упаривању у врло густим графовима. Међутим, решење тог проблема илуструје интересантан приступ, који се може уопштити да би се дошло до решења проблема упаривања за бипартитне графове.

6.9.1. Савршено упаривање у врло густим графовима. У овом примеру размотрићемо специјалан случај проблема савршеног упаривања. Нека је $G = (V, E)$ неусмерени граф код кога је $|V| = 2n$ и степен сваког чвора је бар n. Приказаћемо алгоритам за налажење савршеног упаривања у оваквим графовима. Последица је да ако граф задовољава наведене услове, онда у њему увек постоји савршено упаривање. Користићемо индукцију по величини m упаривања. Базни случај $m = 1$ решава се формирањем упаривања величине један од произвољне гране графа. Показаћемо да се произвољно упаривање које није савршено може проширити или додавањем једне гране или заменом

једне гране двема новим гранама. У оба случаја повећава се величина упаривања за један.

Посматрајмо упаривање M у G са m грана, при чему је $m < n$. Најпре проверавамо све гране ван упаривања M да установимо да ли се нека од њих може додати у M. Ако пронађемо такву грану, проблем је решен — нађено је веће упаривање. У противном, M је максимално упаривање. Ако M није савршено упаривање, постоје бар два неупарена чвора v_1 и v_2. Из та два чвора по претпоставци излази најмање $2n$ грана. Све те гране воде ка упареним чворовима (у противном би се у упаривање могла додати нова грана, супротно претпоставци да је оно максимално). Пошто у M има мање од n грана, а из v_1 и v_2 излази бар $2n$ грана, у упаривању M постоји грана (u_1, u_2) која је суседна са ("покрива") бар три гране из v_1 и v_2. Претпоставимо, без смањења општости, да су то гране (u_1, v_1), (u_1, v_2) и (u_2, v_1), видети слику 28(а). Лако је видети да се уклањањем гране (u_1, u_2) из M, и додавањем двеју нових грана (u_1, v_2) и (u_2, v_1) добија веће упаривање, слика 28(б).

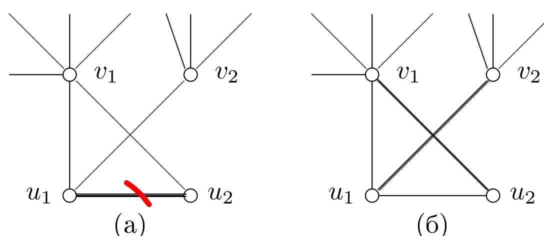
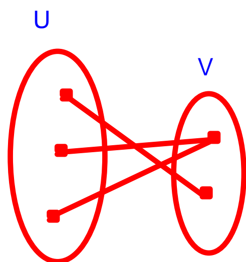


Рис. 28. Проширивање упаривања.

Описани алгоритам је још један пример *похлепног* приступа. У сваком кораку проширивања упаривања за једну грану укључено је највише три гране. У овој ситуацији је то било довољно; међутим, у општем случају је налажење доброг упаривања тежи проблем. Укључивање једне гране у упаривање утиче на избор других грана чак и у удаљеним деловима графа. Показаћемо сада како се овај приступ може применити на други специјални случај проблема упаривања.



6.9.2. Бипартитно упаривање. Нека је $G = (V, E, U)$ бипартитни граф у коме су V и U дисјунктни скупови чворова, а E је скуп грана које повезују неке чворове из V са неким чворовима из U.

Проблем. Пронаћи упаривање са максималним бројем грана у бипартитном графу G.

Један од начина да се формулише проблем је следећи: V је скуп девојака, U је скуп младића, а E је скуп "потенцијалних" парова. Циљ је под овим условима оформити што већи број парова младића и девојака.

Директан приступ је формирати парове у складу са неком стратегијом, до тренутка кад даља упаривања више нису могућа — у нади да ће нам стратегија

обезбедити оптимално или решење блиско оптималном. Може се, на пример, покушати са похлепним приступом, упарујући најпре чворове малог степена, у нади да ће преостали чворови и у каснијим фазама имати неупарене партнере. Другим речима, најпре упарујемо стидљиве особе, оне са мање познанстава, а о осталима бринемо касније. Уместо да се бавимо анализама оваквих стратегија (што је тежак проблем), покушаћемо са приступом коришћеним код претходног проблема. Претпоставимо да се полази од максималног упаривања, које не мора бити оптимално упаривање. Можемо ли га некако поправити? Погледајмо пример на слици 29(а), на коме је упаривање приказано подебљаним гранама. Јасно је да се упаривање може повећати заменом гране $2A$ са две гране $1A$ и $2B$. Ово је трансформација слична оној коју смо применили у претходном проблему. Међутим, не морамо се ограничити заменама једне гране двема гранама. Ако пронађемо сличну ситуацију у којој се неких k грана могу заменити са $k+1$ грана, добијамо алгоритам већих могућности. На пример, упаривање се може даље повећати заменом грана $3D$ и $4E$ са три гране $3C$, $4D$ и $5E$, слика 29(б).

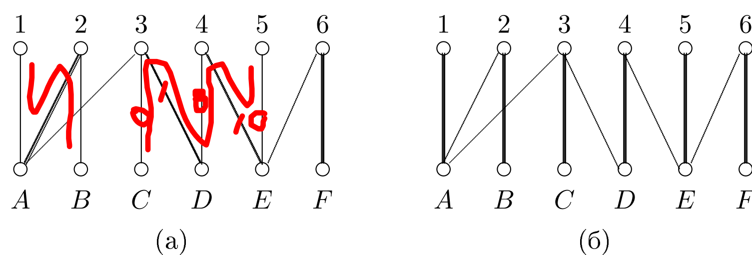


Рис. 29. Проширивање бипартитног упаривања.

Размотримо детаљније ове трансформације. Циљ је повећати број упарених чворова. Полазимо од неупареног чвора v и покушавамо да га упаримо. Пошто полазимо од максималног упаривања, сви суседи чвора v су већ упарени; због тога смо принуђени да из упаривања уклонимо неку од грана које "покривају" суседе v. Претпоставимо да смо изабрали чвор u, суседан са v, који је претходно био упарен са чвором w, на пример. Раскидамо упаривање u са w, и упарујемо v са u. Сада преостаје да пронађемо пара за чвор w. Ако је w повезан граном са неким неупареним чвором, онда смо постигли циљ; такав је био први од горњих случајева. Ако то није случај, онда настављамо даље са раскидањем парова и формирањем нових парова. Да бисмо на основу ове идеје конструисали алгоритам, потребно је да урадимо две ствари: да обезбедимо да се процедура увек завршава, и да покажемо да ако је побољшање могуће, онда ће га процедура сигурно пронаћи. Најпре ћемо формализовати наведену идеју.

Алтернирајући пут P за дато упаривање M је пут од неупареног чвора $v \in V$ до неупареног чвора $u \in U$, при чему су гране пута P наизменично у $E \setminus M$, односно M. Другим речима, прва грана (v, w) пута P не припада M (јер је v неупарен), друга грана (w, x) припада M, и тако даље до последње гране (z, u) пута P која не припада M. Запазимо да су управо алтернирајући

путеви у горњим примерима омогућавали повећавање упаривања. Број грана на путу P мора бити непаран, јер P полази из V и завршава у U. Поред тога, међу гранама пута P, грана у $E \setminus M$ има за једну више од грана у M. Према томе, ако из упаривања избацимо све гране P које су у M, а укључимо све гране P које су у $E \setminus M$, добићемо ново упаривање са једном граном више. На пример, први алтернирајући пут коришћен за повећање упаривања на слици 29(а)) био је $(1A, A2, 2B)$, и он је омогућио замену гране $A2$ гранама $1A$ и $2B$; други алтернирајући пут $(C3, 3D, D4, 4E, E5)$ омогућио је замену грана $3D$ и $4E$ гранама $C3$, $D4$ и $E5$.

Јасно је да ако за дато упаривање M постоји алтернирајући пут, онда M не може бити оптимално упаривање. Испоставља се да је и обрнуто тврђење тачно.

Теорема 6.1 (Теорема о алтернирајућем путу). *Упаривање је оптимално ако и само ако нема алтернирајућих путева.*

Доказ ће бити дат као последица општијег тврђења у следећем одељку.

Теорема о алтернирајућем путу директно сугерише алгоритам, јер произвољно упаривање које није оптимално има алтернирајући пут, а алтернирајући пут даје повећано упаривање. Започињемо са похлепним алгоритмом, додајући гране у упаривање све док је то могуће. Онда прелазимо на тражење алтернирајућих путева и повећавање упаривања, све до тренутка кад више нема алтернирајућих путева. Добијено упаривање је тада оптимално. Пошто алтернирајући пут повећава упаривање за једну грану, а у упаривању има највише $n/2$ грана (где је n број чворова), број итерација је највише $n/2$. Преостаје још један проблем — како проналазити алтернирајуће путеве? Он се може решити на следећи начин. Трансформишемо неусмерени граф G у усмерени граф G' усмеравајући гране из M од U ка V, а гране из $E \setminus M$ од V ка U. Слика 30(а) приказује полазно максимално упаривање за граф са слике 29(а), а слика 30(б) приказује одговарајући усмерени граф G'. Алтернирајући пут у G тада одговара усмереном путу од неупареног чвора у V до неупареног чвора у U. Такав усмерени пут може се пронаћи било којим поступком обиласка графа, нпр. помоћу DFS. Сложеност обиласка (претраге) је $O(|V| + |E|)$, па је сложеност алгоритма $O(|V|(|V| + |E|))$.

6.9.2.1. *Побољшање.* Пошто комплетан обилазак графа у најгорем случају може да траје колико и налажење једног пута, може се покушати са налажењем више алтернирајућих путева једном претрагом. Потребно је, међутим, да будемо сигурни да ови путеви не мењају један другог. Један начин да се гарантује независност таквих алтернирајућих путева је увођење ограничења да скупови чворова на појединим путевима буду дисјунктни. Ако су путеви дисјунктни, онда утичу на упаривање различитих чворова, па се могу истовремено искористити. Нови, побољшани алгоритам за налажење алтернирајућих путева је следећи. Најпре примењујемо BFS на граф G' од скупа неупарених чворова у V, слој по слој, до слоја у коме су пронађени неупарени чворови из U. Затим из графа индукованог претрагом у ширину вадимо *максимални* скуп дисјунктних

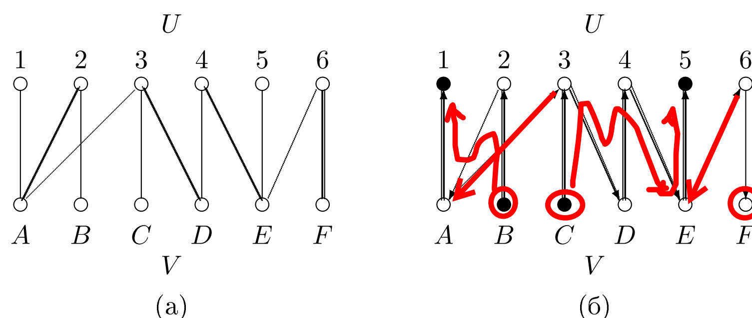


Рис. 30. Налажење алтернирајућих путева

путева у G', којима одговарају алтернирајући путеви у G. То се изводи проналажењем првог пута, уклањањем његових чворова, проналажењем наредног пута, уклањањем његових чворова, итд. (резултат није *оптимални*, него само максимални скуп). Бирамо максимални скуп, да бисмо после претраге добили што веће упаривање; сваки нови дисјунктни пут повећава упаривање за једну грану. На крају повећавамо упаривање коришћењем пронађеног скупа дисјунктних путева. Процес се наставља све док је могуће пронаћи алтернирајуће путеве, односно док је у графу G' неки неупарени чвор из V достижан из неког неупареног чвора из U.

Сложеност. Испоставља се да је у побољшаном алгоритму број итерација $O(\sqrt{|V|})$ у најгорем случају (Хопкрофт и Карп 1973, Hopcroft, Carp ; ово тврђење дајемо без доказа). Укупна временска сложеност алгоритма је дакле $O((|V|+|E|)\sqrt{|V|})$.

6.10. Транспортне мреже

Проблем транспортне мреже је један од основних проблема у теорији графова и комбинаторној оптимизацији. Интензивно је проучаван више од 40 година, па су за њега развијени многи алгоритми и структуре података. Проблем има много варијанти и уопштења. Основна варијанта проблема транспортне мреже може се формулисати на следећи начин. Нека је $G = (V, E)$ усмерени граф са два посебно издвојена чвора, s (извор), са улазним степеном 0, и t (понор) са излазним степеном 0. Свакој грани $e \in E$ придружена је позитивна тежина $c(e)$, **капацитет** гране e. Капацитет гране је мера тока који може бити пропуштен кроз грану. За овакав граф кажемо да је **мрежа**. Због удобности се непостојећим гранама додељује капацитет 0. **Ток** је функција f дефинисана на E која задовољава следеће услове:

(1) $0 \le f(e) \le c(e)$: ток кроз произвољну грану не може да премаши њен капацитет;

(2) за све $v \in V \setminus \{s, t\}$ је $\sum_u f(u, v) = \sum_w f(v, w)$: укупан ток који улази у произвољни чвор v једнак је укупном току који излази из њега ("нестишљивост", закон очувања, односно конзервације тока).

Ова два услова имају за последицу да је укупан ток који излази из s једнак укупном току који улази у t. У то се можемо уверити на следећи начин. Увешћемо најпре појам **пресека**. Нека је A произвољан подскуп V такав да садржи s, а не садржи t. Означимо са $B = V \setminus A$ скуп преосталих чворова. Пресек одређен скупом A је скуп грана $(v, u) \in E$ таквих да $v \in A$ и $u \in B$. Интуитивно, пресек је скуп грана које раздвајају s од t. Индукцијом по броју чворова у A лако се показује да укупан ток кроз пресек не зависи од A. Специјално, за $A = \{s\}$ пресек обухвата гране које излазе из s, а за $A = V \setminus \{t\}$ пресек чине гране које улазе у t. Према томе, укупан ток који излази из s једнак је укупном току који улази у t. Проблем који нас занима је максимизирање тока. У случају кад су капацитети грана реални бројеви, није очигледно чак ни да максимални ток увек постоји; показаћемо да он увек постоји. Један начин да се описани проблем схвати као реалан физички проблем, је да замислимо да мрежу чине цеви за воду. Свака цев има свој капацитет, а услови које ток треба да задовољи су природни. Циљ је "протерати" кроз мрежу што већу количину воде у јединици времена.

Показаћемо најпре да се проблем бипартитног упаривања може свести на проблем мрежног протока. То може на први поглед да изгледа бескорисно, пошто већ знамо да решимо проблем упаривања, а не знамо решење проблема транспортне мреже — редукција је дакле у погрешном смеру. Међутим, за решавање проблема транспортне мреже може се применити поступак сличан поступку за налажење бипартитног упаривања.

Задатом бипартитном графу $G = (V, E, U)$ у коме треба пронаћи упаривање са највећим могућим бројем грана (оптимално упаривање) додајемо два нова чвора s и t, повезујемо s гранама са свим чворовима из V, а све чорове из U повезујемо са t. Означимо добијени граф са G' (видети слику 31, на којој су све гране усмерене слева удесно). Пошто свим гранама доделимо капацитет 1, добијамо регуларан проблем транспортне мреже на графу G'. Нека је M неко упаривање у G. Упаривању M може се на природан начин придружити ток у G'. Додељујемо ток 1 свим гранама из M и свим гранама које s или t повезују са упареним чворовима. Свим осталим гранама додељујемо ток 0. Укупан ток једнак је тада броју грана у упаривању M. Може се показати да је M оптимално упаривање ако и само ако је одговарајући целобројни ток у G' оптималан. У једном смеру доказ је једноставан: ако је ток оптималан и одговара упаривању, онда се не може наћи веће упаривање, јер би му одговарао већи ток. Да бисмо извели доказ у другом смеру, потребно је да некако применимо идеју алтернирајућих путева на ток у транспортној мрежи, и да покажемо да ако нема алтернирајућих путева, онда је одговарајући ток оптималан.

Повећавајући пут у односу на задати ток f је усмерени пут од s до t, који се састоји од грана из G, не обавезно у истом смеру; свака од тих грана (v, u) треба да задовољи тачно један од следећа два услова:

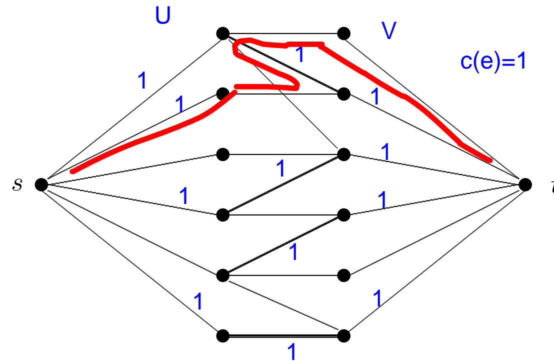


Рис. 31. Свођење бипартитног упаривања на транспортне мреже (све гране усмерене су слева удесно).

(1) (v, u) има исти смер као и у G, и $f(v, u) < c(v, u)$. У том случају грана (v, u) је **директна грана** Директна грана има капацитет већи од тока, па се може повећати ток кроз њу. Разлика $c(v, u) - f(v, u)$ зове се **слек** те гране.

(2) (v, u) има супротан смер у G, и $f(v, u) > 0$. У овом случају грана (v, u) је **повратна грана**. Део тока из повратне гране може се "позајмити".

Повећавајући пут је уопштење алтернирајућег пута, и има исти смисао за транспортне мреже као алтернирајући пут за бипартитно упаривање. Ако постоји повећавајући пут у односу на ток f (односно f **дозвољава** повећавајући ток), онда f није оптимални ток. Ток f може се повећати повећавањем тока кроз повећавајући пут на следећи начин. Ако су све гране повећавајућег пута директне гране, онда се кроз њих може повећати ток, тако да сва ограничења и даље остану задовољена. Повећање тока је у овом случају тачно једнако минималном слеку међу гранама пута. Случај повратних грана је нешто сложенији, видети слику 32. Свака грана означена је са два броја a/b, при чему је a капацитет, а b тренутни ток. Јасно је да се укупан ток не може директно повећати, јер не постоји пут од s до t који се састоји само од директних грана. Ипак, постоји начин да се укупан ток повећа.

Пут $s - v - u - w - t$ је повећавајући пут. Допунски ток 2 може се спровести до u од s (2 је минимални слек на директним гранама до u). Ток 2 може се *одузети* од $f(w, u)$. Тиме се постиже задовољење услова (2) (закона очувања тока) за чвор u, јер је u имао повећање тока за 2 из повећавајућег пута, а затим смањење дотока за 2 из повратне гране. У чвору w је сада излазни ток смањен за 2, па га треба повећати кроз неку излазну грану. Са "протеривањем" тока може се наставити на исти начин од w, повећавањем тока кроз директне гране и смањивањем тока кроз повратне гране. У овом случају постоји само још једна директна грана (w, t) која достиже t, и проблем је решен. Пошто само директне гране могу да излазе из s, односно да улазе у t, укупан ток је повећан. Повећање је једнако мањем од следећа два броја: минималног слека директних

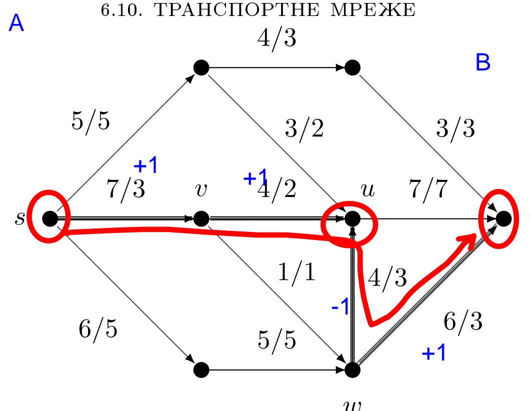


Рис. 32. Пример мреже са повећавајућим путем.

грана, односно минималног тока повратних грана. На слици 33 приказана је иста мрежа са промењеним током; нови ток је уствари оптималан.

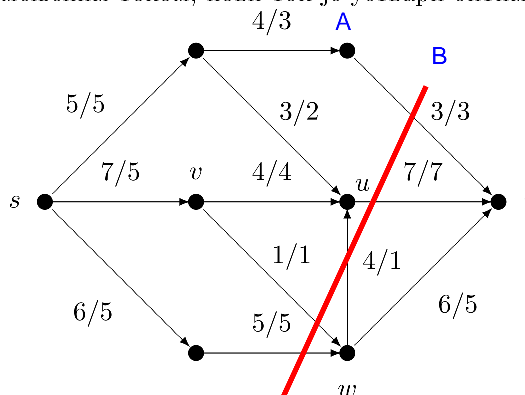


Рис. 33. Резултат повећавања тока у мрежи са слике 32.

Из реченог следи да ако у мрежи постоји повећавајући пут, онда ток није оптималан. Обрнуто је такође тачно.

Теорема 6.2 (Теорема о повећавајућем путу)**.** *Ток кроз мрежу је оптималан ако и само ако у односу на њега не постоји повећавајући пут.*

Доказательство. Доказ у једном смеру смо већ видели — ако у мрежи постоји повећавајући пут, онда ток није оптималан. Претпоставимо сада да у односу на ток f не постоји ни један повећавајући пут, и докажимо да је тада f оптимални ток. За произвољан пресек (одређен скупом A, $s \in A$, $t \in B \equiv V \setminus A$) дефинишемо капацитет, као збир капацитета његових грана које воде из неког чвора A у неки чвор B. Јасно је да ни један ток не може бити већи од капацитета произвољног пресека. Заиста, укупан ток из s једнак је збиру

токова кроз гране пресека од A ка B, умањеном за ток кроз гране пресека од B ка A, па је мањи или једнак од збира капацитета грана које воде од A ка B, односно од капацитета пресека. Према томе, ако пронађемо ток са вредношћу једнаком капацитету неког пресека, онда је тај ток оптималан. Са доказом настављамо у том правцу: показаћемо да ако у односу на ток не постоји повећавајући пут, онда је укупан ток једнак капацитету неког пресека, па дакле оптималан.

Нека је f ток у односу на који не постоји повећавајући пут. Нека је $A \subset V$ скуп чворова v таквих да у односу на ток f постоји повећавајући пут од s до v (прецизније, постоји пут од s до v такав да на њему за све директне гране e важи $f(e) < c(e)$, а за све повратне гране e' важи $f(e') > 0$). Јасно је да $s \in A$ и $t \notin A$, јер по претпоставци за f не постоји повећавајући пут. Према томе, A дефинише пресек. Тврдимо да за све гране (v, w) тог пресека важи $f(v, w) = c(v, w)$ ако је $v \in A$, $w \in B$ ("директне" гране), односно $f(v, w) = 0$ ако је $v \in B$, $w \in A$ ("повратне гране"). Заиста, у противном би директна грана (v, w) продужавала повећавајући пут до чвора $w \notin A$, супротно претпоставци да такав пут постоји само до чворова из A. Слично, повратна грана (v, w) продужавала би повећавајући пут до чвора $v \notin A$. Дакле, укупан ток једнак је капацитету пресека одређеног скупом A, па је ток f оптималан. □

Доказали смо следећу важну теорему.

Теорема 6.3 (Теорема о максималном току и минималном пресеку)**.** *Оптимални ток у мрежи једнак је минималном капацитету пресека.*

Теорема о повећавајућем путу има за последицу и следећу теорему.

Теорема 6.4 (Теорема о целобројном току)**.** *Ако су капацитети свих грана у мрежи целобројни, онда постоји оптимални ток са целобројном вредношћу.*

Доказательство. Тврђење је последица теореме о повећавајућем путу. Сваки алгоритам који користи само повећавајуће путеве доводи до целобројног тока ако су сви капацитети грана целобројни. Ово је очигледно, јер се може кренути од тока 0, а онда се укупан ток после сваке употребе повећавајућег пута повећава за цели број. До истог закључка долази се и на други начин: капацитет сваког пресека је целобројан, па и минималног. □

Вратимо се сада на проблем бипартитног упаривања. Јасно је да сваки алтернирајући пут у G одговара повећавајућем путу у G', и обрнуто. Последица теореме о повећавајућем путу је теорема о алтернирајућем путу из претходног одељка. Ако је M оптимално упаривање, онда за њега не постоји алтернирајући пут, па у G' не постоји повећавајући пут, а одговарајући ток је оптималан. С друге стране, постоји оптимални целобројни ток, и он мора да одговара упаривању, јер је сваки чвор у V повезан само једном граном (са капацитетом 1) са s; због тога, укупан ток кроз сваки чвор из V може да буде највише 1. Исто важи и за чворове из скупа U. Ово упаривање мора бити оптимално, јер ако би се могло повећати, онда би постојао већи укупни ток.

Теорема о повећавајућем путу непосредно се трансформише у алгоритам. Полази се од тока 0, траже се повећавајући путеви, и на основу њих повећава се ток, све до тренутка кад повећавајући путеви више не постоје. Тражење повећавајућих путева може се извести на следећи начин. Дефинишемо **резидуални граф** у односу на мрежу $G = (V, E)$ и ток f, као мрежу $R = (V, F)$ са истим чворовима, истим извором и понором, али промењеним скупом грана и њихових тежина. Сваку грану $e = (v, w)$ са током $f(e)$ замењујемо са највише две гране $e' = (v, w)$ (ако је $f(e) < c(e)$; капацитет e' једнак је слеку гране e: $c(e') = c(e) - f(e)$), односно $e'' = (w, v)$ (ако је $f(e) > 0$; капацитет e'' је $c(e'') = f(e)$). Ако се на овај начин добију две паралелне гране, замењују се једном, са капацитетом једнаком збиру капацитета паралелних грана. На слици 34 приказан је резидуални граф за мрежу са слике 32, у односу на ток задат на тој слици. Гране резидуалног графа одговарају могућим гранама повећавајућег пута. Њихови капацитети одговарају могућем повећању тока кроз те гране. Према томе, повећавајући пут је обичан усмерени пут од s до t у резидуалном графу. Конструкција резидуалног графа захтева $O(|E|)$ корака, јер се свака грана проверава тачно једном.

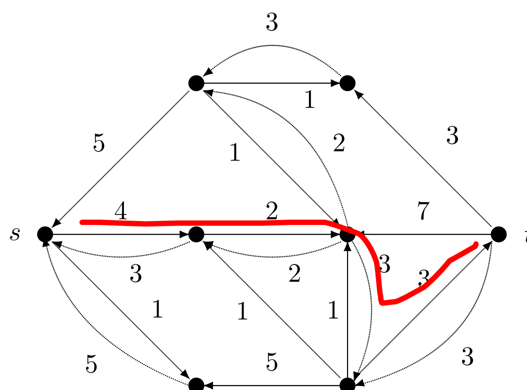


Рис. 34. Резидуални граф мреже са слике 32 у односу на ток дефинисан на тој слици.

На несрећу, избор повећавајућег пута на произвољан начин може се показати врло неефикасним. Време извршења таквог алгоритма у најгорем случају може да чак и не зависи од величине графа. Посматрајмо мрежу на слици 35. Оптимални ток је очигледно $2M$. Међутим, могли бисмо да кренемо од повећавајућег пута $s - a - b - t$ кроз који се ток може повећати само за 1. Затим бисмо могли да изаберемо повећавајући пут $s - b - a - t$ који опет повећава ток само за 1. Процес може да се понови укупно $2M$ пута, где M може бити врло велико, без обзира што граф има само четири чвора и пет грана. Пошто се вредност M може представити са $O(\log M)$ бита, сложеност овог алгоритма је у најгорем случају експоненцијална функција величине улаза (број M је део улаза).

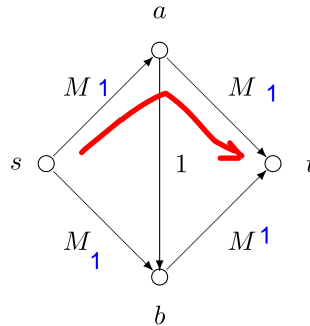


Рис. 35. Пример мреже на којој тражење повећавајућих путева може бити неограничено неефикасно.

Горе наведена могућност је врло непожељна, али се може избећи. Едмондс и Карп (Edmonds, Carp) су 1972. године показали да ако се међу могућим повећавајућим путевима увек бира онај са најмањим бројем грана, онда је број повећавања највише $(|V|^3 - |V|)/4$. То води алгоритму који је у најгорем случају полиномијалан у односу на величину улаза. Од тог времена предложено је више различитих алгоритама, сложенијих и мање сложених, а више њих достиже горњу границу сложености $O(|V|^3)$ у најгорем случају. Овде их нећемо описивати.

6.11. Хамилтонови циклуси

Ово поглавље започели смо разматрањем циклуса који садрже све гране у графу. Завршићемо је освртом на циклусе који садрже све чворове графа. Ово је такође познати проблем, који је име добио по ирском математичару Вилијему Хамилтону (W. R. Hamilton), који је предложио популарну игру засновану на овом проблему 1857. године.

Проблем. Задат је граф $G = (V, E)$. Пронаћи у G прости циклус који сваки чвор из V садржи тачно једном.

Такав циклус зове се **Хамилтонов циклус**. Графови који садрже такве циклусе зову се **Хамилтонови графови**. Проблем има усмерену и неусмерену верзију; ми ћемо се бавити само неусмереном варијантом.

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 36 имају по 16 чворова степена 3, али је први Хамилтонов, а други очигледно није. Овај проблем спада у класу NP-комплетних проблема, који ће бити разматрани у поглављу 11. У овом одељку приказаћемо једноставан поступак налажења Хамилтоновог циклуса у специјалној класи врло густих графова.

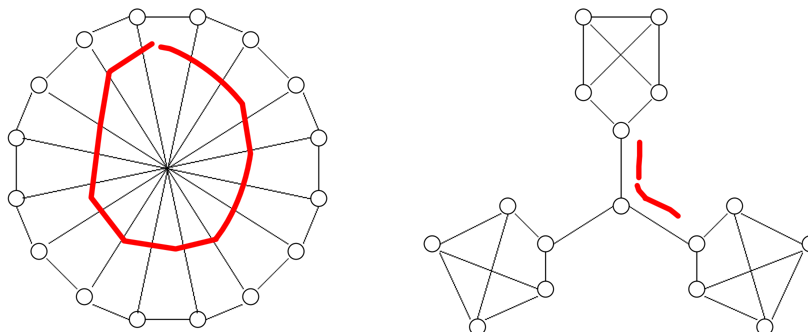


Рис. 36. Два графа са по 16 чворова степена 3, од којих је први Хамилтонов, а други није.

Нека је $G = (V, E)$ повезан неусмерен граф и нека за произвољан чвор $v \in V$ $d(v)$ означава његов степен. Следећи проблем односи се на налажење Хамилтоновог циклуса у врло густим графовима. Показаћемо да услови проблема гарантују да је граф Хамилтонов.

Проблем. Дат је повезан неусмерен граф $G = (V, E)$ са $n \geq 3$ чворова, такав да сваки пар несуседних чворова v и w задовољава услов $d(v)+d(w) \geq n$. Пронаћи у G Хамилтонов циклус.

Алгоритам се заснива на индукцији по броју грана које треба уклонити из комплетног графа да би се добио задати граф. База индукције је комплетан граф. Сваки комплетан граф са бар три чвора садржи Хамилтонов циклус, који је лако пронаћи.

Индуктивна хипотеза. Умемо да пронађемо Хамилтонов циклус у графовима који задовољавају наведене услове ако имају бар $n(n-1)/2-m$ грана.

Сада треба да покажемо како пронаћи Хамилтонов циклус у графу са $n(n-1)/2-(m+1)$ грана који задовољава услове проблема. Нека је $G = (V, E)$ такав граф. Изаберимо произвољна два несуседна чвора v и w у G (то је могуће ако граф није комплетан), и посматрајмо граф G' који се од G добија додавањем гране (v, w). Према индуктивној хипотези ми умемо да пронађемо Хамилтонов циклус у графу G'. Нека је $x_1, x_2, \ldots, x_n, x_1$ такав циклус у G' (видети слику 37). Ако грана (v, w) није део циклуса, онда је исти циклус део графа G, па је проблем решен. У противном, без смањења општости може се претпоставити да је $v = x_1$ и $w = x_n$. Према датим условима је $d(v)+d(w) \geq n$. Потребно је у графу пронаћи нови Хамилтонов циклус.

Посматрајмо све гране графа G суседне са чворовима v или w. Тврдимо да под задатим условима постоје два чвора x_i и x_{i+1} таква да у G постоје гране (w, x_i) и (v, x_{i+1}). Да бисмо то доказали, претпоставимо супротно, да ни за једно i, $2 \leq i \leq n-2$, не постоје истовремено обе гране (w, x_i) и (v, x_{i+1}). Нека је чвор w везан са чвором x_{n-1} и k чворова $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$, при чему

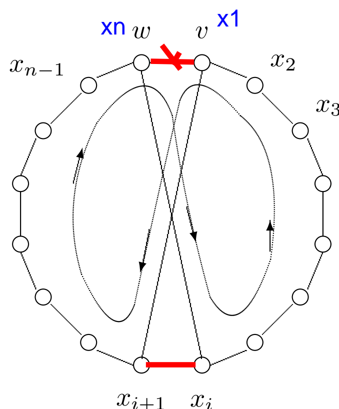


Рис. 37. Модификација Хамилтоновог циклуса после избацивања гране (v, w).

је $2 \leq i_1 < i_2 < \cdots < i_k \leq n-2$. Тада чвор v не сме бити везан ни са једним од чворова $x_{i_1+1}, x_{i_2+1}, \ldots, x_{i_k+1}$, па је $d(v) \leq n-2-k$ (од укупно $n-1$ могућих грана из v не постоје гране ка w, нити ка k наведених чворова, различитих од w). Због тога је $d(w) + d(v) \leq (k+1) + (n-2-k) = n-1$; ово је у контрадикцији са претпоставком да је $d(w) + d(v) \geq n$. Дакле, за неко i постоје гране (w, x_i) и (v, x_{i+1}). Од ових двеју грана може се формирати нови Хамилтонов циклус $v(= x_1), x_{i+1}, x_{i+2}, \ldots, w(= x_n), x_i, x_{i-1}, \ldots, v$, који не садржи грану (v, w), видети слику 37.

Реализација. Директна примена овог доказа полази од комплетног графа, из кога се једна за другом избацују гране које не припадају задатом графу. Боље је решење почети са много мањим графом, на следећи начин. У датом графу G најпре проналазимо дугачак пут (на пример, помоћу DFS), а онда додајемо нове гране тако да пут продужимо до Хамилтоновог циклуса. Тако је добијен већи граф G'. Обично је довољно додати само неколико грана. Међутим, чак и у најгорем случају биће додата највише $n-1$ грана. Полазећи од G', доказ теореме се затим примењује итеративно, све док се не пронађе Хамилтонов циклус у G. Укупан број корака за замену једне гране је $O(n)$. Потребно је заменити највише n грана, па је временска сложеност алгоритма $O(n^2)$.

6.12. Резиме

Графови се користе као модели за односе међу паровима објеката. Пошто већина алгоритама захтева испитивање комплетног улаза, први проблем на који се обично наилази је обилазак графа. Анализирали смо два типа обиласка графова: *претрагу у дубину* (DFS) и *претрагу у ширину* (BFS). Видели смо неколико примера код којих је DFS погоднија од BFS. Може се дакле препоручити да се увек најпре покуша са DFS (иако постоји много примера

где је BFS супериорнија). DFS је посебно погодна за рекурзивне алгоритме на графовима. BFS обично захтева више простора (што опет није правило — зависи од графа). Видели смо такође пример *претраге са приоритетом*, која је коришћена за израчунавање дужина најкраћих путева од задатог чвора. Претрага са приоритетом је сложенија од обичне претраге. Она је корисна код оптимизационих проблема са тежинским графовима.

Циклуси често представљају проблем у графовским алгоритмима. Обично су због тога алгоритми за стабла или усмерене ацикличке графове једноставнији за реализацију и бржи.

Друга корисна техника код графова је *редукција* (свођење). Пошто графови могу бити представљени матрицама, постоји природна повезаност између графовских и алгоритама са матрицама. Проблеми транспортних мрежа и упаривања су област у којој се редукције широко примењују. У поглављу 11 разматраћемо класу проблема, такозване NP-комплетне проблеме, који се вероватно не могу решити алгоритмима сложености која је полиномијална функција од величине улаза у најгорем случају. Та класа садржи велики број графовских проблема. Разлике између лаких и тешких проблема понекад су веома мале. Битно је разумети их и развити интуитивни осећај за те разлике. Због тога је материјал из поглавља 11 јако важан за разумевање графовских алгоритама.

Задаци

6.1. Решити проблем одређивања фактора равнотеже чворова у бинарном стаблу из одељка 4.7 помоћу DFS (потребно је само прецизирати узлану и излазну обраду).

6.2. Конструисати алгоритам који за дати усмерени граф $G = (V, E)$ и његов чвор $v \in V$ испитује да ли је G коренско стабло са кореном v (сматра се да су у коренском стаблу гране усмерене од корена ка листовима).

6.3. Дат је повезан неусмерен граф $G = (V, E)$, његово повезујуће стабло T и чвор $v \in V$. Конструисати алгоритам сложености $O(|E| + |V|)$ који треба да установи да ли је T регуларно DFS стабло графа G са кореном v. Другим речима, треба установити може ли T бити излаз DFS из v за неко уређење грана.

6.4. Окарактерисати неусмерене графове у којима за неки чвор v постоји DFS стабло са кореном у v, које је идентично са BFS стаблом са кореном v (два стабла су подударна ако су им једнаки скупови грана — редослед обиласка тих грана овде није битан).

6.5. Модификовати алгоритам за тополошко сортирање ацикличког графа (слика 16) тако да за произвољан задати граф $G = (V, E)$, ако је G ациклички, онда алгоритам даје његово тополошко уређење чворова, а у противном проналази неки циклус у G. Сложеност алгоритма треба да буде $O(|E| + |V|)$.

6.6. Нека је $T = (V, F)$ подграф који се састоји од свих грана на најкраћим путевима од чвора v у графу $G = (V, E)$. Показати да је T коренско стабло са кореном v.

6.7. Нека је $G = (V, E)$ неусмерени тежински граф и нека је T његово стабло најкраћих путева од чвора $v \in V$. Ако се тежине грана G повећају за једну исту константу c, да ли је T и даље стабло најкраћих путева од v?

6.8. Доказати или оповргнути контрапримером: алгоритам *Najkr_putevi* (слика 20) ради коректно за тежинске графове чије неке гране имају негативне тежине, под условом да не постоји циклус негативне тежине.

6.9. Променити алгоритам MCST (слика 23) тако да проналази повезујуће стабло максималне цене у графу.

6.10. (а) Навести пример повезаног неусмереног тежинског графа $G = (V, E)$ и чвора $v \in V$ тако да је MCST графа G истовремено и стабло најкраћих путева од v. (б) Навести пример повезаног неусмереног тежинског графа $G = (V, E)$, тако да се MCST графа G и стабло најкраћих путева у G од v разликују. (ц) Да ли је могуће да ова два стабла немају заједничких грана?

6.11. Конструисати алгоритам сложености $O(|E| + |V|)$ за налажење савршеног упаривања у врло густом графу, тј. графу са $2n$ чворова таквом да за свака два његова несуседна чвора u и v важи $d(u) + d(v) \geq n$, одељак 6.11.

6.12. Наћи потребан и довољан услов да граф G са n чворова има следећу особину. Постоји низ чворова $v_1, v_2, \ldots, v_n$, тако да је за $i = 1, 2, \ldots, n$ степен чвора v_i у графу G_i, који се од G добија избацивањем чворова $v_1, v_2, \ldots, v_{i-1}$ заједно са гранама које су им суседне, једнак 1.

6.13. Конструисати алгоритам линеарне временске и просторне сложености за налажење Ојлеровог пута у Ојлеровом графу.

6.14. Нека је $G = (V, E)$ неусмерени граф у коме сваки чвор има паран степен. Конструисати алгоритам линеарне временске сложености за усмеравање грана у G, тако да улазни степен сваког чвора буде једнак његовом излазном степену.

6.15. Усмерени Ојлеров циклус је усмерени циклус који сваку грану усмереног графа садржи тачно једном. Доказати да усмерени граф садржи усмерени Ојлеров циклус акко је улазни степен сваког чвора једнак његовом излазном степену, а одговарајући неусмерени граф (добијен претварањем усмерених у неусмерене гране) је повезан.

6.16. Нека је $G = (V, E)$ неусмерени повезан граф са k чворова непарног степена. (а) Доказати да је k паран број. (б) Конструисати алгоритам за налажење $k/2$ отворених путева таквих да је свака грана G садржана у тачно једном од тих путева.

6.17. Нека је дат неусмерени повезан граф G. Конструисати алгоритам линеарне временске сложености за налажење таквог чвора у G чијим уклањањем се G не распада на две или више компоненте повезаности.

6.18. **Бинарни де Бруијнов низ** је (циклички) низ од $N = 2^n$ бита $a_0 a_1 \ldots a_{N-1}$ такав да се сваки бинарни блок s од n бита налази негде у овом низу. Другим речима, постоји јединствени индекс i такав да је $s = a_i a_{i+1} \ldots a_{i+n-1}$ (где се индекси рачунају по модулу N). На пример, низ 11010001 је бинарни де Бруијнов низ за $n = 3$. Нека је $G_n = (V, E)$ усмерени граф дефинисан на следећи начин. Скуп чворова V је скуп свих бинарних блокова од $n - 1$ бита ($|V| = 2^{n-1}$). Из чвора $b_1 b_2 \ldots b_{n-1}$ води грана ка чвору $c_1 c_2 \ldots c_{n-1}$ акко је $b_2 b_3 \ldots b_{n-1} = c_1 c_2 \ldots c_{n-2}$. Доказати да је G_n усмерени Ојлеров граф и објаснити смисао те чињенице за де Бруијнове низове.

6.19. Дато је n природних бројева $d_1, d_2, \ldots, d_n$ којима је збир $2n - 2$. Конструисати стабло са n чворова, којима су степени $d_1, d_2, \ldots, d_n$.

6.20. Нека је $(u_1, i_1), (u_2, i_2), \ldots, (u_n, i_n)$, низ парова целих ненегативних бројева таквих да је $u_1 = 0$, $u_k = 1$ за $1 < k \leq n$, $i_1 > 0$ и $\sum_{j=1}^{n} i_j = n-1$. Конструисати коренско стабло са n чворова такво да је улазни, односно излазни степен чвора k једнак u_k, односно i_k. Сложеност алгоритма треба да буде $O(n)$.

6.21. Нека је $G = (V, E)$ усмерени граф (не обавезно ациклички). Конструисати ефикасан алгоритам који нумерише чворове графа бројевима од 1 до $|V|$, тако да редни број сваког чвора v буде мањи од редног броја бар једног чвора w таквог да постоји грана (w, v) (ако има таквих чворова w), или утврђује да таква нумерација не постоји.

6.22. За неусмерени граф $G = (V, E)$ каже се да је k-обојив ако се сви чворови G могу обојити са k различитих боја, тако да не постоје два суседна чвора обојена истом бојом. Конструисати алгоритам линеарне сложености за бојење графа са две боје, односно за утврђивање чињенице да граф није 2-обојив (или бипартитан!).

6.23. Нека је $G = (V, E)$ 2-обојив неусмерени граф (видети задатак 6.22). Показати да је бојење G са две боје *јединствено* (до на пермутацију боја) акко је G повезан граф.

6.24. Нека је T неусмерено стабло (у општем случају не бинарно) са кореном r, задато листом повезаности. Сваком чвору T придружен је знак из фиксираног алфабета. Нека је стринг P узорак (представљен вектором знакова). Конструисати алгоритам који проверава да ли се узорак појављује бар једном на путу од корена до неког листа. Сложеност алгоритма у најгорем случају треба да буде $O(n+m)$, где је n број чворова стабла, а m дужина узорка.

6.25. Дат је повезан неусмерени граф $G = (V, E)$ који садржи тачно један циклус. Потребно је усмерити све гране тако да улазни степени свих чворова буду највише 1 (такав граф може се назвати **инјективним**, јер одговара инјективном пресликавању $V \to V$). Конструисати алгоритам за описано усмеравање грана и оценити његову сложеност.

6.26. Нека је $G = (V, E)$ неусмерени граф. Конструисати алгоритам за оријентисање грана графа G тако да улазни степен сваког чвора буде бар 1 — ако је могуће, односно да у противном установи да је то немогуће.

6.27. Дат је неусмерени граф $G = (V, E)$. Потребно је усмерити његове гране тако да буду испуњена следећа два услова:

- добијени усмерени граф садржи усмерено коренско стабло T (тј. стабло чије су све гране усмерене од корена);
- свака грана која не припада стаблу T затвара усмерени циклус са гранама из T.

Колика је сложеност конструисаног алгоритма?

6.28. Дат је усмерени ациклички граф $G = (V, E)$. Конструисати алгоритам линеарне временске сложености за налажење најдужег усмереног простог пута у G (било ког међу њима ако их има више).

6.29. Нека је $G = (V, E)$ усмерени ациклички граф и нека је k највећи број грана неког пута у G. Конструисати алгоритам линеарне сложености који чворове графа дели у највише $k+1$ група, тако да за било која два чвора v, w из исте групе не постоји пут од v до w, ни пут од w до v.

6.30. Нека је $G = (V, E)$ усмерени граф и нека су v, w два чвора у G. Конструисати алгоритам линеарне сложености за одређивање *броја* различитих најкраћих путева од v до w (не обавезно са дисјунктним скупом чворова; гранама нису придружене тежине).

6.31. Како треба променити алгоритам $Najkr_putevi$ (слика 20) тако да му временска сложеност у најгорем случају буде $O(|V|^2)$, независно од $|E|$?

6.32. Нека је $G = (V, E)$ тежински усмерени граф. Конструисати алгоритам за одређивање циклуса минималне тежине у G.

6.33. Гранама графа $G = (\{a, b, c, d, e\}, E)$ придружене су дужине као на слици 38. Одредити дужине најкраћих путева између свака два чвора графа.

6.34. Алгоритми за налажење најкраћих путева из одељка 6.5 бирају произвољан од неколико најкраћих путева. Како треба променити те алгоритме да, ако има више различитих путева исте дужине, онда се бира онај са најмањим бројем грана? У случају да има више најкраћих путева са најмањим бројем грана, дозвољено је изабрати произвољан међу њима. Претпоставка је да су дужине грана цели бројеви.

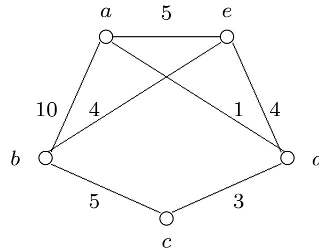


Рис. 38. Уз задатак 6.33.

6.35. Нека је $G = (V, E)$ усмерен тежински граф, при чему су све тежине грана позитивне. Нека су v и w два чвора у G и нека је $k \leq |V|$ природан број. Конструисати алгоритам за налажење најкраћег пута од v до w који се састоји од тачно k грана. Пут не мора да буде прост.

6.36. Нека је $G = (V, E)$ усмерени ациклички тежински граф коме су тежине неких грана негативне. Конструисати алгоритам линеарне временске сложености за одређивање најкраћих путева од датог чвора $v \in V$.

6.37. Наћи потребан и довољан услов да се скуп грана E неусмереног графа $G = (V, E)$ може разложити на дисјунктне подскупове $E_1, E_2, \ldots, E_r$ који одговарају простим циклусима. Конструисати ефикасан алгоритам за налажење таквог разлагања у графу G који задовољава тај услов.

6.38. Доказати да постоји неусмерен граф са n чворова и $O(n)$ грана који садржи $2^{\Omega}(n)$ различитих простих циклуса (односно бар 2^{cn} различитих циклуса за неко $c > 0$).

6.39. Дат је усмерен граф $G = (V, E)$ са $n+1$ чвором и n грана, такав да је одговарајући неусмерени граф стабло, при чему је свака грана (u, w) означена јединственим бројем $\lambda(u, w)$ из скупа $A = \{1, 2, \ldots, n\}$. Потребно је одредити функцију S која чворове пресликава у подскупове скупа A, тако да буду задовољена следећа два услова: 1) ако је $(u, w) \in E$, онда $S(w) = S(u) \cup \{\lambda(u, w)\}$, и 2) ако $u \neq w$, онда је $S(u) \neq S(w)$. При томе $S(u)$ може бити произвољан подскуп A, укључујући $\emptyset$ и A. Конструисати алгоритам за одређивање функције S.

6.40. **Језгро** усмереног графа $G = (V, E)$ је такав подскуп $V' \subseteq V$ у коме чворови нису међусобно повезани гранама, а за сваки чвор $w \notin V'$ постоји таква грана (v, w) да $v \in V'$. Задат је усмерен граф $G = (V, E)$ са $n+1$ чвором и n грана, такав да је одговарајући неусмерени граф стабло. Конструисати алгоритам који проналази језгро G или утврђује да G не садржи језгро.

6.41. Нека је $G = (V, E)$ повезан усмерен граф, такав да ако $(v, w) \in E$, онда и $(w, v) \in E$, за сваки пар $v, w \in V$. Нека је $f : E \rightarrow R$ таква функција, да за сваку грану $(v, w) \in E$ важи $f(v, w) = -f(w, v)$, а за произвољни циклус који чине гране $e_1, e_2, \ldots, e_k$ важи $\sum_{i=1}^{k} f(e_i) = 0$. Конструисати алгоритам линеарне временске сложености за одређивање функције $p : V \rightarrow R$, такве да за сваку грану $(v, w) \in E$ важи $f(v, w) = p(w) - p(v)$.

6.42. Минимално повезујуће стабло (MCST) графа може се одредити и на следећи начин. Уместо да се ради са једним стаблом коме се додаје грана по грана, може се радити са колекцијом дисјунктних стабала (која су део MCST) и повезивати их додајући једну по једну грану. На почетку су сви чворови дисјунктна стабла са по 0 грана. У сваком кораку алгоритам проналази грану најмање цене чији крајеви припадају различитим стаблима, и повезује та два стабла додајући нађену грану. Конструисати алгоритам заснован на овом приступу. Колика је сложеност алгоритма?

6.43. Нека је $G = (V, E)$ неусмерен тежински граф, и нека је F шума, подграф графа G (тј. F не садржи циклусе). Конструисати ефикасан алгоритам за одређивање повезујућег стабла G које садржи све гране F, и (под тим условима) има минималну цену.

6.44. Нека је e грана максималне тежине у неком циклусу графа $G = (V, E)$. Доказати да постоји MCST графа $G' = (V, E \setminus \{e\})$ које је истовремено и MCST графа G.

6.45. Нека је $G = (V, E)$ повезан неусмерен тежински граф. Може се претпоставити да су тежине грана позитивне и различите. За произвољну грану $e \in E$ нека $T(e)$ означава повезујуће стабло G са минималном ценом међу свим повезујућим стаблима G која садрже e. Конструисати алгоритам сложености $O(|V|^2)$, који одређује $T(e)$ за све гране $e \in E$.

6.46. Нека је $G = (V, E)$ неусмерен тежински граф и нека је T неко минимално повезујуће стабло (MCST) графа G. Претпоставимо да су тежине свих грана у G повећане за константу c. Да ли је T и даље MCST?

6.47. Нека је $G = (V, E)$ повезан неусмерени граф са n чворова нумерисаних од 1 до n. Конструисати ефикасан алгоритам за одређивање најмањег k таквог да ако се редом уклоне чворови $1, 2, \ldots, k$, онда се добија граф чије све компоненте садрже највише $n/2$ чворова. Уклањање чвора подразумева и уклањање свих њему суседних грана.

6.48. Нека је $G = (V, E)$ неусмерен граф. Скуп грана $F \subseteq E$ зове се **скуп повратних грана** ако сваки циклус у G садржи бар једну грану из F. Конструисати алгоритам за налажење минималног скупа повратних грана.

6.49. Доказати да алгоритам за одређивање свих најкраћих путева са слике 25 ради коректно за тежинске графове са неким негативним тежинама, под условом да тежина ни једног циклуса није негативна.

6.50. Нека је $G = (V, E)$ усмерен тежински граф у коме неке тежине могу бити негативне, али нема циклуса негативне тежине (циклуса у коме је збир тежина грана негативан). Нека је T повезујуће стабло G са кореном v. Конструисати алгоритам линеарне сложености који утврђује да ли T садржи само најкраће путеве од v до свих осталих чворова G (излаз је само одговор "да" или "не").

6.51. Нека је $G = (V, E)$ усмерен тежински граф у коме неке гране имају негативну тежину. Конструисати ефикасан алгоритам за утврђивање да ли граф садржи циклус негативне тежине (довољно је одговорити са "да" или "не").

6.52. **Хамилтонов пут** је прости пут који садржи све чворове графа. Конструисати алгоритам линеарне сложености који утврђује да ли задати ациклички усмерени граф $G = (V, E)$ садржи неки Хамилтонов пут.

6.53. Алгоритам $Tranzit_zatvorenje2$, (слика 27) има три уметнуте петље. Спољашња петља бира колону, друга врсту, а трећа обрађују изабрану врсту. Претпоставимо да је замењен редослед прве и друге петље. Показати контрапримером да овај алгоритам не налази увек транзитивно затворење.

6.54. **База чворова** усмереног графа $G = (V, E)$ је подскуп $B \subseteq V$ најмање величине са особином да за сваки чвор $v \in V$ постоји чвор $b \in B$, такав да постоји пут дужине веће или једнаке од 0 од b до v. Доказати следећа два тврђења.

(а) Чвор који не припада ни једном циклусу и има улазни степен већи од нуле не може бити ни у једној бази чворова.

(б) Ациклички усмерени граф има јединствену базу чворова, коју је лако одредити.

6.55. Конструисати алгоритам линеарне сложености за одређивање оптималног упаривања у стаблу.

6.56. Дат је бипартитни граф G, чији су чворови a, b, c, d, e, f, g и 1, 2, 3, 4, 5, 6, 7, односно гране $\{(a, 1)$, $(a, 2)$, $(a, 5)$, $(b, 1)$, $(b, 2)$, $(b, 6)$, $(c, 1)$, $(c, 2)$, $(c, 3)$, $(c, 7)$, $(d, 1)$, $(d, 2)$, $(e, 1)$, $(e, 4)$, $(f, 5)$, $(g, 6)\}$, видети слику 39. Одредити оптимално упаривање у G.

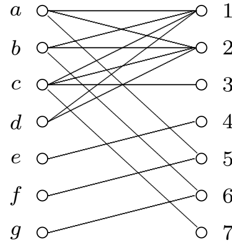


Рис. 39. Уз задатак 6.56.

6.57. Дат је усмерени граф (мрежа) $G = (V, E)$ са два издвојена чвора s и t (видети слику 40). Гранама графа придружени су бројеви, њихови капацитети. Одредити оптимални (највећи могући) ток кроз мрежу.

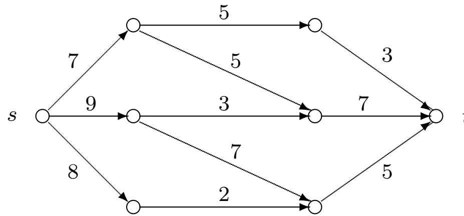


Рис. 40. Уз задатак 6.57.

6.58. **Звезда** је стабло са бар два чвора у коме је један чвор суседан свим осталим чворовима. Нека је $G = (V, E)$ неусмерен повезан граф. Задатак је конструисати алгоритам за налажење (било које) колекције дисјунктних звезда (без заједничких чворова), такве да сваки чвор припада некој звезди. Другим речима, звезде треба да покривају све чворове, али не обавезно и све гране графа.

(а) Пронаћи грешку у следећем алгоритму заснованом на индукцији. Индуктивна хипотеза је да умемо да решимо проблем за графове са мање од n чворова. За дати граф $G = (V, E)$ са n чворова, бирамо његов произвољан чвор v и уклањамо га из G заједно са свим суседима. У преосталом графу се затим свака компонента повезаности посебно разматра и на њу се примењује алгоритам који по индуктивној хипотези постоји.

(б) Конструисати ефикасан (и коректан) алгоритам за решавање овог проблема.

6.59. Задат је тежински бипартитни граф $G = (V, E)$ са n чворова и m грана **Критична тежина** упаривања M (у G) је тежина најтеже гране у M. Конструисати алгоритам сложености $O(\sqrt{n}m\log n)$ за налажење оптималног упаривања са минималном критичном тежином.

6.60. Резултати турнира између n играча, у коме се сваки учесник одмерава са свима осталим и нема нерешених резултата, дати су матрицом. У општем случају учесници се не могу рангирати, јер се може десити нпр. да је A победио B, B победио C, и C победио A (тј. резултати не морају бити транзитивни). Задатак је одредити такав редослед играча $P_1, P_2, \ldots, P_n$, да је P_i победио P_{i+1} за $i = 1, 2, \ldots, n-1$. Алгоритам треба да буде сложености $O(n \log n)$.

6.61. * **Бојење графа** је придруживање боја гранама графа, такво да гране суседне истом чвору имају различите боје. Нека је k степен двојке. Конструисати алгоритам сложености $O(|E| \log k)$ за бојење неусмереног бипартитног графа чији сви чворови имају степен k.

6.62. Покривач грана неусмереног графа $G = (V, E)$ је скуп чворова U такав да је свака грана из E суседна бар једном чвору из U. Конструисати ефикасан алгоритам за налажење најмањег покривача грана датог стабла.

6.63. Нека је $G = (V, E)$ стабло, чијем је сваком чвору додељена тежина једнака његовом степену. Конструисати алгоритам за налажење **покривача грана минималне тежине** графа G.

6.64. Нека је $G = (V, E)$ повезан неусмерен граф, и нека је k природан број. Конструисати алгоритам за утврђивање да ли у G постоји покривач грана (видети задатак 6.62) са највише k чворова, који је истовремено и независан скуп (тј. не постоји грана између било која два од тих k чворова).

6.65. Конструисати алгоритам који за дати граф $G = (V, E)$ утврђује да ли садржи подскуп U чворова који је истовремено и минимални покривач грана и максимални независни скуп. Алгоритам треба да пронађе такав скуп U ако постоји.

6.66. Интервални граф је неусмерени граф чији чворови одговарају интервалима на реалној оси, а два чвора су повезана ако се одговарајући интервали секу. Нека је G интервални граф неког скупа интервала. Конструисати ефикасан алгоритам за налажење максималног независног скупа у G (максималног скупа чворова таквог да никоја два његова чвора нису повезана граном).

6.67. (а) Конструисати алгоритам сложености $O(|V|^3)$ за утврђивање да ли задати неусмерени граф $G = (V, E)$ садржи квадрат (циклус дужине четири) као подграф.

(б) Усавршити алгоритам, тако да му сложеност буде $O(|V||E|)$. Може се користити матрица повезаности или листа повезаности графа.

6.68. Показати да не постоји алгоритам сложености $O(|V||E|)$ за налажење *свих* квадрата (видети задатак 6.67) који су подграфови неусмереног графа $G = (V, E)$.

Глава 7

Геометријски алгоритми

7.1. Увод

Геометријски алгоритми играју важну улогу у многим областима, на пример у рачунарској графици, пројектовању помоћу рачунара, пројектовању VLSI (интегрисаних кола високе резолуције), роботици и базама података. У рачунарски генерисаној слици може бити на хиљаде или чак милионе тачака, линија, квадрата или кругова; пројектовање компјутерског чипа може да захтева рад са милионима елемената. Сви ови проблеми садрже обраду геометријских објеката. Пошто величина улаза за ове проблеме може бити врло велика, веома је значајно знати ефикасне алгоритме за њихово решавање.

Постоје две у извесној мери раздвојене области у којима се појављују геометријски алгоритми, али се, на жалост, обе зову *рачунарска графика*. Једна од њих бави се **континуалним**, а друга **дискретним** аспектима геометријских објеката. Граница није прецизна, па постоје многи слични проблеми и технике. Овде ћемо се више бавити дискретном рачунарском графиком. У овом поглављу размотрићемо неколико основних геометријских алгоритама — оних који се могу користити као елементи за изградњу сложенијих алгоритама, односно алгоритама који илуструју занимљиве технике.

Објекти са којима се ради су тачке, праве, дужи и многоуглови. Алгоритми обрађују ове објекте, односно израчунавају неке њихове карактеристике. Најпре дајемо основне дефиниције и наводимо структуре података погодне за представљање појединих објеката. **Тачка** p у равни представља се као пар координата (x, y) (претпоставља се да је изабран фиксирани координатни систем). **Права** је представљена паром тачака p и q (произвољне две различите тачке на правој), и означава се са $-p-q-$. **Дуж** се пак представља паром тачака p и q — крајева дужи, и означава са $p-q$. **Пут** P је низ тачака p_1, $p_2, \ldots, p_k$ и дужи $p_1 - p_2$, $p_2 - p_3, \ldots$, $p_{k-1} - p_k$ које их повезују. Дужи које чине пут су његове **ивице** (странице). **Затворени пут** је пут чија се последња тачка поклапа са првом. Затворени пут зове се и **многоугао**. 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). **Прости многоугао** је онај код кога одговарајући пут нема пресека са самим собом; другим речима, једине ивице које имају заједничке тачке су суседне ивице са њиховим заједничким

теменом. Прости многоугао ограничава једну област у равни, **унутрашњост** многоугла. **Конвексни многоугао** је многоугао чија унутрашњост са сваке две тачке које садржи, садржи и све тачке те дужи. **Конвексни пут** је пут од тачака $p_1, p_2, \ldots, p_k$ такав је многоугао $p_1 p_2 \ldots p_k$ конвексан.

Претпоставља се да је читалац упознат са основама аналитичке геометрије. У алгоритмима које ћемо разматрати наилази се, на пример, на израчунавање пресечне тачке двеју дужи, утврђивање да ли тачка лежи са задате стране праве, израчунавање растојања између две тачке. Све ове операције могу се извести за време ограничено константом, помоћу основних аритметичких операција. При томе, на пример, претпостављамо да се квадратни корен може израчунати за константно време; овај проблем биће споменут у одељку 7.3.

Честа неугодна карактеристика геометријских проблема је постојање многих "специјалних случајева". На пример, две праве у равни обично 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. Компликованији објекти проузрокују појаву много већег броја специјалних случајева, о којима треба водити рачуна. Обично се већина тих специјалних случајева непосредно решава, али потреба да се они узму у обзир чини понекад конструкцију геометријских алгоритама врло исцрпљујућом. Ми ћемо намерно игнорисати детаље који нису од суштинског значаја за разумевање основних идеја алгоритама.

7.2. Утврђивање да ли задата тачка припада многоуглу

Разматрање започињемо једним једноставним проблемом.

Проблем. Задат је прост многоугао P и тачка q. Установити да ли је тачка у или ван многоугла P.

Проблем изгледа једноставно на први поглед, али ако се разматрају сложени неконвексни многоуглови, као онај на слици 1, проблем сигурно није једноставан. Први интуитивни приступ је покушати некако "изаћи напоље", полазећи од задате тачке. Посматрајмо произвољну полуправу са теменом q. Види се да је довољно пребројати пресеке са страницама многоугла, све до достизања спољашње области. У примеру на слици 1, идући на североисток од дате тачке (пратећи испрекидану линију), наилазимо на шест пресека са многоуглом до достизања спољашње области (истина, већ после четири пресека стиже се ван многоугла, али то рачунар "не види"; зато се броји укупан број пресека полуправе са страницама многоугла). Пошто нас последњи пресек пре изласка изводи из многоугла, а претпоследњи нас враћа у многоугао, итд, тачка је ван многоугла. Уопште (специјалне случајеве на тренутак занемарујемо) тачка је у многоуглу ако и само ако је непаран. Имамо дакле скицу алгоритма, видети слику 2.

Као што је речено у претходном одељку, обично постоје неки специјални случајеви које треба посебно размотрити. Нека је s тачка ван многоугла, и нека је L дуж која спаја q и s. Циљ је утврдити да ли q припада унутрашњости

Рис. 1. Утврђивање припадности тачке унутрашњости простог многоугла.

Алгоритам Tačka_u_mnogouglu_1(P, q); {први покушај}
Улаз: P (прост многоугао са теменима $p_1, p_2, \ldots, p_n$ и ивицама $e_1, e_2, \ldots, e_n$), и q (тачка).
Излаз: *Pripada* (Булова променљива, *true* акко q припада P).
begin
    Изабрати произвољну тачку s ван многоугла;
    Нека је L дуж $q - s$;
    broj := 0;
    for све гране e_i многугла **do**
        if e_i сече L **then**
            {претпостављамо да пресек није ни теме ни ивица, видети текст}
            broj := *broj* + 1;
    if *broj* је непаран **then** *pripada* := *true*
    else *pripada* := *false*
end

Рис. 2. Провера припадности тачке унутрашњости задатог простог многоугла.

P на основу броја пресека L са ивицама P. Међутим, дуж L може се делом преклапати са неким ивицама P. Ово преклапање очигледно не треба бројати у пресеке. Други специјални случај је пресек L са теменом многоугла. На слици 3(а) види се пример случаја кад пресек L са теменом не треба бројати, а на слици 3(б) пример кад тај пресек треба бројати. Прецизирање алгоритма у овом смислу препушта се читаоцу (видети задатак 7.1).

Развијајући овај алгоритам, имплицитно смо претпостављали да радимо са сликом. Проблем је нешто другачији кад је улаз дат низом координата, што је уобичајено. На пример, кад посао радимо ручно и видимо многоугао, лако је наћи добар пут (онај са мало пресека) до неке тачке ван многоугла. У случају многоугла датог низом координата, то није лако. Највећи део времена троши се на израчунавање пресека. Тај део посла може се битно упростити ако је дуж $q - s$ паралелна једној од оса — на пример y-оси. Број пресека са

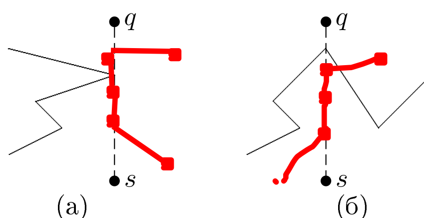


Рис. 3. Специјални случајеви кад права сече ивицу кроз теме.

овом специјалном дужи може бити много већи него са оптималном дужи (коју није лако одредити — видети задатак 7.3), али је налажење пресека много једноставније. Модификовани алгоритам приказан је на слици 4.

Алгоритам Tačka_u_mnogouglu_2(P, q); {други покушај}
Улаз: P (прости многоугао са теменима $p_1, p_2, \ldots, p_n$ и страницама $e_1, e_2, \ldots, e_n$),
и q (тачка са координатама (x_0, y_0)).
Излаз: *Pripada* (булова променљива, *true* акко q припада P).
begin
  Broj := 0;
  for све гране e_i многоугла **do**
    if права $x = x_0$ сече e_i **then**
      {Претпостављамо да пресек није ни теме ни страница многоугла}
      Нека је y_i y-координата пресека праве $x = x_0$ са e_i;
      if $y_i < y_0$ **then** {пресек је испод q}
        Broj := *Broj* + 1;
  if *Broj* је непаран **then** *Pripada* := *true*
  else *Pripada* := *false*
end

Рис. 4. Усавршена варијанта алгоритма за проверу да ли тачка припада задатом простом многоуглу.

Сложеност. За израчунавање пресека две дужи потребно је константно време. Нека је n број ивица многоугла. Кроз основну петљу алгоритма пролази се n пута. У сваком проласку налази се пресек две дужи и извршавају се још неке операције за константно време. Дакле, укупно време извршавања алгоритма је $O(n)$.

Коментар. У много случајева једноставан поступак инспирисан обичним алгоритмом (оним којим се ручно решава проблем) није ефикасан за велике улазе. Понекад је пак такав приступ не само једноставан, него и ефикасан. Приступ решавању проблема поступком који се визуелно намеће је обично добар избор. Тиме се може доћи до више корисних запажања о проблему. У

овом случају, посматрајући слику, приметили смо да се проблем може решити пратећи неки пут од тачке до спољашњости многоугла.

7.3. Конструкција простог многоугла

Скуп тачака у равни дефинише много различитих многоуглова, зависно од изабраног редоследа тачака. Размотрићемо сада проналажење простог многоугла са задатим скупом темена.

Проблем. Дато је n 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. Приказаћемо најпре геометријски приступ овом проблему. Нека је C велики круг, чија унутрашњост садржи све тачке. За налажење таквог круга довољно је $O(n)$ операција — израчунавања највећег међу растојањима произвољне тачке равни (центра круга) до свих n тачака. Површина C може се "пребрисати" (прегледати) ротирајућом полуправом којој је почетак центар C, видети слику 5. Претпоставимо засад да ротирајућа полуправа у сваком тренутку садржи највише једну тачку. Очекујемо да ћемо спајањем тачака оним редом којим полуправа наилази на њих, добити прост многоугао. Покушајмо да то докажемо. Означимо тачке, уређене у складу са редоследом наиласка полуправе на њих, са $p_1, p_2, \ldots, p_n$ (прва тачка бира се произвољно). За свако i, $1 \leq i \leq n$, страница $p_i - p_{i-1}$ (односно $p_1 - p_n$ за $i = 1$) садржана је у новом (дисјунктном) исечку круга, па се не сече ни са једном другом страном. Ако би ово тврђење било тачно, добијени многоугао би морао да буде прост. Међутим, 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 $p_i - p_{i+1}$ садржи више од пола круга и није конвексна фигура, а дуж $p_i - p_{i+1}$ пролази кроз друге исечке круга, па може да сече друге стране многоугла. Да бисмо се уверили да је то могуће, довољно је да замислимо круг са центром ван круга са слике 5. Ово је добар пример специјалних случајева на које се наилази при решавању геометријских проблема. Потребно је да будемо пажљиви, да бисмо били сигурни да су сви случајеви размотрени. На ову појаву наилази се код свих врста алгоритама, али је она код геометријских алгоритама драстичније изражена.

Да би се решио уочени проблем, могу се, на пример, фиксирати произвољне три тачке из скупа, а за центар круга изабрати неку тачку унутар њима одређеног троугла (на пример тежиште, које се лако налази). Овакав избор гарантује да ни један од добијених сектора круга неће имати угао већи од π. Могуће је изабрати и друго решење, да се за центар круга узме једна од тачака из скупа — тачка z са највећом x-координатом (и са најмањом y-координатом, ако има више тачака са највећом x-координатом). Затим користимо исти основни алгоритам. Сортирамо тачке према положају у кругу са центром z. Прецизније, сортирају се *углови* између x-осе и полуправих од z ка осталим тачкама. Ако

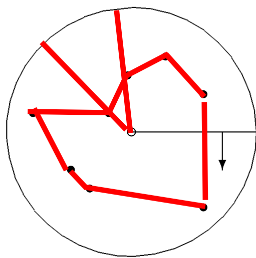


Рис. 5. Пролазак тачака у кругу ротирајућом полуправом.

две или више тачака имају исти угао, оне се даље сортирају према растојању од тачке z. На крају, z се повезује са тачком са најмањим и највећим углом, а остале тачке повезују се у складу са добијеним уређењем, по две узастопне. Пошто све тачке леже лево од z, до дегенерисаног случаја о коме је било речи не може доћи. Прост многоугао добијен овим поступком за тачке са слике 5 приказан је на слици 6.

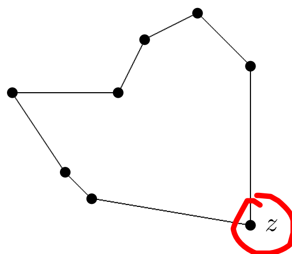


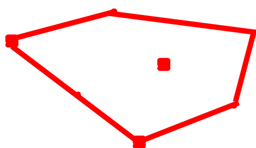
Рис. 6. Конструкција простог многоугла.

Описани метод може се усавршити на два начина. Прво, углови се не морају експлицитно израчунавати. Углови се користе само за налажење редоследа којим треба повезати тачке. Исти редослед добија се уређењем *нагиба* (односно односа прираштаја y- и x-координата) одговарајућих полуправих; то чини непотребним израчунавање аркустангенса. Из истог разлога непотребно је израчунавање растојања кад две тачке имају исти нагиб — довољно је израчунати квадрате растојања. Дакле, нема потребе за израчунавањем квадратних коренова. Алгоритам је приказан на слици 7.

Сложеност. Основна компонента временске сложености овог алгоритма потиче од сортирања. Сложеност алгоритма је дакле $O(n \log n)$.

7.4. Конвексни омотач

Конвексни омотач скупа тачака дефинише се као најмањи конвексни многоугао који садржи све тачке скупа. Конвексни омотач се представља на исти начин као обичан многоугао, теменима наведеним у цикличком редоследу.



Алгоритам Prost_mnogougao$(p_1, p_2, \ldots, p_n)$;
Улаз: $p_1, p_2, \ldots, p_n$ (тачке у равни).
Излаз: P (прост многоугао са теменима $p_1, p_2, \ldots, p_n$ у неком редоследу).
begin
  променити ознаке тако да p_1 буде екстремна тачка;
  {тачка са највећом x-координатом, а ако таквих има више,}
  {она од њих која има најмањуy-координату}
  for $i := 2$ **to** n **do**
    израчунати угао α_i између праве $-p_1 - p_i-$ и x-осе;
  сортирати тачке према угловима $\alpha_2, \ldots, \alpha_n$;
  {у групи са истим углом сортирати их према растојању од p_1}
  P је многоугао дефинисан сортираном листом тачака
end

Рис. 7. Алгоритам за конструкцију простог многоугла са задатим скупом темена.

Проблем. Конструисати конвексни омотач задатих n тачака у равни.

Обрада конвексних многоуглова једноставнија је од обраде произвољних многоуглова. На пример, постоји алгоритам сложености $O(\log n)$ за утврђивање припадности тачке конвексном n-тоуглу (видети задатак 7.11). Темена конвексног многоугла су неке од тачака из задатог скупа. Кажемо да тачка *припада* омотачу ако је теме омотача. Ковексни омотач може се састојати од најмање три, а највише n тачака. Конвексни омотачи имају широку примену, па су због тога развијени многобројни алгоритми за њихову конструкцију.

7.4.1. Директни приступ. Као и обично, покушаћемо најпре са директним индуктивним приступом. Конвексни омотач за три тачке лако је наћи. Претпоставимо да умемо да конструишемо конвексни омотач скупа од $< n$ тачака, и покушајмо да конструишемо конвексни омотач скупа од n тачака. Како n-та тачка може да промени конвексни омотач за првих $n-1$ тачака? Постоје два могућа случаја: или је нова тачка у претходном конвексном омотачу (тада он остаје непромењен), или је она ван њега, па се омотач "шири" да обухвати и нову тачку, видети слику 8. Потребно је дакле решити два потпроблема: утврђивање да ли је нова тачка унутар омотача и проширивање омотача новом тачком. Они нису једноставни. Ствар се може упростити погодним избором n-те тачке. Звучи изазовно покушати са избором тачке унутар омотача; то, међутим, није увек могуће јер у неким случајевима све тачке припадају омотачу. Друга могућност, која се показала успешном у претходном проблему, је избор екстремне тачке за n-ту тачку.

Изаберимо поново тачку са највећом x-координатом (и минималном y-координатом ако има више тачака са највећом x-координатом). Нека је то тачка q. Јасно је да тачка q мора бити теме конвексног омотача. Питање је како

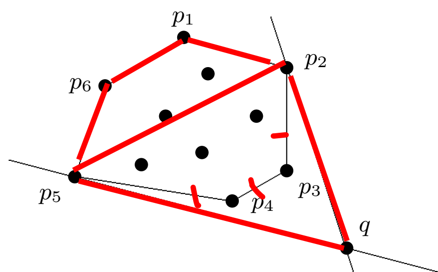


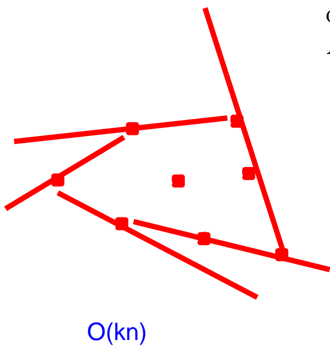
Рис. 8. Проширивање конвексног омотача новом тачком.

променити конвексни омотач осталих $n-1$ тачака тако да обухвати и q. Потребно је најпре пронаћи темена старог омотача која су у унутрашњости новог омотача (p_3 и p_4 на слици 8) и уклонити их; затим се умеће ново теме q између два постојећа (p_2 и p_5 на слици 8). **Права ослонца** конвексног многоугла је права која многоугао сече у тачно једној тачки. Многоугао увек цео лежи са једне стране своје праве ослонца. Посматрајмо сада праве ослонца $-q-p_2-$ и $-q-p_5-$ на слици 8. Обично само два темена многоугла повезивањем са q одређују праве ослонца (игнорисаћемо специјални случај кад два или више темена многоугла леже на истој правој са q). Многоугао лежи између две праве ослонца, што указује како треба извести модификацију омотача. Праве ослонца заклапају минимални и максимални угао са x-осом међу свим правим кроз q и неко теме многоугла. Да бисмо одредили та два темена, треба да размотримо праве из q ка свим осталим теменима, да израчунамо углове које оне заклапају са x-осом, и међу тим угловима изаберемо минималан и максималан (видети задатак 7.4). После проналажења два екстремна темена лако је конструисати модификовани омотач. Изоставићемо детаље алгоритма, јер, као што ћемо видети, постоји и ефикаснији алгоритам.

Сложеност. За сваку тачку треба израчунати углове правих ка свим претходним теменима и x-осе, пронаћи минимални и максимални угао, додати нови чвор и избацити неке чворове. Сложеност додавања k-те тачке је дакле $O(k)$, а видели смо већ да је решење диференцне једначине $T(n) = T(n-1)+cn$ облика $O(n^2)$. Према томе, сложеност овог алгоритма је $O(n^2)$. Алгоритам на почетку такође захтева и сортирање, али је време сортирања асимптотски мање од времена потребног за остале операције.

7.4.2. Увијање поклона. Како се може побољшати описани алгоритам? Кад проширујемо многоугао теме по теме, доста времена трошимо на формирање конвексних многоуглова од тачака које могу бити унутрашње за коначни конвексни омотач. Може ли се то избећи? Уместо да правимо 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 (која увек припада омотачу), пронаћи њој суседна темена омотача налазећи праве ослонца, и наставити на исти начин

од тих суседа. Овај алгоритам из разумљивих разлога зове се **увијање поклона**. Полази се од једног темена "поклона", и онда се он увија у конвексни омотач проналазећи теме по теме омотача. Алгоритам је приказан на слици 9. Алгоритам се може преправити тако да ради и у просторима веће димензије.



Алгоритам Uvijanje_poklona$(p_1, p_2, \ldots, p_n)$;
Улаз: $p_1, p_2, \ldots, p_n$ (скуп тачака у равни).
Излаз: P (конвексни омотач тачака $p_1, p_2, \ldots, p_n$).
begin
Нека је P празан скуп;
Нека је p тачка са највећом x-координатом
(и са најмањом y-координатом, ако има више
тачка са највећом x-координатом);
Укључи p у скуп P;
Нека је L права кроз p паралелна са x-осом;
while омотач P није завршен **do**
Нека је q тачка за коју је најмањи угао између L и $-p-q-$;
Укључи q у скуп P;
$L :=$ права $-p-q-$;
$p := q$
end

Рис. 9. Алгоритам "увијање поклона" за конструкцију конвексног омотача задатог скупа тачака.

Алгоритам увијање поклона је директна последица примене следеће индуктивне хипотезе (по k):

Индуктивна хипотеза. За задати скуп од n тачака у равни, умемо да пронађемо конвексни пут дужине $k < n$ који је део конвексног омотача скупа.

Код ове хипотезе нагласак је на проширивању *пута*, а не омотача. Уместо да проналазимо конвексне омотаче мањих скупова, ми проналазимо део коначног конвексног омотача.

Сложеност. Да бисмо додали k-то теме омотачу, треба да пронађемо праву са најмањим и највећим углом у скупу од $n-k$ правих. Због тога је временска сложеност алгоритма увијање поклона $O(n^2)$, што асимптотски није боље од алгоритма заснованог на проширивању омотача.

7.4.3. Грахамов алгоритам. Сада ћемо размотрити алгоритам за налажење конвексног омотача сложености $O(n \log n)$. Започиње се сортирањем тачака према угловима, слично као при конструкцији простог многоугла у одељку 7.3. Нека је p_1 тачка са највећом x-координатом (и са најмањом y-координатом, ако има више тачака са највећом x-координатом). За сваку тачку

p_i израчунавамо угао између праве $-p_1 - p_i-$ и x-осе, и сортирамо тачке према величини ових углова, видети слику 10. Тачке пролазимо редоследом којим се појављују у (простом) многоуглу, и покушавамо да идентификујемо темена конвексног омотача. Као и код алгоритма увијање поклона памтимо пут састављен од дела пређених тачака. Прецизније, то је конвексни пут чији конвексни многоугао садржи све до сада прегледане тачке (одговарајући конвексни многоугао добија се повезивањем прве и последње тачке пута). Због тога, у тренутку кад су све тачке прегледане, конвексни омотач скупа тачака је конструисан. Основна разлика између овог алгоритма и увијања поклона је у чињеници да текући конвексни пут не мора да буде део коначног конвексног омотача. То је само део конвексног омотача до сада прегледаних тачака. Пут може да садржи тачке које не припадају коначном конвексном омотачу; те тачке биће елиминисане касније. На пример, пут од p_1 до q_m на слици 10 је конвексан, али q_m и q_{m-1} очигледно не припадају конвексном омотачу. Ово разматрање сугерише алгоритам заснован на следећој индуктивној хипотези.

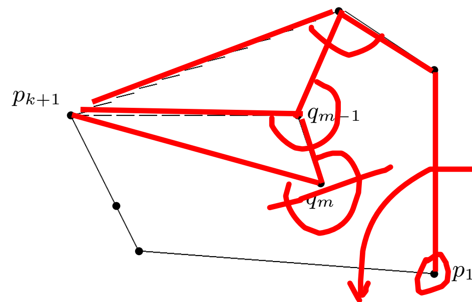


Рис. 10. Грахамов алгоритам за налажење конвексног омотача скупа тачака.

Индуктивна хипотеза. Ако је дато n тачака у равни, уређених према алгоритму *Prost_mnogougao* (одељак 7.3), онда умемо да конструишемо конвексни пут преко неких од првих k тачака, такав да одговарајући конвексни многоугао обухвата првих k тачака.

Случај $k = 1$ је тривијалан. Означимо конвексни пут добијен (индуктивно) за првих k тачака са $P = q_1, q_2, \ldots, q_m$. Сада треба да проширимо индуктивну хипотезу на $k+1$ тачака. Посматрајмо угао између правих $-q_{m-1} - q_m-$ и $-q_m - p_{k+1}-$ (видети слику 10). Ако је тај угао мањи од π (при чему се угао мери из *унутрашњости* многоугла), онда се p_{k+1} може додати постојећем путу (нови пут је због тога такође конвексан), чиме је корак индукције завршен. У противном, тврдимо да q_m лежи у многоуглу добијеном заменом q_m у P тачком p_{k+1}, и повезивањем p_{k+1} са p_1. Ово је тачно јер су тачке уређене преме одговарајућим угловима. Права $-p_1 - p_{k+1}-$ лежи "лево" од првих k тачака. Због тога q_m јесте унутар горе дефинисаног многоугла, може се избацити из P, а p_{k+1} се може додати. Да ли је тиме све урађено? Не сасвим. Иако се q_m може

елиминисати, модификовани пут не мора увек да буде конвексан. Заиста, слика 10 јасно показује да постоје случајеви кад треба елиминисати још тачака. На пример, тачка q_{m-1} може да буде унутар многоугла дефинисаног модификованим путем. Морамо да (уназад) наставимо са проверама последње две гране пута, све док угао између њих не постане мањи од π. Пут је тада конвексан, а хипотеза је проширена на $k+1$ тачку. Детаљи алгоритма приказани су на слици 11.

```
Алгоритам Grahamov_algoritam(p_1, p_2, ..., p_n);
Улаз: p_1, p_2, ..., p_n (скуп тачака у равни).
Излаз: q_1, q_2, ..., q_m (конвексни омотач тачака p_1, p_2, ..., p_n).
begin
    Нека је p_1 тачка са највећом x-координатом (а најмањом y-координатом,
        ако има више тачака са највећом x-координатом);
    Помоћу алгоритма Prost_mnogougao уредити тачке у односу на p_1;
        нека је то редослед p_1, p_2, ..., p_n;
    q_1 := p_1;
    q_2 := p_2;
    q_3 := p_3; {пут P се на почетку састоји од p_1, p_2 и p_3}
    m := 3;
    for k := 4 to n do
        while угао између −q_{m−1} − q_m− и −q_m − p_k−) ≥ π do
            m := m − 1;
        m := m + 1;
        q_m := p_k {на слици 10 то је p_{k+1}}
end
```

Рис. 11. Грахамов алгоритам за конструкцију конвексног омотача задатог скупа тачака.

Сложеност. Главни део сложености 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+1}. 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, јер тачка може бити елиминисана само једном! Према томе, троши се највише константно време за елиминацију сваке тачке и константно време за њено додавање. Укупно је за ову фазу потребно $O(n)$ корака. Због сортирања је време извршавања комплетног алгоритма $O(n \log n)$.

7.5. Најближи пар тачака

Претпоставимо да су задате локације n објеката и да је задатак да проверити постоје ли међу њима два, који су преблизу један другом. Ови објекти могу бити, на пример, делови микропроцесорског чипа, звезде у галаксији или системи за наводњавање. Овде ћемо размотрити само једну варијанту овог проблема, као представника шире класе.

Проблем. У задатом скупу од n тачака у равни пронаћи две које су на најмањем међусобном растојању.

Слични овом су проблеми налажења најближе тачке (или k најближих тачака) за сваку тачку задатог скупа, или налажење најближе тачке новододатој тачки.

7.5.1. Директни приступ. Могу се израчунати растојања између сваке две тачке, и затим пронаћи најмање међу растојањима. То обухвата $n(n-1)/2$ израчунавања растојања и $n(n-1)/2-1$ упоређивања. Директно индуктивно решење могло би се заснивати на уклањању једне тачке, решавању проблема за $n-1$ тачака, и додавању нове тачке. Међутим, једина корисна информација која се добија решавањем проблема за $n-1$ тачака је минимално растојање, па се морају проверити растојања нове тачке до свих претходних $n-1$ тачака. Због тога укупан број $T(n)$ израчунавања растојања за n тачака задовољава диференцну једначину $T(n) = T(n-1) + n - 1$, $T(2) = 1$, чије је решење $T(n) = O(n^2)$. Два описана решења суштински су еквивалентна. Циљ је да пронаћи ефикаснији алгоритам за велике n.

7.5.2. Алгоритам заснован на декомпозицији. Уместо да разматрамо тачке једну по једну, можемо да скуп тачака поделимо на два једнака дела. Индуктивна хипотеза остаје непромењена, изузев што проблем сводимо не на један проблем са $n-1$ тачком, него на два проблема са $n/2$ тачака. Због једноставности претпоставимо да је n степен двојке, тако да је скуп увек могуће поделити на два једнака дела. Постоји више начина да се скуп подели на два једнака дела. Бирамо начин који највише одговара нашим циљевима. Волели бисмо да добијемо што више корисних информација из решења мањих проблема, односно да што већи део тих информација важи и за комплетан проблем. Изгледа разумно да се скуп подели на два дела поделом равни на два дисјунктна дела, тако да сваки од њих садржи половину тачака. Пошто се пронађе најмања растојања у сваком делу, треба размотрити само растојања између тачака блиских граници скупова. Најједноставнији начин поделе је сортирати тачке према (на пример) x-координатама и поделити раван правом паралелном са y-осом, која дели скуп на два једнака дела, видети слику 12 (ако више тачака лежи на правој поделе, тачке се могу на произвољан начин разделити између скупова). Начин поделе изабран је тако да се максимално поједноставиобједињавање решења мањих проблема. Сортирање треба извршити само једном.

T(n)=2T(n/2)+n^2/4

+O(n log n) O(n(log n)^2)

+O(n) O(n log n)

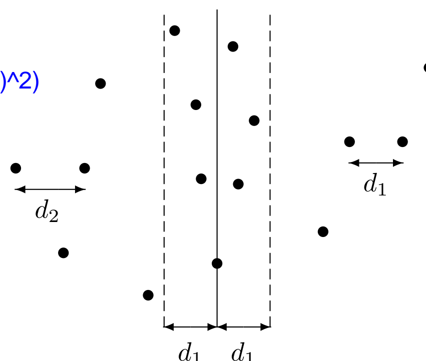


Рис. 12. Проблем налажења две најближе тачке.

Због једноставности ограничићемо се на налажење *вредности* најмањег растојања између тачака (а не и пара тачака за које се оно достиже). Проналажење две најближе тачке изводи се минималном дорадом алгоритма. Нека је P скуп тачака и нека је n степен двојке. Најпре се скуп P дели на два истобројна подскупа P_1 и P_2 на описани начин. Најмање растојање у сваком подскупу проналази се на основу индуктивне хипотезе. Нека је d_1 минимално растојање у P_1, а d_2 минимално растојање у P_2. Без смањења општости може се претпоставити да је $d_1 \leq d_2$. Потребно је пронаћи најмање растојање у целом скупу, односно установити да ли у P_1 постоји тачка на растојању мањем од d_1 од неке тачке у P_2. Приметимо најпре да је довољно разматрати тачке у траци ширине $2d_1$, симетричној у односу на праву поделе (видети слику 12). Остале тачке не могу бити на растојању мањем од d_1 од неке тачке из другог подскупа. На основу овог запажања обично се из разматрања елиминише велики број тачака. Међутим, у најгорем случају све тачке могу бити у траци, па не можемо да себи приуштимо примену тривијалног алгоритма на њих.

Друго мање очигледно запажање је да за произвољну тачку p у траци постоји само мали број тачака у супротној траци чије растојање од p може бити мање од d_1. Искористићемо чињеницу да је $\sqrt{2}/2 > 0.7$. Нека је са y_p означена y-координата тачке p. Конструишимо правоугаоник висине $2.1d_1 < \frac{3}{2}\sqrt{2}d_1$ и ширине $1.4d_1 < \sqrt{2}d_1$ у супротној траци (видети слику 13), тако да належе на праву поделе и да му је y-координата пресека дијагонала једнака y_p. Поделимо га једном вертикалном правом и са две хоризонталне праве на шест једнаких квадрата странице $0.7d_1$ (којима је дијагонала мања од d_1!). У сваком од малих квадрата може се наћи највише једна тачка, јер је растојање било које две тачке у квадрату мање од d_1. Све тачке у другом подскупу, са растојањем од p мањим од d_1, очигледно се морају налазити у правоугаонику. Дакле, у другом подскупу се може налазити највише шест тачака на растојању од p мањем од d_1: тачка кандидат са y-координатом y_q мора да задовољи и услов $|y_p - y_q| < d_1$, па се мора налазити у једном од 6 квадрата. Према томе, ако све тачке у траци

сортирамо према y-координатама и прегледамо их тим редом, довољно је да за сваку тачку проверимо растојање осам суседа у супротној траци, четири испод и четири изнад (а не од свих $n-1$ тачака у најгорем случају). Скица алгоритма дата је на слици 14.

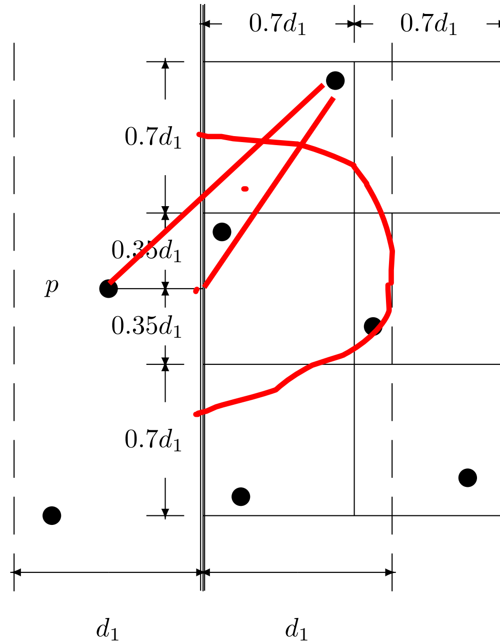


Рис. 13. Оцена максималног броја тачака из супротног подскупа, блиских фиксираној тачки.

Сложеност. Сортирање сложености $O(n \log n)$ извршава се само једном. Затим се решавају два потпроблема величине $n/2$. Елиминација тачака ван централне траке може се извести за $O(n)$ корака. За сортирање тачака у траци по y-координатама у најгорем случају потребно је $O(n \log n)$ корака. Коначно, потребно је $O(n)$ корака да се прегледају тачке у траци, и да се провере растојања сваке од њих од константног броја суседа у сортираном редоследу. Укупно, да би се решио проблем величине n, треба решити два потпроблема величине $n/2$ и извршити $O(n \log n)$ корака за комбиновање њихових решења (плус $O(n \log n)$ корака за једнократно сортирање тачака по x-координатама на почетку). Добијамо диференцну једначину $T(n) < 2T(n/2) + cn \log_2 n, \quad T(2) = 1$, при чему у $T(n)$ није укључено почетно сортирање по x-координатама. Индукцијом се показује да је њено решење $T(n) = O(n \log^2 n)$, односно прецизније $T(n) < cn \log_2^2 n$: из $T(n/2) < c(n/2) \log_2^2(n/2)$ и диференцне једначине следи да за $n > 2$ важи $T(n) < cn(\log_2 n - 1)^2 + cn \log_2 n = cn(\log_2^2 n - \log_2 n + 1) < cn \log_2^2 n$.

Алгоритам Najbliži_par1$(p_1, p_2, \ldots, p_n)$; {први покушај}
Улаз: $p_1, p_2, \ldots, p_n$ (скуп тачака у равни).
Излаз: d (растојање између две најближе тачке скупа).
begin
    Сортирати тачке према x-координатама;
    {ово сортирање извршава се само једном, на почетку}
    Поделити скуп на два истобројна подскупа;
    Рекурзивно израчунати минимално растојање у оба подскупа;
    Нека је d мање од два минимална растојања;
    Елиминисати тачке које су на растојању већем од d од праве поделе;
    Сортирати преостале тачке према y-координатама;
    За преостале тачке проверити њихова растојања од 4 претходне
        и 4 наредне тачке из супротне траке;
            {довољно је проверити и само 4 претходне тачке!}
    if неко од ових растојања је мање од d **then**
        поправити d
end

Рис. 14. Прва варијанта алгоритма за налажење најближег пара тачака.

Сложеност $O(n \log^2 n)$ је асимптотски је боља од квадратне, али видећемо да се може још мало побољшати.

7.5.3. Алгоритам сложености $O(n \log n)$. Основна идеја је појачати индуктивну хипотезу. У току обједињавања решења потпроблема изводи се $O(n \log n)$ корака због сортирања тачака по y-координатама. Може ли се сортирање извести успут у току налажења двеју најближих тачака? Другим речима, циљ је појачати индуктивну хипотезу за проблем две најближе тачке тако да обухвати и сортирање.

Индуктивна хипотеза. За задати скуп од $< n$ тачака у равни умемо да пронађемо најмање растојање и да скуп тачака сортирамо по y-координатама.

Већ смо видели како се може пронаћи минимално растојање ако се тачке у сваком кораку сортирају по y-координатама. Дакле, потребно је још само проширити индуктивну хипотезу тако да обухвати сортирање n тачака кад су два подскупа величине $n/2$ већ сортирана. Међутим, то је управо сортирање обједињавањем (одељак 5.3.3). Основна предност овог приступа је у томе што се при обједињавању решења не мора извести комплетно сортирање, него само обједињавање два већ сортирана подниза. Пошто се обједињавање сортираних поднизова изводи за $O(n)$ корака, диференцна једначина за сложеност (без почетног сортирања по x-координатама) постаје $T(n) < 2T(n/2) + cn$, $T(2) = 1$. Њено решење је $T(n) = O(n \log n)$, што је асимптотски једнако сложености

почетног сортирања тачака по x-координатама. Побољшани алгоритам приказан је на слици 15.

Алгоритам Najbliži_par2$(p_1, p_2, \ldots, p_n)$; {Побољшана верзија}
Улаз: $p_1, p_2, \ldots, p_n$ (скуп тачака у равни).
Излаз: d (растојање између две најближе тачке скупа).
begin
 Сортирати тачке према x-координатама;
 {ово сортирање извршава се само једном, на почетку}
 Поделити скуп на два истобројна подскупа;
 Рекурзивно извршити следеће:
  израчунати минимално растојање у оба подскупа;
  сортирати тачке у сваком делу према y-координатама;
 Обединити два сортирана низа у један;
 {Обједињавање се мора извршити пре елиминације:}
 {следећем нивоу рекурзије мора се испоручити сортиран комплетан скуп}
 Нека је d мање од два минимална растојања;
 Елиминисати тачке које су на растојању већем од d од праве поделе;
 За преостале тачке проверити њихова растојања од 4 претходне
  и 4 наредне тачке из супротне траке;
   {довољно је проверити и само 4 претходне тачке!}
 if неко од ових растојања је мање од d **then**
  поправити d
end

Рис. 15. Побољшана варијанта алгоритма за налажење најближег пара тачака.

7.6. Пресеци хоризонталних и вертикалних дужи

Често се наилази на проблеме налажења пресека. Понекад је потребно израчунавање пресека више објеката, а понекад само треба открити да ли је пресек непразан скуп. Проблеми детекције су обично лакши. У овом одељку приказаћемо један проблем налажења пресека, који илуструје важну технику за решавање геометријских проблема. Иста техника може се применити и на друге проблеме.

Проблем. За задати скуп од n хоризонталних и m вертикалних дужи пронаћи све њихове пресеке.

Овај проблем важан је, на пример, при пројектовању VLSI кола (интегрисаних кола са огромним бројем елемената). Коло може да садржи на

хиљаде "жичица", а пројектант треба да буде сигуран да не постоје неочекивани пресеци. На проблем се такође наилази при елиминацији скривених линија; тај проблем је обично компликованији, јер се не ради само о хоризонталним и вертикалним линијама. Пример проблема приказан је на слици 16.

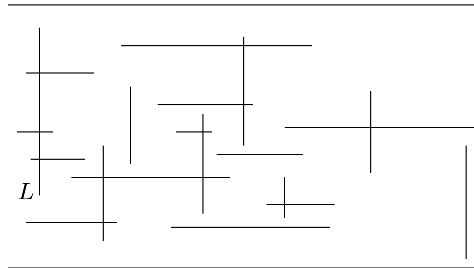


Рис. 16. Пресеци хоризонталних и вертикалних дужи.

Налажење свих пресека вертикалних дужи је једноставан проблем, који се оставља читаоцу за вежбање (исто се односи и на хоризонталне дужи). Претпоставимо због једноставности да не постоје пресеци између произвољне две хоризонталне, односно произвољне две вертикалне дужи. Ако покушамо да проблем решимо уклањањем једне по једне дужи (било хоризонталне, било вертикалне), онда ће бити неопходно да се пронађу пресеци уклоњене дужи са свим осталим дужима, па се добија алгоритам са $O(mn)$ налажења пресека дужи. 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. То се може постићи комбиновањем двеју идеја: избора специјалног редоследа индукције и појачавања индуктивне хипотезе.

Редослед примене индукције може се одредити покретном правом која прелази ("скенира") раван слева удесно; дужи се разматрају оним редом којим на њих наилази покретна права. Поред налажења пресечних тачака, треба чувати и неке податке о дужима које је покретна права већ захватила. Ти подаци биће корисни за ефикасније налажење наредних пресека. Ова техника зове се **техника покретне линије**.

Замислимо вертикалну праву која прелази раван слева удесно. Да бисмо остварили ефекат пребрисавања, сортирамо све крајеве дужи према њиховим x-координатама. Две крајње тачке вертикалне дужи имају исте x-координате, па је довољна једна од њих. За хоризонталне дужи морају се користити оба краја. После сортирања крајева, дужи се разматрају једна по једна утврђеним редоследом. Као и обично при индуктивном приступу, претпостављамо да смо пронашли пресечне тачке претходних дужи, и да смо обезбедили неке допунске информације, па сада покушавамо да обрадимо следећу дуж и да унесемо неопходне допуне информација. Према томе структура алгоритма је

у основи следећа. Разматрамо крајеве дужи један по један, слева удесно. Користимо информације прикупљене до сада (нисмо их још специфицирали) да обрадимо крај дужи, пронађемо пресеке у којима она учествује, и допуњујемо информације да бисмо их користили при следећем наиласку на неки крај дужи. Основни део посла је дефинисање информација које треба прикупљати. Покушајмо да покренемо алгоритам да бисмо открили које су то информације потребне.

Природно је у индуктивну хипотезу укључити познавање свих пресечних тачака дужи које се налазе лево од тренутног положаја покретне праве. Да ли је боље проверавати пресеке кад се разматра хоризонтална или вертикална дуж? Кад разматрамо вертикалну дуж, хоризонталне дужи које је могу сећи још увек се разматрају (пошто није достигнут њихов десни крај). С друге стране, кад посматрамо било леви, било десни крај хоризонталне дужи, ми или нисмо наишли на вертикалне дужи које је секу, или смо их већ заборавили. Дакле, боље је пресеке бројати приликом наиласка на вертикалну дуж. Претпоставимо да је покретна права тренутно преклопила вертикалну дуж L (видети слику 16). Какве информације су потребне да се пронађу сви пресеци дужи L? Пошто се претпоставља да су сви пресеци лево од тренутног положаја покретне праве већ познати, нема потребе разматрати хоризонталну дуж ако је њен десни крај лево од покретне праве. Према томе, треба разматрати само оне хоризонталне дужи чији су леви крајеви лево, а десни крајеви десно од тренутног положаја покретне праве (на слици 16 таквих дужи има шест). Потребно је чувати листу таквих хоризонталних дужи. Кад се наиђе на вертикалну дуж L, потребно је проверити да ли се она сече са тим хоризонталним дужима. Важно је приметити да за налажење пресека са L овде x-координате крајева дужи нису од значаја. Ми већ знамо да хоризонталне дужи из листе имају x-координате које "покривају" x-координату дужи L. Потребно је проверити само y-координате хоризонталних дужи из листе, да би се проверило да ли су оне обухваћене опсегом y-координата дужи L. Сада смо спремни да формулишемо индуктивну хипотезу.

Индуктивна хипотеза. Нека је задата листа од k сортираних x-координата крајева дужи као што је описано, при чему је x_k највећа од x-координата. Умемо да пронађемо све пресеке дужи који су лево од x_k, и да елиминишемо све хоризонталне дужи које су лево од x_k.

За хоризонталне дужи које се још увек разматрају рећи ћемо да су **кандидати** (то су хоризонталне дужи чији су леви крајеви лево, а десни крајеви десно од текућег положаја покретне праве). Формираћемо и одржавати структуру података која садржи скуп кандидата. Одложићемо за тренутак анализу реализације ове структуре података.

Базни случај за наведену индуктивну хипотезу је једноставан. Да бисмо је прошириле, потребно је да обрадимо $(k+1)$-и крај дужи. Постоје три могућности.

(1) $(k+1)$-и крај дужи је десни крај хоризонталне дужи; дуж се тада просто елиминише из списка кандидата. Као што је речено, пресеци се проналазе при разматрању вертикалних дужи, па се ни један од пресека не губи елиминацијом хоризонталне дужи. Овај корак дакле проширује индуктивну хипотезу.

(2) $(k+1)$-и крај дужи је леви крај хоризонталне дужи; дуж се тада додаје у списак кандидата. Пошто десни крај дужи још није достигнут, дуж се не сме још елиминисати, па је и у овом случају индуктивна хипотеза проширена на исправан начин.

(3) $(k+1)$-и крај дужи је вертикална дуж. Ово је основни део алгоритма. Пресеци са овом вертикалном дужи могу се пронаћи упоређивањем y-координата свих хоризонталних дужи из скупа кандидата са y-координатама крајева вертикалне дужи.

Алгоритам је сада комплетан. Број упоређивања ће обично бити много мањи од mn. На жалост, у најгорем случају овај алгоритам ипак захтева $O(mn)$ упоређивања, чак иако 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 ни једна вертикална дуж не сече ни једну хоризонталну дуж.

Да би се усавршио алгоритам, потребно је смањити број упоређивања y-координата вертикалне дужи са y-координатама хоризонталних дужи у скупу кандидата. Нека су y-координате дужи која се тренутно разматра y_D и y_G, и нека су y-координате хоризонталних дужи из скупа кандидата $y_1, y_2, \ldots, y_r$. Претпоставимо да су хоризонталне дужи у скупу кандидата задате сортирано према y-координатама (односно низ $y_1, y_2, \ldots, y_r$ је растући). Хоризонталне дужи које се секу са вертикалном могу се пронаћи извођењем две бинарне 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 са координатама y_{i+1}, $y_{i+2}, \ldots y_j$, и само оне. Може се такође извршити једна бинарна претрага за нпр. y_D, а затим пролазити y-координате док се не дође до y_j. Иако је полазни проблем дводимензионалан, налажење $y_{i+1}, \ldots y_j$ је једнодимензионални проблем. Тражење бројева у једнодимензионалном опсегу (у овом случају од y_D до y_G) зове се *једнодимензионална претрага опсега.* Ако су бројеви сортирани, онда је време извршења једнодимензионалне претраге опсега 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.

Присетимо се још једном захтева. Потребна је структура података погодна за чување кандидата, која дозвољава уметање новог елемента, брисање елемента и ефикасно извршење једнодимензионалне претраге опсега. На срећу, постоји више структура података — на пример, уравнотежено стабло — која омогућују извршење уметања, брисања и тражења сложености $O(\log n)$ по операцији (где је n број елемената у скупу), и линеарно претраживање за време

пропорционално броју пронађених елемената. Алгоритам је приказан на слици 17.

Алгоритам Preseci$((v_1, v_2, \ldots, v_m), (h_1, h_2, \ldots, h_n))$;
Улаз: $v_1, v_2, \ldots, v_m$ (скуп вертикалних дужи), и$h_1, h_2, \ldots, h_n$ (скуп хоризонталних дужи).
{$y_G(v_i)$ и $y_G(v_i)$ су мања и већа y-координата дужи v_i}
Излаз: Скуп свих парова дужи које имају пресек.
begin
  Сортирати све x-координате у неопадајући низ и сместити их у Q;
  $V := \emptyset$;
  {V је скуп хоризонталних дужи, тренутних кандидата за пресеке; организован}
  {је као уравнотежено стабло према y-координатама хоризонталних дужи}
  while Q је непразан **do**
    уклонити из Q наредни крај дужи, p;
    if p је десни крај дужи h_k **then**
      уклонити h_k из V
    else if p је леви крај дужи h_k **then**
      укључити h_k у V
    else if p је x-координата вертикалне дужи v_i **then**
      {налажење пресека кандидата са вертикалном дужи}
      једнодимензионална претрага опсега од $y_G(v_i)$ до $y_G(v_i)$ у V
end

Рис. 17. Алгоритам за налажење пресека хоризонталних и вертикалних дужи у равни.

Сложеност. Почетно сортирање према x-координатама крајева дужи захтева $O((m+n)\log(m+n))$ корака. Пошто свако уметање и брисање захтева $O(\log n)$ корака, укупно време за обраду хоризонталних дужи је $O(n \log n)$. Обрада вертикалних дужи захтева једнодимензионалну претрагу опсега, која се може извршити за време $O(\log n + r)$, где је r број пресека ове вертикалне дужи. Временска сложеност алгоритма је дакле

$$O((m+n)\log(m+n) + R),$$

где је R укупан број пресека.

7.7. Резиме

Геометријски алгоритми су на неки начин мање апстрактни од графовских, јер смо навикли да видимо и радимо са геометријским објектима. Међутим, то је само први утисак. Радити са огромним бројем објеката није исто што и радити са малим сликама, па морамо бити опрезни да нас представа коју имамо не

наведе на погрешне закључке. Морају се предвидети многи специјални случајеви. Алгоритам за утврђивање припадности тачке унутрашњости многоугла (одељак 7.2) је добар пример. Због тога конструкција геометријских алгоритама захтева посебан опрез.

Технике за конструкцију (дискретних) геометријских алгоритама сличне су техникама које смо разматрали у претходним поглављима. Индукција у томе игра важну улогу. Техника покретне праве, заснована на индукцији, заједничка је за више геометријских алгоритама; један од њих приказан је у одељку 7.6. Такође се често користи декомпозиција. Геометријски алгоритми (изузев најједноставнијих) често захтевају компликоване структуре података, па су многе сложене структуре података развијене управо за ту намену. Овде нисмо разматрали ни једну од таквих специјалних структура података.

Задаци

7.1. Прецизирати алгоритам *Tačka_u_mnogouglu* (слика 7.2), тако да буду обухваћени специјални случајеви кад полуправа L садржи теме или страницу многоугла.

7.2. Конструисати алгоритам који утврђује да ли је датих n тачака у равни колинеарно. Колика је временска сложеност тог алгоритма?

7.3. Нека је S произвољан скуп тачака у равни. Да ли постоји само један многоугао са скупом темена S? Ако је тврђење тачно, доказати га, а у противном га оповргнути контрапримером.

7.4. Дат је конвексни многоугао $P_0P_1\dots P_{n-1}$ и тачка A ван њега. Како се могу конструисати две праве ослонца многоугла кроз тачку A алгоритмом сложености $O(\log n)$.

7.5. Нека су тачке $P_1P_2\dots P_n$ уређене циклички у односу на круг са центром негде унутар конвексног омотача ових тачака. Модификовати Грахамов алгоритам тако да може да ради са овим скупом тачака без почетног сортирања.

7.6. Показати да Грахамов алгоритам може да одбаци узастопце p тачака за произвољно велико p.

7.7. Навести пример n тачака у равни са различитим x-координатама за које извршавање алгоритма $Najbliži_par2$ са слике 15 траје $\Omega(n\log n)$ корака.

7.8. Дат је скуп од n хоризонталних дужи у равни. Пронаћи све њихове пресеке. Алгоритам треба да буде сложености $O(n\log n)$ у најгорем случају.

7.9. Дато је n тачака у равни и права p. Конструисати алгоритам линеарне сложености за налажење праве паралелне са p која дати скуп тачака разбија на два подскупа једнаке величине (ако тачка лежи на тој правој, може се урачунати у било који од подскупова).

7.10. Нека је P прост (не обавезно конвексни) многоугао садржан у датом правоугаонику R, и нека је q произвољна тачка унутар R. Конструисати ефикасни алгоритам за одређивање такве тачке r ван R да је број ивица P које сече дуж $q-r$ минималан.

7.11. Нека је P конвексни многоугао задат низом темена (наведених редом којим су повезана). Конструисати алгоритам временске сложености $O(\log n)$ који треба да за дату тачку q установи да ли је унутар P.

7.12. Конструисати алгоритам за формирање конвексног омотача скупа од n тачака у равни, који одговара сортирању уметањем. У свакој итерацији треба разматрати нову тачку и евентуално је укључити у конвексни омотач претходно размотрених тачака. Тачке треба разматрати произвољним редоследом (није дозвољено почетно сортирање). Алгоритам

треба да користи ефикасну структуру података за утврђивање да ли је дата тачка унутар конвексног многоугла. Колика је временска сложеност алгоритма у најгорем случају?

7.13. Претпоставимо да нам је на располагању црна кутија која проналази конвексни омотач уније два дисјунктна конвексна многоугла P_1 и P_2 за време пропорционално збиру броја њихових темена. Конструисати алгоритам временске сложености $O(n \log n)$ који користи ову црну кутију да би одредио конвексни омотач датог скупа од n тачака у равни.

7.14. **d-приближан конвексни омотач** скупа тачака P је конвексни многоугао Q, чија су сва темена из P, при чему су све тачке из P или у Q, или на растојању највише d од Q (растојање тачке од многоугла је најмање од растојања тачке до неке тачке у многоуглу). Нека је P скуп од n тачака такав да је највећа разлика између x-координата неке две тачке у P једнака X. Конструисати алгоритам за налажење d-приближног конвексног омотача скупа P, временске и просторне сложености $O(n + X/d)$.

7.15. (а) За тачку p у равни каже се да **доминира** над тачком q ако су x- и y-координата тачке p веће од одговарајућих координата тачке q. Тачка p је **максимална** у датом скупу тачака P ако ни једна тачка из P не доминира над њом. Конструисати алгоритам сложености $O(n \log n)$ за налажење свих максималних тачака датог скупа P од n тачака.

(б) Решити одговарајући проблем у три димензије (дефиниција доминирања треба да обухвати све три координате).

7.16. Нека је S скуп тачака у равни. За свако $p \in S$ нека је $D(p)$ скуп тачака из S над којима доминира p (видети задатак 7.15). Конструисати алгоритам временске сложености $O(n \log n)$ за израчунавање величина свих скупова $D(p)$, $p \in S$.

7.17. Дато је n тачака у равни. Одредити међу њима такве две, да дуж која их повезује има највећи нагиб. Сложеност алгоритма треба да буде $O(n \log n)$.

7.18. Дато је n тачака у равни, представљених вектором повезаних листа на следећи начин. Сваки елеменат вектора има два поља: X, које садржи x-координату, и $Naredni$, које показује на (непразну) повезану листу свих тачака скупа са x-координатом једнаком X, сортираних према својим y-координатама. Вектор је сортиран према x-координатама. Конструисати алгоритам сложености $O(n)$ који проналази најближи пар тачака са једнаким x-координатама, или са x-координатама које су суседне у низу. Да ли је у оквиру алгоритма неопходно израчунавање квадратних коренова? Да ли алгоритам проналази најближи пар тачака (без икаквих ограничења)?

7.19. Дат је скуп дужи у равни, које са x-осом заклапају угао $k\pi/4$, $k = 0, 1, 2, 3$. Конструисати алгоритам за налажење свих пресека ових дужи исте сложености као алгоритам за налажење пресека хоризонталних и вертикалних дужи са слике 17.

7.20. Дато је n дужи у равни. Конструисати алгоритам сложености $O(n \log n)$ који утврђује да ли међу њима постоји бар један пресек.

7.21. На правој је дато n плавих и n црвених дужи. Конструисати алгоритам сложености $O(n \log n)$ за проверу да ли постоји заједничка тачка неке плаве и неке црвене дужи (може се претпоставити да не постоје две дужи са заједничким теменом).

7.22. Дат је скуп интервала на правој, представљених координатама крајева. Конструисати алгоритам сложености $O(n \log n)$ за налажење свих интервала садржаних у неком другом интервалу из скупа.

7.23. Дато је n правоугаоника у равни са страницама паралелним осама. Помоћу алгоритма из задатка 7.22 пронаћи све правоугаонике садржане у неком другом правоугаонику. Можете ли да конструишете алгоритам сложености $O(n \log n)$?

7.24. Дато је n правоугаоника у равни са страницама паралелним осама. Конструисати алгоритам за налажење пресека свих тих правоугаоника.

7.25. Дато је n кружница у равни. Конструисати алгоритам сложености $O(n \log n)$ који утврђује да ли се неке две од тих кружница секу.

7.26. Дат је скуп од n k-углова. Конструисати алгоритам који проверава да ли међу њима постоје нека два са непразним пресеком. Колика његова сложеност алгоритма?

7.27. Дата су два конвексна многоугла листама својих темена (цикличким редоследом). Конструисати алгоритам линеарне временске сложености за одређивање пресека ових многоуглова. Излаз (такође конвексни многоугао) треба да буде представљен циклички уређеном листом темена.

7.28. Дато је n троуглова у равни (тројкама својих темена). Конструисати алгоритам сложености $O(n \log n)$ за налажење њиховог пресека.

7.29. Конвексни n-тоугао дат је циклички уређеном листом својих темена. Конструисати алгоритам линеарне временске сложености за одређивање n троуглова чији је пресек дати многоугао.

7.30. Конструисати алгоритам сложености $O(n^2 \log n)$, који утврђује да ли су колинеарне неке три од задатих n тачака у равни.

7.31. Конструисати алгоритам сложености $O(n^2 \log n)$, који утврђује да ли постоји квадрат са теменима у неке четири од задатих n тачака у равни.

7.32. Дато је n тачака у равни са целобројним координатама. Потребно је одредити најмањи скуп правих, паралелних било некој оси, било некој симетрали квадранта, а који садржи свих n тачака.

Глава 8

Алгебарски алгоритми

8.1. Увод

Кад год извршимо неку алгебарску операцију, ми уствари извршавамо један алгоритам. До сада смо те операције прихватали као нешто обично, као алгоритме који постоје сами по себи. Међутим, без обзира да ли је у питању множење, дељење или нека сложенија алгебарска операција, уобичајени алгоритам није увек најбољи ако се ради са врло великим бројевима или низовима бројева. Иста појава на коју смо наилазили у претходним поглављима испољава се и овде: алгоритми који су добри за мали улаз постају неефикасни за веће улазе.

Сложеност алгоритма мери се бројем "операција" које алгоритам извршава. У највећем броју случајева претпостављамо да се основне аритметичке операције (као што су сабирање, множење, дељење) извршавају за јединично време. То је разумна претпоставка кад се операнди могу представити са једном или две рачунарске речи (на пример, не превелики цели бројеви, реални бројеви једноструке или двоструке тачности). Постоје међутим ситуације кад су операнди огромни, на пример бројеви са 2000 цифара. Тада се мора узмети у обзир и величина операнада, а основне операције престају да буду једноставне. Ако се игноришу величине операнада, могуће је да на први поглед добар алгоритам буде уствари врло неефикасан.

Значење "величине улаза" понекад није сасвим јасно. Нека је дат цели број n са којим треба извршити неку аритметичку операцију. На први поглед изгледа природно да се за величину улаза сматра сама вредност n. Ово се, међутим, не уклапа у неформалну дефиницију величине улаза као мери величине меморијског простора потребног за његово смештање. Разлика је врло велика. Сабирање два броја од по 100 цифара може се извршити врло брзо, чак и ручно. С друге стране, бројање до вредности представљене 100-цифреним бројем практично је немогуће, чак и на најбржем рачунару. Пошто се број n може представити са $\lceil \log_2 n \rceil$ бита, за његову **величину** може се сматрати $\lceil \log_2 n \rceil = O(\log n)$. Тако се алгоритам који захтева $O(\log n)$ операција кад је улаз n сматра линеарним (на пример израчунавање $2n$), јер је $O(\log n)$ линеарна функција величине улаза. Алгоритам који захтева извршавање $O(\sqrt{n})$ операција за улаз n (на пример факторизација n провером дељивости n свим бројевима мањим или једнаким од $\sqrt{n}$) има експоненцијалну сложеност.

Као и обично, у овом поглављу ћемо пажњу усмерити на интересантне технике конструкције алгоритама. Најпре се разматра степеновање датог броја, затим вероватно најстарији нетривијални алгоритам, Еуклидов алгоритам за израчунавање највећег заједничког делиоца два броја. Врло је интересантно да модерни рачунари користе 2200 година стар алгоритам. Ова два алгоритма користе се у познатом асиметричном шифарском систему RSA, видети следеће поглавље. Затим се разматрају алгоритми за множење полинома и матрица, а поглавље се завршава једним од најважнијих и најлепших алгоритама — брзом Фуријеовом трансформацијом.

8.2. Степеновање

Започињемо једном од основних аритметичких операција.

Проблем. Дата су два природна броја n и k. Израчунати n^k.

Проблем се лако може свести на израчунавање n^{k-1}, јер је $n^k = n \cdot n^{k-1}$. Према томе, проблем се може решити индукцијом по k; добијени директни алгоритам приказан је на слици 1. Смањена је вредност k, али не и његова величина. Тривијални алгоритам захтева k множења. Пошто је величина податка k приближно $\log_2 k$, број итерација је експоненцијална функција величине k ($k = 2^{\log_2 k}$). Ово није лоше за мале, али је неприхватљиво за велике вредности k.

Алгоритам Stepen(n, k); {први покушај}
Улаз: n и k (два природна броја).
Излаз: P (вредност израза n^k).
begin
  $P := n$;
  for $i := 1$ **to** $k-1$ **do**
    $P := n \cdot P$
end

Рис. 1. Тривијални алгоритам за степеновање.

Други начин да се проблем реши је свођење на проблем са двоструко мањим експонентом помоћу једнакости $n^k = (n^{k/2})^2$. Половљење k одговара смањењу његове величине за константу. Због тога ће број множења бити линеарна функција од величине k. Најједноставнији случај је $k = 2^j$, за неки природан број j:

$$n^k = n^{2^j} = \overbrace{\left(\left(n^2\right)^2\right)^{\cdots 2}}^{j \text{ пута}}.$$

Шта ако k није степен двојке? Размотримо још једном примењени поступак редукције. Пошавши од параметара n и k, проблем је сведен на мањи, са

параметрима n и $k/2$. Ако $k/2$ није цели број, онда $(k-1)/2$ јесте, па се може применити слична редукција:

$$n^k = n\left(n^{(k-1)/2}\right)^2.$$

Сада је алгоритам комплетиран. Ако је k парно, једноставно квадрирамо резултат степеновања изложиоцем $k/2$. Ако је пак k непарно, квадрирамо резултат степеновања изложиоцем $(k-1)/2$ и множимо га са n. Број множења је највише $2\log_2 k$. Алгоритам је приказан на слици 2.

```
Алгоритам Stepen_kvadriranjem(n, k);
Улаз: n и k (два природна броја).
Излаз: P (вредност израза n^k).
begin
    if k = 1 then P := n
    else
        z := Stepen_kvadriranjem(n, k div 2);
        if k mod 2 = 0 then
            P := z · z
        else
            P := n · z · z
end
```

Рис. 2. Степеновање квадрирањем.

Сложеност. Укупан број множења је $O(\log k)$. Међутим, како се напредује са извршавањем алгоритма, међурезултати постају све већи, па множења постају све компликованија. Остављамо читаоцу да анализира сложеност овог алгоритма под реалнијом претпоставком о сложености множења (видети задатак 8.8). Ако се множење обавља у прстену остатака по неком модулу, онда међурезултати не расту у току извршења алгоритма и тачна је наведена оцена временске сложености степеновања; једну примену овакво степеновање је нашло у оквиру алгоритма за шифровање RSA (видети одељак 9.5).

8.3. Еуклидов алгоритам

Највећи заједнички делилац два природна броја n и m (означава се са NZD(n,m)) је јединствени природни број d који 1) дели m и n, и 2) већи је или једнак од сваког другог природног броја d' који дели n и m.

Проблем. Одредити највећи заједнички делилац два дата природна броја.

Као и обично, идеја је свести полазни проблем на проблем са мањим улазом. Могу ли се n или m некако смањити, а да се резултат не промени? Еуклид

је приметио да је то могуће: ако d дели n и m, онда дели и њихову разлику. Важи и обрнуто; ако је на пример $n \geq m$, а d дели m и $n-m$, онда d дели и збир $m+(n-m)=n$. Другим речима, $\mathrm{NZD}\,(n,m) = \mathrm{NZD}\,(n-m,m)$, и добијен је мањи улаз. Међутим, одузимањем су смањене *вредности* бројева са којима се ради, али не и њихове *величине*. Да би алгоритам био ефикасан, морају се смањити величине бројева. На пример, ако је n врло велики број (нпр. 1000 цифара) и $m=24$, од n треба одузети 24 приближно $n/24$ пута. Ово рачунање састоји се од $O(n)$ корака, што је експоненцијална функција од величине n.

Размотримо ову идеју још једном. После одузимања m од n исти алгоритам треба применити на $n-m$ и m. Ако је $n-m$ и даље веће од m, од њега треба одузети m. Са одузимањем m наставља се све док разлика не постане мања од m. Број оваквих одузимања може бити огроман; срећом, резултат који се тако добија је једнак остатку $n \bmod m$ при дељењу n са m. Према томе, $\mathrm{NZD}\,(n,m) = \mathrm{NZD}\,(n \bmod m, m)$. Дељење са остатком извршава се ефикасно.

Полазећи од бројева $r_0 = n$ и $r_1 = m$, израчунава се остатак $r_2 = r_0 \bmod r_1$, затим остатак $r_3 = r_1 \bmod r_2$, итд. Тако се добија опадајући низ остатака

r0 = q1 r1 + r2   0<=r2<r1

$$r_{i-1} = q_i r_i + r_{i+1}, \quad 0 \leq r_{i+1} < r_i, \text{ за } i = 1,2,\ldots,k \tag{8.1}$$

r2 <= r0/2:

(при дељењу r_{i-1} са r_i количник је q_i, а остатак r_{i+1}). Добијени низ је коначан јер је опадајући и састоји се од природних бројева. Нека је нпр. $r_{k+1}=0$ и нека је $r_k \neq 0$ последњи члан овог низа различит од нуле. Како је

g1 = 1   q1>1

$$\mathrm{NZD}\,(r_0,r_1) = \mathrm{NZD}\,(r_1,r_2) = \ldots = \mathrm{NZD}\,(r_{k-1},r_k) = \mathrm{NZD}\,(r_k,0) = r_k,$$

видимо да је $d = \mathrm{NZD}\,(n,m)$ управо једнако r_k, последњем остатку различитом од нуле. Описани поступак за израчунавање највећег заједничког делиоца два броја зове се Еуклидов алгоритам, видети слику 3.

Алгоритам NZD(m,n);
Улаз: m и n (два природна броја).
Излаз: nzd (највећи заједнички делилац бројева m и n).
begin
  $a := \max(n,m)$;
  $b := \min(n,m)$;
  $r := 1$; {вредност која омогућује улазак у петљу}
  while $r > 0$ **do** {r је остатак}
    $r := a \bmod b$;
    $a := b$;
    $b := r$;
  $nzd := a$
end

Рис. 3. Еуклидов алгоритам.

Сложеност. Тврдимо да Еуклидов алгоритам има линеарну временску сложеност у односу на величину броја $m+n$; другим речима, његова временска сложеност је $O(\log(m+n))$ (ако се претпостави да се све елементарне операције извршавају за јединично време, независно од величине операнада). Да се то докаже, довољно је доказати да вредност a у алгоритму постаје бар два пута мања после две итерације. После прве итерације пар (a, b) $(a > b)$ замењује се паром $(b, a \bmod b)$, а после друге итерације паром $(a \bmod b, b \bmod (a \bmod b))$. Према томе, после две итерације је број a замењен бројем $a \bmod b$, који је увек мањи од $a/2$. Заиста, ако $b \le a/2$ онда је $a \bmod b < b \le a/2$; у противном, за $b > a/2$, такође се добија $a \bmod b = a - b = a/2 - (b - a/2) < a/2$.

Уз малу допуну, Еуклидов алгоритам се може искористити и за решавање следећег проблема.

Проблем. Највећи заједнички делилац d два природна броја n и m изразити као њихову целобројну линеарну комбинацију. Другим речима, одредити целе бројеве x, y, тако да важи $d = \mathrm{NZD}\,(n, m) = nx + my$.

Циљ је d изразити у облику линеарне комбинације $d = r_0 x + r_1 y$ остатака $r_0 = n$ и $r_1 = m$. Проблем се може решити индукцијом. Полази се од базе, израза за d у облику линеарне комбинације остатака $r_{k-1} = n$ и $r_{k-2} = m$: $d = r_k = r_{k-2} - q_{k-1} r_{k-1}$; овај израз је еквивалентан претпоследњем дељењу у Еуклидовом алгоритму (израз (8.1) за $i = k - 1$). Корак индукције био би изражавање d у облику линеарне комбинације остатака r_{i-1} и r_i, полазећи од израза $d = x' r_i + y' r_{i+1}$ — целобројне линеарне комбинације r_i и r_{i+1}. Заиста, заменом r_{i+1} у овом изразу са $r_{i+1} = r_{i-1} - q_i r_i$, добија се

r0, r1, r2, ..., r k-4 r k-3 r k-2 r k-1 r k

$$d = x' r_i + y'(r_{i-1} - q_i r_i) = y' r_{i-1} + (x' - q_i y') r_i,$$

тј. израз за d у облику целобројне линеарне комбинације остатака r_{i-1} и r_i. Дакле, индукцијом по i, $i = k-2, k-1, \dots, 1, 0$ доказано је да се d може изразити као целобројна линеарна комбинација било која два узастопна члана r_i, r_{i+1} низа остатака. Специјално, за $i = 0$, добија се тражени израз.

Сложеност. Број операција је пропорционалан са бројем операција у Еуклидовом алгоритму, тј. $O(m + n)$.

Значај ове допуне Еуклидовог алгоритма је у томе што она омогућује решавања тзв. *линеарних Диофантових једначина* $ax + by = c$, где су a, b, c дати цели бројеви, а x и y су цели бројеви које треба одредити. Без смањења општости може се претпоставити да је $a, b > 0$. Да би наведена једначина имала бар једно решење, потребно је да $d = \mathrm{NZD}\,(a, b)$ дели c (пошто d дели леву страну једначине, мора да дели и десну). Ако је овај услов испуњен, једно од решења лако се добија описаним поступком. Пошто се d изрази у облику $d = ax' + by'$, множењем са целим бројем c/d добија се $c = a(x'c/d) + b(y'c/d)$, тј. види се да је једно решење једначине пар $(x, y) = (x'c/d, y'c/d)$. Пример одређивања бројева x и y, коефицијената линеарне комбинације $d = nx + my$, дат је у одељку 9.5.

8.4. Множење полинома

Нека су $P = \sum_{i=0}^{n-1} p_i x^i$ и $Q = \sum_{i=0}^{n-1} q_i x^i$ два полинома степена $n-1$. Полином је представљен низом својих коефицијената.

Проблем. Израчунати производ два задата полинома степена $n-1$.

Природно је поћи од израза

$$PQ = \left(p_{n-1}x^{n-1} + \cdots + p_0\right)\left(q_{n-1}x^{n-1} + \cdots + q_0\right) = \tag{8.2}$$
$$p_{n-1}q_{n-1}x^{2n-2} + \cdots + \left(p_{n-1}q_{i+1} + p_{n-2}q_{i+2} + \cdots + p_{i+1}q_{n-1}\right)x^{n+i} + \cdots + p_0q_0.$$

Коефицијенти полинома PQ могу се израчунати директно из (8.2), при чему је јасно да ће тада број множења и сабирања бити $O(n^2)$. Може ли се исти посао обавити ефикасније? До сада смо видели више примера да се тривијални квадратни алгоритми могу побољшати, па није изненађујуће да је и у овом случају одговор позитиван. Компликовани алгоритам сложености $O(n \log n)$ биће размотрен у одељку 8.6. Овде ћемо размотрити једноставан алгоритам заснован на декомпозицији.

Претпоставимо због једноставности да је n степен двојке. Сваки од полинома делимо на два једнака дела. Нека је дакле $P = P_1 + x^{n/2}P_2$ и $Q = Q_1 + x^{n/2}Q_2$, где је

$$P_1 = p_0 + p_1x + \cdots + p_{n/2-1}x^{n/2-1}, \quad P_2 = p_{n/2} + p_{n/2+1}x + \cdots + p_{n-1}x^{n/2-1},$$

односно

$$Q_1 = q_0 + q_1x + \cdots + q_{n/2-1}x^{n/2-1}, \quad Q_2 = q_{n/2} + q_{n/2+1}x + \cdots + q_{n-1}x^{n/2-1}.$$

Сада имамо

$$PQ = (P_1 + P_2x^{n/2})(Q_1 + Q_2x^{n/2}) = P_1Q_1 + (P_1Q_2 + P_2Q_1)x^{n/2} + P_2Q_2x^n.$$

У изразу за PQ појављују се производи полинома степена $n/2-1$, који се могу израчунати индукцијом (рекурзивно). Сабирањем добијених резултата добија се решење. Узимајући у обзир да је множење полинома степена 0 исто што и множење бројева, овим је комплетно дефинисан рекурзивни алгоритам за множење полинома. Укупан број операција $T(n)$ које се извршавају у оквиру овог алгоритма задовољава следећу диференцну једначину:

$$T(n) = 4T(n/2) + O(n), \quad T(1) = 1.$$

Фактор 4 одговара израчунавању четири производа мањих полинома, а члан $O(n)$ одговара сабирању производа. Решење диференцне једначине је $T(n) = O(n^2)$ (видети одељак 2.5.4), па овај алгоритам није бољи од претходног.

Да би се дошло до побољшања у односу на квадратни алгоритам, потребно је, на пример, да проблем решимо свођењем на мање од четири потпроблема. Означимо производе P_1Q_1, P_2Q_1, P_1Q_2, P_2Q_2 редом са A, B, C, D. Треба да израчунамо $A + (B+C)x^{n/2} + Dx^n$. Запажа се да нису неопходни сами производи B и C, него само њихов збир. Ако знамо производ $E = (P_1 + P_2)(Q_1 + Q_2)$, онда је тражени збир $B + C = E - A - D$. Дакле, довољно је израчунати само три производа мањих полинома: A, D и E. Све остало су сабирања и

одузимања полинома, што ионако улази у члан $O(n)$ у диференцној једначини. Диференцна једначина за сложеност побољшаног алгоритма је

$$T(n) = 3T(n/2) + O(n),$$

а њено решење је $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$.

Запажамо да су полиноми P_1+P_2 и Q_1+Q_2 са полазним полиномима везани необичан начин: добијени су од њих сабирањем коефицијената чији индекси се разликују за $n/2$. Овај неинтуитивни начин множења полинома знатно смањује број операција за велике вредности n.

Пример 8.1. Нека је $n = 4$, $P = 1 - x + 2x^2 - x^3$ и $Q = 2 + x - x^2 + 2x^3$. Израчунајмо производ PQ на описани начин. Ако се линеарни полиноми множе директно, рекурзија се примењује само једном,

$$A = (1 - x)(2 + x) = 2 - x - x^2,$$

$$D = (2 - x)(-1 + 2x) = -2 + 5x - 2x^2,$$

$$E = (3 - 2x)(1 + 3x) = 3 + 7x - 6x^2.$$

На основу E, A и D израчунава се $B + C = E - A - D$:

$$B + C = 3 + 3x - 3x^2.$$

Сада је $PQ = A + (B + C)x^{n/2} + Dx^n$, односно

$$\begin{aligned} PQ &= (2 - x - x^2) + (3 + 3x - 3x^2)x^2 + (-2 + 5x - 2x^2)x^4 = \\ &= 2 - x + 2x^2 + 3x^3 - 5x^4 + 5x^5 - 2x^6. \end{aligned}$$

Примећује се да је извршено 12 множења, у односу на 16 код тривијалног алгоритма, и 12 уместо 9 сабирања/одузимања; број множења био би сведен на 9 да је рекурзија примењена још једном. Уштеда је наравно много већа за велике n.

8.5. Множење матрица

Ако су $P = (p_{ij})$ и $Q = (q_{ij})$ дате квадратне матрице реда n, онда је елеменат r_{ij} матрице $R = PQ$ дат је изразом

$$r_{ij} = \sum_{k=1}^{n} p_{ik} q_{kj}. \tag{8.3}$$

Проблем. Израчунати производ $R = PQ$ две реалне $n \times n$ матрице.

Директни (и на први поглед *једини*) поступак множења матрица заснива се на дефиницији (8.3), што подразумева n^3 множења и $(n-1)n^2$ сабирања. Запазимо да је n број врста, односно колона матрице, а не величина улаза, која је у овом случају n^2. Приказаћемо сада два различита побољшања овог алгоритма.

8.5.1. Виноградов алгоритам. (S. Winograd) Претпоставимо због једноставности да је n парно. Уведимо ознаке

$$P_i = \sum_{k=1}^{n/2} p_{i,2k-1} p_{i,2k}, \quad i = 1, 2, \dots, n,$$

$$Q_j = \sum_{k=1}^{n/2} q_{2k-1,j} q_{2k,j}, \quad j = 1, 2, \dots, n.$$

Прегруписавањем сабирака добија се

$$r_{ij} = \sum_{k=1}^{n/2} (p_{i,2k-1} + q_{2k,j})(p_{i,2k} + q_{2k-1,j}) - P_i - Q_j.$$

Бројеви P_i и Q_j израчунавају се само једном за сваку врсту P, односно колону Q, за шта је потребно само n^2 множења. Укупан број множења је дакле смањен на $n^3/2 + n^2$. Број сабирања повећан је приближно за $n^3/2$. Алгоритам је према томе бољи од директног у случају кад се сабирања извршавају брже од множења (што је типично).

Коментар. Виноградов алгоритам показује да се променом редоследа израчунавања може постићи уштеда, чак и код израза као што је производ матрица, који имају једноставан облик. Следећи алгоритам исту идеју експлоатише много ефикасније.

8.5.2. Штрасенов алгоритам. На множење матрица може се применити поступак декомпозиције, слично као на множење полинома, одељак 8.4. Због једноставности претпоставићемо да је n степен двојке. Нека је

$$P = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad Q = \begin{pmatrix} A & B \\ C & D \end{pmatrix}, \tag{8.4}$$

aA+bC aB+bD
cA+dC cB+dD

где су a, b, c, d, односно A, B, C, D $n/2 \times n/2$ матрице. Применом декомпозиције се проблем своди на израчунавање четири $n/2 \times n/2$ подматрице матрице R. Производ блок матрица израчунава се на исти начин као кад се блокови замене елементима, па се проблем може схватити као тражење ефикасног начина за израчунавање производа две 2×2 матрице. Од алгоритма за множење 2×2 матрица добија се алгоритам за множење $n \times n$ матрица тако што се уместо производа елемената уметну рекурзивни позиви процедуре за множење. Обичан алгоритам за множење 2×2 матрица користи 8 множења. Замењујући свако множење рекурзивним позивом, добијамо диференцну једначину $T(n) = 8T(n/2) + O(n^2)$ (комбиновање мањих решења састоји се од неколико сабирања матрица реда $n/2$, сложености $O(n^2)$), чије је решење $T(n) = O(n^{\log_2 8}) = O(n^3)$. Ово није изненађујуће, јер се користи обичан алгоритам за множење. Ако бисмо успели да израчунамо производ 2×2 матрица изводећи мање од 8 множења елемената, добили бисмо алгоритам који је асимптотски бржи од кубног.

Фактор који највише утиче на рекурзију је број множења потребних за израчунавање производа 2×2 матрица. Број сабирања није тако важан, јер мења само чинилац уз члан $O(n^2)$ у диференцној једначини, па не утиче на асимптотску сложеност (он међутим утиче на константни фактор). Штрасен (Strassen) је открио да је довољно седам множења елемената да се израчуна производ две матрице реда два Ако матрице P, Q (8.4) схватимо као матрице реда два, онда се њихов производ може израчунати на следећи начин:

$$PQ = \begin{pmatrix} z_1 + z_4 & z_2 - z_3 + z_4 + z_5 \\ z_1 + z_3 + z_6 + z_7 & z_2 + z_6 \end{pmatrix},$$

при чему су са z_1, z_2, $\ldots$, z_7 означени следећи производи $z_1 = b(A + C)$, $z_2 = c(B + D)$, $z_3 = (c - b)(A + D)$, $z_4 = (a - b)A$, $z_5 = (a - c)(B - A)$, $z_6 = (d - c)D$ и $z_7 = (d - b)(C - D)$.

Сложеност. У алгоритму се израчунава седам производа матрица два пута мање димензије и константни број сабирања таквих матрица. Сабирања су мање важна од производа, јер се две матрице реда n сабирају за време $O(n^2)$, што је линеарна функција од величине улаза. Члан $O(n^2)$ није доминантан у диференцној једначини $T(n) = 7T(n/2) + O(n^2)$. Решење ове диференцне једначине је $T(n) = O(n^{\log_2 7})$, односно приближно $O(n^{2.81})$, што значи да је Штрасенов алгоритам асимптотски бржи од обичног множења матрица.

Коментар. Штрасенов алгоритам има три важна недостатка:

(1) Практичне провере показују да n мора бити веће од 100 да би Штрасенов алгоритам био бржи од обичног множења матрица сложености $O(n^3)$.

(2) Штрасенов алгоритам мање је стабилан од обичног. За исте величине грешке улазних података, Штрасенов алгоритам обично доводи до већих грешака у излазним подацима.

(3) Штрасенов алгоритам је компликованији и тежи за реализацију од обичног. Поред тога, њега није лако паралелизовати.

Без обзира на ове недостатке, Штрасенов алгоритам је веома важан. Бржи је од обичног за велике n, а може се искористити и у другим проблемима са матрицама, као што су инверзија матрице и израчунавање детерминанте. У поглављу 10 видећемо да се неколико других проблема своди на множење матрица. Штрасенов алгоритам се у пракси може побољшати коришћењем само за велике матрице, и изласком из рекурзије кад димензија матрице постане мања од око 100. То је слично идеји пажљивог избора базе индукције, која је размотрена у одељку 5.3.4. Штрасенов алгоритам је директно побољшао неке друге алгоритме и поставио много питања о сличним проблемима који су изгледали нерешиви.

8.6. Брза Фуријеова трансформација

Брза Фуријеова трансформација (или FFT, што је скраћеница од fast Fourier transform) је важна из више разлога. Она ефикасно решава важан практичан проблем, елегантна је, и отвара нове, неочекиване области примене. Због тога је она један од најважнијих алгоритама од свог открића средином шездесетих година.

Алгоритам FFT није једноставан, и до њега се не долази директно. Ограничићемо се на само једну његову примену, множење полинома.

Проблем. Израчунати производ два задата полинома $p(x)$ и $q(x)$.

Формулација проблема је прецизна само на први поглед, јер није прецизиран начин представљања полинома. Обично се полином

$$P = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1 x + a_0$$

представља низом својих коефицијената уз 1, x, x^2, ..., x^{n-1}; али то није једина могућност. Алтернатива је представљање полинома степена $n-1$ својим вредностима у n различитих тачака: те вредности једнозначно одређују полином. Други начин представљања је интересантан због једноставности множења. 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 дате у $2n-1$ тачака, онда се производ полинома израчунава помоћу $2n-1$, односно $O(n)$ обичних множења.

Нажалост, представљање полинома вредностима за неке примене није погодно. Пример је израчунавање вредности полинома у задатим тачкама; при репрезентацији вредностима, ово је много теже него ако су задати коефицијенти полинома. Међутим, ако бисмо могли да ефикасно преводимо полиноме из једне у другу представу, добили бисмо одличан алгоритам за множење полинома. Управо то се постиже применом FFT.

Прелаз од представе полинома коефицијентима на представу вредностима у тачкама, решава се израчунавањем вредности полинома. Вредност полинома $p(x)$ (задатог коефицијентима) у било којој тачки може се помоћу Хорнерове шеме (одељак 4.2) израчунати помоћу n множења. Израчунавање вредности $p(x)$ у n произвољних тачака изводљиво је дакле помоћу n^2 множења. Прелаз од представе полинома вредностима на представу коефицијентима зове се **интерполација**. Интерполација у општем случају такође захтева $O(n^2)$ операција. Овде је кључна идеја (као и у многим другим примерима које смо видели) да се не користи *произвољних* n тачака: ми имамо слободу да по жељи изаберемо *произвољан* скуп од n различитих тачака. Брза Фуријеова трансформација користи специјалан скуп тачака, тако да се обе трансформације, израчунавање вредности и интерполација, могу ефикасно извршавати.

8.6.1. Директна Фуријеова трансформација. Размотримо проблем израчунавања вредности полинома. Потребно је израчунати вредности два полинома степена $n-1$ у $2n-1$ тачака, да би се њихов производ, полином

степена $2n-2$, могао интерполирати. Међутим, полином степена $n-1$ може се представити као полином степена $2n-2$ изједначавањем са нулом водећих $n-1$ коефицијената. Због тога се без губитка општости може претпоставити да је проблем израчунати вредности произвољног полинома $P = \sum_{j=0}^{n-1} a_j x^j$ степена $n-1$ у n различитих тачака. Циљ је пронаћи таквих n тачака, у којима је лако израчунати вредности полинома. Због једноставности претпостављамо да је n степен двојке.

Користићемо матричну терминологију да бисмо упростили означавање. Израчунавање вредности полинома P у n тачака x_0, x_1, $\ldots$, x_{n-1} може се представити као израчунавање производа матрице и вектора:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \ldots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} P(x_0) \\ P(x_1) \\ \ldots \\ P(x_{n-1}) \end{pmatrix}.$$

Питање је како изабрати вредности x_0, x_1, $\ldots$, x_{n-1}, тако да се ово множење упрости. Посматрајмо две произвољне врсте r и s. Волели бисмо да их учинимо што сличнијим, да бисмо уштедели на множењима. Не може се ставити $x_r = x_s$, јер су тачке различите, али се $x_r^2 = x_s^2$ може постићи стављајући $x_s = -x_r$. Ово је добар избор, јер је сваки паран степен x_r једнак одговарајућем парном степену x_s; непарни степени разликују се само по знаку. Исто се може урадити и са осталим паровима врста. Наслућује се у ком правцу треба тражити n специјалних врста, за које би се горњи производ сводио на само $n/2$ производа врста матрице са колоном коефицијената. Резултат би био половљење величине улаза, а тиме и врло ефикасан алгоритам. Покушајмо да поставимо овај проблем као два одвојена проблема двоструко мање величине.

Подела полазног проблема на два потпроблема величине $m = n/2$ може се описати следећим изразом

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & x_{m-1} & x_{m-1}^2 & \ldots & x_{m-1}^{n-1} \\ 1 & -x_0 & (-x_0)^2 & \ldots & (-x_0)^{n-1} \\ 1 & -x_1 & (-x_1)^2 & \ldots & (-x_1)^{n-1} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & -x_{m-1} & (-x_{m-1})^2 & \ldots & (-x_{m-1})^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \ldots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} P(x_0) \\ P(x_1) \\ \ldots \\ P(x_{m-1}) \\ P(-x_0) \\ \ldots \\ P(-x_{m-1}) \end{pmatrix}. \tag{8.5}$$

Полазна $n \times n$ матрица подељена је на две врло сличне подматрице димензија $n/2 \times n$. За свако $j = 0, 1, \ldots, n/2-1$, важи $x_{n/2+j} = -x_j$. Згодно је дакле написати изразе за $P(x_j)$ и $P(-x_j)$, односно уопште $P(x)$, са раздвојеним члановима парног и непарног степена:

$$P(x) = \sum_{j=0}^{n/2-1} a_{2j} x^{2j} + \sum_{j=0}^{n/2-1} a_{2j+1} x^{2j+1}.$$

Ако са $P_0(x) = \sum_{j=0}^{n/2-1} a_{2j}x^j$, односно $P_1(x) = \sum_{j=0}^{n/2-1} a_{2j+1}x^j$ означимо полиноме степена $n/2-1$ са коефицијентима полинома P парног, односно непарног индекса, долазимо до једнакости

$$P(x) = P_0(x^2) + xP_1(x^2). \tag{8.6}$$

Заменом x са $-x$ добијамо $P(-x) = P_0(x^2) + (-x)P_1(x^2)$. Израчунавање $P(x_j)$, $j = 0, 1, \dots, n-1$ своди се на рачунање $P(x_j)$ и $P(-x_j)$ за $j = 0, 1, \dots, n/2-1$, односно на израчунавање само $n/2$ вредности $P_0(x_j^2)$, $n/2$ вредности $P_1(x_j^2)$, и допунских $n/2$ сабирања, $n/2$ одузимања и n множења. Дакле, имамо два потпроблема величине $n/2$ и $O(n)$ допунских операција.

Може ли се наставити рекурзивно на исти начин? Ако би нам то пошло за руком, дошли бисмо до познате диференцне једначине $T(n) = 2T(n/2) + O(n)$, чије је решење $T(n) = O(n \log n)$. Проблем израчунавања $P(x)$ (полинома степена $n-1$) у n тачака свели смо на израчунавање $P_0(x^2)$ и $P_1(x^2)$ (два полинома степена $n/2-1$) у $n/2$ тачака. То је регуларна редукција, изузев једног детаља: вредности x у $P(x)$ могу се произвољно бирати, али вредности x^2 у изразу (на пример) $P_0(x^2)$ могу бити само позитивне. Пошто смо до редукције дошли коришћењем негативних бројева, ово представља проблем. Издвојимо из (8.5) матрицу која одговара израчунавању вредности $P_0(x^2)$:

$$\begin{pmatrix} 1 & x_0^2 & x_0^4 & \dots & x_0^{n-2} \\ 1 & x_1^2 & x_1^4 & \dots & x_1^{n-2} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & x_{n/2-1}^2 & x_{n/2-1}^4 & \dots & x_{n/2-1}^{n-2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_2 \\ \dots \\ a_{n-2} \end{pmatrix} = \begin{pmatrix} P_0(x_0^2) \\ P_0(x_1^2) \\ \dots \\ P_0(x_{n/2-1}^2) \end{pmatrix}.$$

Да бисмо још једном извели редукцију на исти начин, морали бисмо да ставимо нпр. $x_{n/4}^2 = -(x_0)^2$. Пошто су квадрати реалних бројева увек позитивни, ово је немогуће, бар ако се ограничимо на реалне бројеве. Потешкоћа се превазилази преласком на комплексне бројеве. Проблем се може опет поделити на два дела стављајући $x_{j+n/4} = ix_j$, за $j = 0, 1, \dots, n/4-1$ (i је овде корен из -1, комплексан број). Ово раздвајање задовољава исте услове као и претходно. Према томе, проблем величине $n/2$ може се решити свођењем на два проблема величине $n/4$, изводећи $O(n)$ допунских операција.

За следеће раздвајање потребан нам је број z такав да је $z^8 = 1$ и $z^j \neq 1$ за $0 < j < 8$, односно примитивни осми корен из јединице; тада је $z^4 = -1$ и $z^2 = i$. Општије, потребан нам је примитивни n-ти корен из јединице. Означимо га са ω (због једноставности се n не спомиње експлицитно; у оквиру овог одељка ради се увек о једном истом n). Број ω задовољава следеће услове:

$$\omega^n = 1, \ \omega^j \neq 1 \text{ за } 0 < j < n. \tag{8.7}$$

За n тачака $x_0, x_1, \ldots, x_{n-1}$ бирамо бројеве $1, \omega, \omega^2, \ldots, \omega^{n-1}$. Према томе, израчунава се следећи производ:

$$\begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\ 1 & \omega^2 & \omega^{2\cdot 2} & \ldots & \omega^{2\cdot(n-1)} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & \omega^{n-1} & \omega^{(n-1)\cdot 2} & \ldots & \omega^{(n-1)\cdot(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \ldots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} P(1) \\ P(\omega) \\ \ldots \\ P(\omega^{n-1}) \end{pmatrix}. \tag{8.8}$$

V(w)

V(w) V(1/w) = n I n

r,s:

1 w^r w^(2r) ... w^((n-1)r)

1 w^(-s) w^(-2s)...w^(-(n-1)s)

1 w^(r-s) w^(2(r-s))... w((n-1)(r-s))

r=s: n
r!=s: w^(n(r-s))-1 / w^(r-s)-1 =0

v(w) (1/n V(1/w)) = I

Овај производ се зове **Фуријеова трансформација** вектора $(a_0, a_1, \ldots, a_{n-1})$. Запазимо најпре да је испуњен услов

$$x_{n/2+j} = \omega^{n/2+j} = \omega^{n/2}\omega^j = -x_j, \quad j = 0, 1, \ldots, n/2 - 1.$$

Према томе, прва редукција проблема величине n на два мања је и даље исправна. Даље, два потпроблема произашла из ове редукције имају по $n/2$ тачака $1, \omega^2, \omega^4, \ldots, \omega^{n-2}$, што је управо проблем величине $n/2$, у коме уместо ω фигурише ω^2 — примитивни $n/2$-ти корен из јединице. Да је ω^2 примитивни $n/2$-ти корен из јединице непосредно следи из услова (8.7). Према томе, даље се може наставити рекурзивно. Сложеност алгоритма задовољава диференцну једначину $T(n) = 2T(n/2) + O(n)$, чије је решење $O(n \log n)$. Алгоритам омогућује ефикасно израчунавање Фуријеове трансформације вектора коефицијената полинома, па је добио име брза Фуријеова трансформација, односно FFT. Алгоритам је приказан на слици 4.

Алгоритам FFT$(n, a_0, a_1, \ldots, a_{n-1}, \omega,$ **var** $V)$;
Улаз: n (природни број), $a_0, a_1, \ldots, a_{n-1}$ (низ *елемената* типа који зависи од примене) и ω (примитивни n-ти корен из јединице).
Излаз: V (низ излазних елемената, са индексима од 0 до $n-1$).

```
    {претпоставља се да је n степен двојке}
begin
    if n = 1 then
        V[0] := a0;
    else
        FFT(n/2, a0, a2, ..., a(n-2), ω^2, U);
        FFT(n/2, a1, a3, ..., a(n-1), ω^2, W);
        for j := 0 to n/2 - 1 do   {према (8.6) за x = ω^j}
            V[j] := U[j] + ω^j W[j];
            V[j + n/2] := U[j] - ω^j W[j];
end
```

Рис. 4. Алгоритам брзе Фуријеове трансформације, FFT.

Пример 8.2. Демонстрираћемо израчунавање FFT на примеру полинома са коефицијентима $(0, 1, 2, 3, 4, 5, 6, 7)$. Да бисмо избегли забуну, потпроблеме ћемо означавати са $P_{j_0,j_1,\ldots,j_k}(x_0, x_1, \ldots, x_k)$, где $j_0, j_1, \ldots, j_k$ означавају коефицијенте полинома, а $x_0, x_1, \ldots, x_k$ тачке у којима се израчунавају вредности полинома. Задатак је дакле решити проблем $P_{0,1,2,3,4,5,6,7}(1, \omega, \omega^2, \ldots, \omega^7)$, и то на основу (8.6).

Први корак је свођење $P_{0,1,2,3,4,5,6,7}(1, \omega, \omega^2, \ldots, \omega^7)$ на $P_{0,2,4,6}(1, \omega^2, \omega^4, \omega^6)$ и $P_{1,3,5,7}(1, \omega^2, \omega^4, \omega^6)$. Настављамо рекурзивно и сводимо $P_{0,2,4,6}(1, \omega^2, \omega^4, \omega^6)$ на $P_{0,4}(1, \omega^4)$ и $P_{2,6}(1, \omega^4)$. $P_{0,4}(1, \omega^4)$ се затим своди на $P_0(1) = 0$, и $P_4(1) = 4$. Комбиновањем ових резултата добијамо

$$
\begin{aligned}
P_{0,4}(1) &= P_0(1) + 1 \cdot P_4(1) = 0 + 1 \cdot 4 = 4, \\
P_{0,4}(\omega^4) &= P_0(\omega^8) + \omega^4 P_4(\omega^8) = 0 + \omega^4 \cdot 4.
\end{aligned}
$$

Пошто је $\omega^4 = -1$, добијамо $P_{0,4}(\omega^4) = -4$, односно после обједињавања $P_{0,4}(1, \omega^4) = (4, -4)$. На исти начин добијамо $P_{2,6}(1, \omega^4) = (8, -4)$.

Сада комбиновањем ова два вектора добијамо $P_{0,2,4,6}(1, \omega^2, \omega^4, \omega^6)$:

$$
\begin{aligned}
P_{0,2,4,6}(1) &= P_{0,4}(1) + 1 \cdot P_{2,6}(1) = 4 + 8 = 12, \\
P_{0,2,4,6}(\omega^2) &= P_{0,4}(\omega^4) + \omega^2 P_{2,6}(\omega^4) = -4 + \omega^2(-4), \\
P_{0,2,4,6}(\omega^4) &= P_{0,4}(\omega^8) + \omega^4 P_{2,6}(\omega^8) = P_{0,4}(1) - 1 P_{2,6}(1) = 4 - 8 = -4, \\
P_{0,2,4,6}(\omega^6) &= P_{0,4}(\omega^{12}) + \omega^6 P_{2,6}(\omega^{12}) = P_{0,4}(\omega^4) - \omega^2 P_{2,6}(\omega^4) = -4 - \omega^2(-4),
\end{aligned}
$$

односно

$$P_{0,2,4,6}(1, \omega^2, \omega^4, \omega^6) = (12, -4(1 + \omega^2), -4, -4(1 - \omega^2)).$$

На сличан начин добија се

$$P_{1,3,5,7}(1, \omega^2, \omega^4, \omega^6) = (16, -4(1 + \omega^2), -4, -4(1 - \omega^2)).$$

Преостаје још израчунавање 8 вредности $P_{0,1,2,3,4,5,6,7}(1, \omega, \omega^2, \ldots, \omega^7)$. На пример, $P_{0,1,2,3,4,5,6,7}(1) = 12 + 1 \cdot 16 = 28$, и слично $P_{0,1,2,3,4,5,6,7}(\omega^4) = 12 - 1 \cdot 16 = -4$; $P_{0,1,2,3,4,5,6,7}(\omega) = (-4(1+\omega^2)) + \omega \cdot (-4(1+\omega^2))$, и слично $P_{0,1,2,3,4,5,6,7}(\omega^5) = (-4(1 + \omega^2)) - \omega \cdot (-4(1 + \omega^2))$, итд. Резултат је

$$
\begin{aligned}
&P_{0,1,2,3,4,5,6,7}(1, \omega, \omega^2, \ldots, \omega^7) = \\
&4(7, -1 - \omega - \omega^2 - \omega^3, -1 - \omega^2, -1 + \omega^2 - \omega^3 + \omega^5, \\
&-1, -1 + \omega - \omega^2 + \omega^3, -1 + \omega^2, -1 + \omega^2 + \omega^3 - \omega^5).
\end{aligned}
$$

8.6.2. Инверзна Фуријеова трансформација. Брза Фуријеова трансформација решава само пола проблема: вредности задатих полинома $p(x)$ и $q(x)$ могу се ефикасно израчунати у тачкама $1, \omega, \ldots, \omega^{n-1}$, измножити парови добијених вредности, и тако наћи вредности полинома $p(x)q(x)$ у наведеним тачкама. Остаје проблем интерполације, односно одређивања коефицијената производа полинома на основу вредности у тачкама. На срећу, испоставља се да је проблем интерполације врло сличан проблему израчунавања вредности, и да га решава практично исти алгоритам.

Вратимо се матричној нотацији. Нека A^T транспонована матрица матрице A. Означимо вектор коефицијената полинома са $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})^T$, а вектор вредности полинома са $\mathbf{v} = (P(1), P(\omega), \ldots, P(\omega^{n-1}))^T$. Нека је $V(\omega)$ матрица из једнакости (8.8). Ако су задати су коефицијенти полинома $\mathbf{a}$, његове вредности $\mathbf{v}$ у n тачака $1, \omega, \ldots, \omega^{n-1}$ добијају се према (8.8) израчунавањем производа $\mathbf{v} = V(\omega)\mathbf{a}$. С друге стране, ако су задате вредности полинома $\mathbf{v} = (P(1), P(\omega), \ldots, P(\omega^{n-1}))^T = (v_0, v_1, \ldots, v_{n-1})^T$, а потребно је израчунати његове коефицијенте, једнакост (8.8), односно $V(\omega)\mathbf{a} = \mathbf{v}$ је систем линеарних једначина по $\mathbf{a}$. Решавање система једначина има у општем случају доста велику временску сложеност $O(n^3)$, али се овде ради о једном специјалном систему једначина. Лако се проверава да је

$$V(\omega)V(\omega^{-1}) = nI,$$

где је са I означена јединична матрица реда n. Заиста, ако је $r \neq s$, онда је производ $(r+1)$-е врсте матрице $V(\omega)$ и $(s+1)$-е колоне матрице $V(\omega^{-1})$ једнак

$$\sum_{k=0}^{n-1} \omega^{rk}\omega^{-sk} = \sum_{k=0}^{n-1} \omega^{k(r-s)} = \frac{1-\omega^{n(r-s)}}{1-\omega^{r-s}} = 0.$$

Ако је пак $r = s$, онда је тај производ $\sum_{k=0}^{n-1} \omega^{rk}\omega^{-rk} = n$. Тиме је доказана следећа теорема.

Теорема 8.1. *Инверзна матрица матрице $V(\omega)$ Фуријеове трансформације је*

$$V(\omega)^{-1} = \frac{1}{n}V(\omega^{-1}).$$

Решавање система једначина $\mathbf{v} = V(\omega)\mathbf{a}$ своди се дакле на израчунавање производа

$$\mathbf{a} = \frac{1}{n}V(\omega^{-1})\mathbf{v}.$$

Посао се даље поједностављује захваљујући следећој теореми.

Теорема 8.2. *Ако је ω онда је ω^{-1} такође примитивни n-ти корен из јединице.*

Према томе, производ $\frac{1}{n}V(\omega^{-1})\mathbf{v}$ може се израчунати применом брзе Фуријеове трансформације, замењујући ω са ω^{-1}. Ова трансформација зове се **инверзна Фуријеова трансформација**.

Сложеност. Узимајући све у обзир, производ два полинома може се израчунати изводећи $O(n \log n)$ операција (са комплексним бројевима).

8.7. Резиме

Алгоритми приказани у овом поглављу само су мали избор из скупа познатих нумеричких алгоритама. Још једном смо се уверили да директни алгоритми нису увек и најбољи. Штрасенов алгоритам је један од најистакнутијих

примера потпуно неинтуитивног алгоритма за решавање на први поглед једноставног проблема. Видели смо још неколико примера коришћења индукције, и специјално, алгоритама заснованих на декомпозицији. Друга техника, уобичајена нарочито код проблема са матрицама, је свођење једног проблема на други. Тај метод биће илустрован новим примерима у поглављу 10.

Задаци

8.1. Окарактерисати везу између алгоритма за израчунавање n^k поновљеним квадрирањем (слика 2) и бинарне представе броја k.

8.2. Алгоритам *Stepen_kvadriranjem* (слика 2) за израчунавање n^k не минимизира увек број множења. Навести пример израчунавања n^k ($k > 10$) са мањим бројем множења него у овом алгоритму.

8.3. Претпостављамо да се позитивни рационални бројеви x представљају паровима природних бројева (a, b) таквим да је $x = a/b$. Конструисати алгоритам који за дати број (a, b) одређује минималну репрезентацију (a', b') броја $x = a/b$.

8.4. Доказати да директни алгоритам за множење полинома заснован на декомпозицији, који израчунава сва четири производа мањих полинома, извршава идентичне операције као и обичан алгоритам множења (8.2). Претпоставити да је n степен двојке.

8.5. Метод декомпозиције може се искористити за ефикасније множење бинарних бројева. Конструисати одговарајући алгоритам и размотрити разлике између њега и алгоритма за множење полинома. Уопштити алгоритам на множење бројева у систему са основом $b > 1$.

8.6. Како се могу помножити два комплексна броја $(a + bi)(c + di)$ помоћу само три реална множења?

8.7. Претпоставимо да се израчунавање производа две квадратне матрице реда четири може свести на k производа елемената. Колика би била асимптотска сложеност општег алгоритма за израчунавање производа две квадратне матрице реда n заснованог на овом свођењу? За коју највећу вредност k би овакав алгоритам био још увек бржи од Штрасеновог алгоритма?

8.8. Посматрајмо два алгоритма за израчунавање степена n^k из одељка 8.2, поновљено множење и поновљено квадрирање. Нека је n природан број са d цифара. Претпоставимо да се множење целих бројева врши обичним алгоритмом, који се састоји од $d_1 d_2$ корака ако се множе бројеви са d_1 и d_2 цифара. Колико траје степеновање n^k помоћу споменута два алгоритма? (Може се претпоставити да је k степен двојке, и да је производ два броја са d_1 и d_2 цифара број са $d_1 + d_2$ цифара.)

8.9. Конструисати алгоритам за одређивање НЗД k задатих природних бројева.

8.10. Конструисати алгоритам за одређивање најмањег заједничког садржаоца (НЗС) датих природних бројева a и b (односно најмањег природног броја дељивог и са a и са b).

8.11. Конструисати алгоритам за одређивање највећег заједничког садржаоца (НЗС) датих k природних бројева.

8.12. Фибоначијеви бројеви су дефинисани следећом диференцном једначином:

$$F(n) = F(n-1) + F(n-2),\ (n > 2), \quad F(1) = F(2) = 1.$$

(а) Доказати да се сваки природан број $n > 2$ може представити у облику збира највише различитих $\log_2 n$ Фибоначијевих бројева. (б) Конструисати алгоритам за одређивање такве представе датог броја n.

8.13. Каже се да полином $b = b(x)$ дели полином a, ако постоји полином q такав да је $a = bq$. Полином d је највећи заједнички делилац (НЗД) полинома a и b ако је то полином највећег степена, са најстаријим коефицијентом 1, који дели a и b. (а) Доказати да је НЗД два полинома дефинисан једнозначно. (б) Показати да се помоћу Еуклидовог алгоритма може одредити НЗД два дата полинома.

8.14. **Хамилтонови кватерниони** су суме облика $a + bi + cj + dk$, где су a, b, c и d реални бројеви, а i, j и k су специјални симболи. Кватерниони се сабирају и множе по обичним правилима, при чему се користе једнакости: $i^2 = j^2 = k^2 = -1$, $ij = -ji = k$, $jk = -kj = i$, $ki = -ik = j$ (симболи i, j и k комутирају са реалним бројевима). Колико је потребно извршити множења реалних бројева да би се израчунао производ два кватерниона на обичан начин? Показати како се може број реалних множења смањити на 10.

8.15. Показати како се квадрат квадратне матрице реда два може израчунати помоћу пет множења.

8.16. **Пермутациона матрица** је квадратна матрица реда n у чијој се свакој врсти и колони налази тачно један елеменат различит од нуле, једнак јединици. Пермутациона матрица може се представити вектором P, тако да буде $P[i] = j$, ако i-та врста садржи јединицу у j-тој колони, $i = 1, 2, \ldots, n$. (а) Доказати да је производ две пермутационе матрице такође пермутациона матрица. (б) Конструисати алгоритам сложености $O(n)$ за множење две пермутационе матрице представљене векторима.

8.17. Нека је $B = \{0, 1\}$. Свака Булова функција f од n променљивих може се представити полиномом $f = \sum_{\sigma \in B^n} a_\sigma x^\sigma$, при чему је сабирање екслузивна дисјункција (сабирање по модулу два), множење је конјункција, $x^\sigma = \prod_{i=1}^n x_i^{\sigma_i}$, а $x_i^{\sigma_i}$ означава x_i ако је $\sigma_i = 0$, односно негацију $\bar{x}_i$ ако је $\sigma_i = 1$. Тако на пример, $x_1 \vee x_2 = 1 \cdot x_1 x_2 + 1 \cdot x_1 \bar{x}_2 + 1 \cdot \bar{x}_1 x_2 + 0 \cdot \bar{x}_1 \bar{x}_2$, тј. за ову функцију је $a_{00} = a_{01} = a_{10} = 1$, и $a_{11} = 0$. Конструисати алгоритам временске сложености $O(n2^n)$ за израчунавање коефицијената a_σ, $\sigma \in B^n$, Булове функције f од n променљивих, задате таблицом својих вредности $f(\sigma)$, $\sigma \in B^n$.

8.18. За две Булове функције f и g од n променљивих растојање $d(f, g)$ може се дефинисати као број аргумената $x = (x_1 x_2 \ldots x_n) \in \{0, 1\}^n$ на којима им се вредности разликују. Функција f је линеарна Булова функција од n променљивих ако се може изразити у облику $f(x) = a_0 + \sum_{i=1}^n a_i x_i$, где је сабирање екслузивна дисјункција (сабирање по модулу два), а множење је конјункција. Конструисати алгоритам временске сложености $O(n2^n)$ који за дату Булову функцију f одређује бар једну њој најближу линеарну функцију.

8.19. Нека су елементи квадратних матрица A и B реда n реализације независних случајних променљивих које са једнаким вероватноћама узимају вредности из скупа $\{0, 1\}$. Ако $\wedge$, $\vee$ означавају конјункцију, односно дисјункцију, показати да се "производ" $C = AB$ са елементима $c_{ij} = \vee_{k=1}^n (a_{ik} \wedge b_{kj})$, $1 \le i, j \le n$, може израчунати алгоритмом очекиване (средње) временске сложености $O(n^2)$.

8.20. Нека су A и B квадратне матрице реда n чије су врсте отворени Грејови кодови (елементи су им из скупа $\{0, 1\}$, а сваке две узастопне врсте разликују се на тачно једном месту). Конструисати алгоритам сложености $O(n^2)$ за множење оваквих матрица.

8.21. Дато је n реалних матрица $M_1, M_2, \ldots, M_n$. Матрица M_i је димензија $a_i \times a_{i+1}$, $1 \le i \le n$. Претпоставимо да се производ матрица димензија $a_i \times a_{i+1}$ и $a_{i+1} \times a_{i+2}$ израчунава обичним алгоритмом сложености $O(a_i a_{i+1} a_{i+2})$. У производу $M_1 M_2 \ldots M_n$ треба распоредити заграде (и тиме изабрати редослед множења) тако да буде минималан укупан број операција. Конструисати алгоритам за налажење оптималног редоследа израчунавања производа.

Глава 9

Примене у криптографији

9.1. Увод

Криптографија је област у којој алгоритми имају значајну примену. Значај криптографије може се оценити поред осталог и на основу чињенице да су први рачунари направљени у току другог светског рата, и коришћени за разбијање немачких шифри, што је вероватно пресудно утицало на исход рата. Сврха овог поглавља је да читалац добије увид у неке основне појмове и проблеме ове области, данас изузетно важне. За дубље упознавање са криптографијом добро може да послужи књига [**6**]

Увешћемо најпре неопходне појмове. Претпоставимо да особа A (**пошиљалац**) жели да пошаље **поруку** особи B (**примаоцу**). Пошиљалац жели да буде сигуран да нико сем примаоца не може да прочита ту поруку.

За поруку кажемо да је **отворени текст**. Процес маскирања поруке, које за сврху има сакривање њеног садржаја зове се **шифровање**. Резултат шифровања отвореног текста је **шифрат**. Дешифровањем шифрата добија се оригинална порука, видети слику 1.

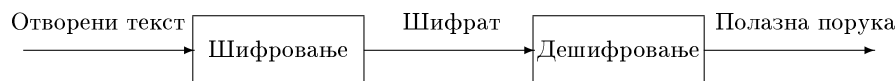


Рис. 1. Шифровање и дешифровање.

Вештина или наука која на овај начин обезбеђује сигурност порука зове се **криптографија**. **Криптоанализа** је вештина или наука разбијања шифрата, односно читања шифрованих порука. Област математике која обухвата и криптографију и криптоанализу зове се **криптологија**. У данашње време скоро сви криптолози су математичари — они то морају да буду.

Отворени текст означаваћемо са P. То може да буде низ бита, текстуални фајл, дигитализовани говорни сигнал или нпр. дигитализована слика. Кад се ради о рачунару, P су просто бинарни подаци. Отворени текст се или шаље или смешта у меморију. У оба случаја порука P се претходно шифрује.

Шифрат ћемо означавати са C. То су такође бинарни подаци, понекад исте величине као и P, понекад дужи (или краћи, што се постиже комбиновањем компресије и шифровања). Функција шифровања E делује на P и тако се добија C: $C = E(P)$. Супротан процес је дејство функције дешифровања D на C

да би се добило P, односно $P = D(C)$. Шифровање и дешифровање су дакле функције које задовољавају услов $D(E(P)) = P$.

Криптографски алгоритам или **шифра** је математичка функција која се користи за шифровање, односно дешифровање. Отворени текст се шифрује применом **алгоритма за шифровање**, а шифрат се дешифрује **алгоритмом за дешифровање**. Да би се обезбедила сигурност порука, сви савремени алгоритми за шифровање користе **кључ** k — параметар који утиче на шифровање, односно дешифровање: $C = E_k(P)$, односно $P = D_k(C)$ (слика 2). Разуме се да за сваки кључ k важи $D_k(E_k(P)) = P$. Кључ узима једну од више могућих вредности из **простора кључева**. Из разумљивих разлога пожељно је да простор кључева има што већи број елемената. Постоје алгоритми са посебним кључем за шифровање k_1, односно за дешифровање k_2, слика 3. У том случају је $C = E_{k_1}(P)$, $P = D_{k_2}(C)$, и $D_{k_2}(E_{k_1}(P)) = P$.

Рис. 2. Шифровање и дешифровање једним кључем.

Рис. 3. Шифровање и дешифровање са два кључа.

Алгоритми са кључем могу се поделити на симетричне и асиметричне. **Симетрични** су алгоритми код којих се кључ за шифровање може израчунати полазећи од кључа за дешифровање, и обрнуто. У многим симетричним системима су кључеви за шифровање и дешифровање идентични. За употребу таквих алгоритама, који се такође зову и **алгоритми са тајним кључем**, неопходно је да се пошиљалац и прималац договоре који ће кључ користити пре него што почну да размењују шифроване поруке. Сигурност симетричног алгоритма заснива се на кључу; неко ко сазна кључ (а зна и алгоритам шифровања/дешифровања) може да шифрује и дешифрује поруке у том систему. Шифровање, односно дешифровање симетричним алгоритмом могу се означити са $E_k(P) = C$, односно $D_k(C) = P$.

Симетрични алгоритми могу се условно поделити у две категорије. Неки од њих обрађују отворени текст бит по бит; они се зову **ланчане шифре**. Други раде на отвореном тексту подељеном на групе бита. Те групе бита зову се **блокови**, а алгоритми су **блоковске шифре**. Типична величина блока је

64 бита — довољно велика да отежа анализу, а довољно мала да буде практична. Код оба типа алгоритама за шифровање и дешифровање користи се исти кључ. Пре појаве рачунара алгоритми су шифровали отворени текст знак по знак. Такав алгоритам може се схватити као ланчани (који трансформише низ знакова) или блоковски, са блоком величине 8 бита.

Асиметрични, односно алгоритми са јавним кључем су другачији. Замишљени су тако да кључ за шифровање буде различит од кључа за дешифровање. Поред тога, кључ за дешифровање не може се (у било ком разумном временском интервалу) израчунати полазећи од кључа за шифровање. Назив системи са јавним кључем потиче од чињенице да се кључ за шифровање може доставити свима: особа, коју прималац никад није срео, може да шифрује поруку кључем за шифровање, али само онај ко има одговарајући кључ за дешифровање може да дешифрује ту поруку. У таквим системима кључ за шифровање зове се **јавни кључ**, а кључ за дешифровање — **тајни кључ**. Шифровање јавним кључем означава се са $C = E_k(P)$. Иако су јавни и тајни кључ различити, дешифровање одговарајућим тајним кључем означава се са $C = D_k(P)$.

Понекад се поруке шифрују тајним, а дешифрују јавним кључем; ово се користи за такозвани дигитални потпис, (замену за обичан потпис) који се може прикључити шифрату поруке. Ако се са $CRC(P)$ означи "контролна сума" поруке (блок фиксиране дужине од нпр. 32, 64 или више бита за контролу исправности поруке), а са E_a, D_a, односно E_b, D_b, шифровање и дешифровање (јавни и тајни кључ) учесника A, односно B, онда се као потпис поруке $E_b(P)$ коју A шаље B може искористити шифрат $D_a(CRC(P))$. Прималац B дешифрује $E_b(P)$ својим тајним кључем D_b, израчунава контролну суму $CRC(P)$ и веродостојност потписа проверава тако што израчунату контролну суму упореди са резултатом дешифровања потписа $D_a(CRC(P))$ (јавним кључем E_a). Ако се та два резултата слажу, онда је прималац уверен да је поруку могао добити само од некога ко зна тајни кључ D_a, дакле од учесника A.

Основни циљ криптографије је очување тајности поруке, односно одбрана тајности њеног садржаја од некога ко "прислушкује" канал везе (у даљем тексту **нападач**. Криптоанализа је наука или вештина одређивања отвореног текста поруке без познавања кључа. Криптоанализа такође може да пронађе недостатке шифарског система који би довели до таквог резултата.

Покушај криптоанализе зове се **напад**. Успешан напад (**декриптирање**) подразумева да нападач детаљно познаје алгоритам за шифровање. У стварном животу то није увек случај, али је то уобичајена претпоставка при криптоанализи. Та претпоставка је разумна: ако сигурност система зависи од тајности алгоритма, онда то није велика сигурност.

Постоји неколико типова криптоаналитичких напада, према количини информација које стоје на располагању нападачу. У свим варијантама претпоставља се да нападач потпуно познаје алгоритам за шифровање.

- **Напад на основу шифрата**. Криптоаналитичар има шифрате више порука, добијене истим алгоритмом за шифровање. Посао криптоаналитичара је да реконструише текстове што више порука, или још

боље, да реконструише кључ коришћен за шифровање порука, да би се могле дешифровати друге поруке шифроване истим кључем. Дакле, дато је $C_1 = E_k(P_1)$, $C_2 = E_k(P_2)$, ..., $C_n = E_k(P_n)$, а треба одредити $P_1, P_2, \ldots, P_n$; k; или алгоритам за одређивање P_{n+1} на основу $C_{n+1} = E_k(P_{n+1})$.

- **Напад на основу парова (отворени текст, шифрат)**. Криптоаналитичар зна не само шифрате више порука, него и одговарајуће отворене текстове. Задатак је реконструисати кључ коришћен за шифрован-је порука, или пронаћи поступак за отварање наредних порука шифрованих истим кључем. Дакле, дато је $P_1, C_1 = E_k(P_1)$, $P_2, C_2 = E_k(P_2)$, ..., $P_n, C_n = E_k(P_n)$, а треба одредити k или алгоритам за одређивање P_{n+1} на основу $C_{n+1} = E_k(P_{n+1})$.
- **Напад на основу изабраног отвореног текста**. Криптоаналитичар зна парове (отворени текст, шифрат) за више порука, при чему он са̂ м може да бира отворене текстове. Ово је моћнији напад од претходног, јер криптоаналитичар може да изабере такве отворене текстове са шифратом, који дају много више информације о кључу. Задатак је реконструисати кључ коришћен за шифровање порука, или пронаћи поступак за отварање нових порука шифрованих истим кључем. Дакле, дато је $P_1, C_1 = E_k(P_1)$, $P_2, C_2 = E_k(P_2)$, ..., $P_n, C_n = E_k(P_n)$, при чему криптоаналитичар бира P_1, P_2, ..., P_n, а треба одредити k или алгоритам за одређивање P_{n+1} на основу $C_{n+1} = E_k(P_{n+1})$.

Други и трећи тип напада су чешћи него што то изгледа на први поглед: може се украсти порука која је већ шифрована, или се може потплатити неко да шифрује изабрану поруку. Чак ни подмићивање не мора да буде неопходно: ако предате провокативну поруку амбасадору стране земље, можете са сигурношћу очекивати да ће је он шифровану послати на разматрање у своју земљу.

Различити шифарски системи обезбеђују различите нивое сигурности, у зависности од тога колико их је тешко разбити. Теоријски је могуће разбити (декриптирати) сваки алгоритам сем једног (случајна шифра, одељак 9.3), под претпоставком да је на располагању довољно времена и рачунарских ресурса. Ако је цена декриптирања алгоритма већа од вредности шифрованих података, онда је шифровање вероватно безбедно. Рачунари постају све бржи и јефтинији, а просечна вредност података постаје све већа. Важно је обезбедити да се те две линије не пресеку.

Неки алгоритми се могу разбити само ако се утроши време дуже од старости свемира, и уз помоћ рачунара већег од све материје у свемиру. Такви алгоритми се теоријски могу разбити, али не и практично, и могу се сматрати **сигурним**.

Алгоритам је **безусловно сигуран** ако, без обзира на количину шифрата који нападач има на располагању, та количина није довољна за реконструкцију отвореног текста. Једино се случајна шифра, (одељак 9.3) не може разбити, чак ни ако су на располагању неограничени ресурси. Криптографија се више

бави системима за које је разбијање *практично* неизводљиво са расположивим (садашњим или будућим) ресурсима; то су такозвани **практично сигурни** шифарски системи. При томе се наравно "практично расположиви ресурси" не могу прецизно дефинисати.

Мера сигурности алгоритма за шифровање је број операција које треба извршити приликом његовог разбијања. Ако је, на пример, за неки алгоритам тај број 2^{128}, и на располагању је милион процесора који извршавају по билион таквих операција у секунди, онда је за разбијање алгоритма потребно више од 10^{16} година (треба имати на уму да се старост свемира процењује на 10^{10} година). Разумно је такав алгоритам сматрати практично сигурним.

За разлику од броја операција за разбијање алгоритма, који је константан, брзина рачунара сигурно није константна. У току последње половине века моћ рачунара је фантастично повећана, и нема разлога претпоставити да се овај тренд неће наставити. Многи криптоаналитички напади су савршено прилагођени паралелним рачунарима. Посао се може разбити на милионе делова, тако да их поједини процесори обрађују потпуно независно (тј. није потребна никаква комуникација између паралелних процесора). Рећи да је неки алгоритам сигуран зато што се не може разбити садашњом технологијом, најчешће је неозбиљно. Добри системи направљени су тако да буду сигурни и против напада рачунарима какви ће бити прављени у далекој будућности.

9.2. Класична криптографија

Пре појаве рачунара криптографија се бавила системима заснованим на шифровању слова (знакова). Криптографски алгоритми замењивали су слова једно другим или су их међусобно премештали. Бољи системи радили су обе ове ствари, и то више пута.

Данашњи алгоритми су компликованији, али је филозофија остала практично непромењена. Основна промена је у томе да алгоритми шифрују бите уместо слова. То је, међутим, само промена величине алфабета, са нпр. 26 на два. Већина добрих алгоритама за шифровање и даље комбинује елементе замене (супституције) и транспозиције (премештања).

Шифре замене су оне код којих се отворени текст маскира тако што се слова отвореног текста замењују другим у шифрату. Прималац примењује инверзне замене и тако добија полазни отворени текст. У класичној криптографији постоји неколико основних типова шифри замене.

- **Шифра просте замене** је она код које се слова отвореног текста замењују одговарајућим словом (увек на исти начин) у шифрату. Криптограми у енигматским часописима спадају у ову категорију
- **Хомофонска шифра** слична је шифри просте замене, изузев што се знак отвореног текста на неколико начина може заменити знаковима шифрата. На пример, "А" се може заменити са 5, 13, 25 или 56, "Б" се може заменити са 7, 19, 31 или 42, итд.

- **Полиалфабетска шифра замене** формира се од више шифара просте замене. Може се, на пример, користити пет различитих шифара просте замене; која од њих ће бити коришћена зависи од редног броја слова у отвореном тексту.

Чувена Цезарова шифра, код које се свако слово замењује словом које је три слова удесно до њега (по модулу 26) (A се замењује са D, B са E, ..., W са Z, ..., X са A, Y са B и Z са C) је шифра просте замене. Ако се у имену HAL рачунара из филма "Одисеја у свемиру 2001" слова замене следећим у абецеди, добија се IBM, што вероватно није случајно. Шифре просте замене се могу лако разбити, јер не мењају скуп учестаности слова у отвореном тексту.

Хомофонске шифре су биле корищене већ око 1401. године. Много их је теже разбити од шифре просте замене, али оне ипак не маскирају потпуно статистичке особине отвореног текста. Напад са познатим отвореним текстом је тривијалан. Напад са познавањем шифрата је тежи, али се помоћу рачунара извршава за неколико секунди.

Полиалфабетске шифре открио је Леон Батиста 1568. године. Користила их је армија Уније у америчком грађанском рату. Иако се доста лако разбијају (нарочито помоћу рачунара), многи комерцијални програми за шифровање користе овакве шифре (на пример, шифровање у оквиру текст процесора WordPerfect).

Полиалфабетске шифре замене имају више једнословних кључева, који се користе за шифровање по једног слова отвореног текста. Први кључ шифрује прво слово отвореног текста, други кључ шифрује друго слово отвореног текста, итд. Пошто се искористе сви кључеви, почиње се поново испочетка. Ако има укупно 20 једнословних кључева, онда ће свако двадесето слово отвореног текста бити шифровано истим кључем. Ово је **период** шифре. У класичној криптографији шифре са дугим периодом било је знатно теже разбити од оних са кратким периодом. Постоје технике које (помоћу рачунара) лако разбијају овакве шифре са врло дугачким периодом.

Шифра са текстуалним кључем је она код које се један текст (у шпијунским филмовима је то често библија) користи за шифровање другог текста; то је такође пример полиалфабетске шифре. Иако ова шифра има период дужине једнаке дужини отвореног текста, она се лако може разбити.

Шифра транспозиције је она код које слова отвореног текста остају непромењена, али им се мења редослед. На пример, може се отворени текст преписати у врсте једнаке дужине, па затим шифрат прочитати по колонама.

Око 1920. године појавиле су се прве машине намењене аутоматизацији процеса шифровања. Заснивале су се на концепту **ротора**, механичког диска намењеног за извођење произвољне замене. Најпознатија роторска машина је Енигма, коју су Немци знатно побољшану, користили у другом светском рату. Ма колико да је била компликована, Енигма је разбијена, што је битно утицало на исход рата.

Размотрићемо сада неколико примера савремених алгоритама за шифровање: случајну шифру, DES и RSA.

9.3. Случајна шифра

Можда звучи необично, али постоји апсолутно тајни шифарски систем. То је такозвана шифра са **једнократним кључем** , или **случајна шифра**, коју су 1917. године патентирали Моборн и Вернам (J. Mauborgne, G. Vernam). У свом класичном облику, ова шифра је просто велики скуп случајних, независних знакова кључа, исписаних на листове папира и спојених заједно у свеску. Пошиљалац користи сваки знак кључа за шифровање тачно једног знака отвореног текста. Прималац има идентичну свеску, и користи слова из ње да дешифрује одговарајуће знакове шифрата.

Сваки кључ се користи само једном, за само једну поруку. Пошиљалац шифрује поруку, а затим уништава искоришћене листове из свеске кључева — то је уобичајена мера предострожности. Прималац поступа исто после дешифровања поруке. Нова порука — нова страна, и нова слова кључа.

Под претпоставком да нападач не може доћи до страна свеске искоришћених за шифровање, овај систем је савршено сигуран. Од задатог шифрата се дешифровањем (уз случајни избор одговарајућег кључа, са униформном расподелом вероватноће) са једнаком вероватноћом може добити сваки отворени текст исте дужине. На пример, претпоставимо да су слова алфабета нумерисана на следећи начин

A	B	C	Č	Ć	D	Đ	E	F	G	H	I	J	K
1	2	3	4	5	6	7	8	9	10	11	12	13	14
L	M	N	O	P	R	S	Š	T	U	V	Z	Ž	
15	16	17	18	19	20	21	22	23	24	25	26	27	

Шифровање знака отвореног текста врши се његовим сабирањем са словом кључа (прецизније, сабирају се њихови редни бројеви; ако се добије збир већи од 27, умањује се за 27; добијени број замењује се знаком према горњој таблици). Претпоставимо да је порука POČNITENAPAD , а да је одговарајући низ кључа из свеске TBFRGFARFMŽČ . Тада је шифрат LRJGŠĆFGGEAG . Пошто је сваки низ кључа једнако вероватан (јер се кључеви генеришу на случајан начин), противник нема информацију којом би дешифровао шифрат. Низ кључа може да буде, на пример, BDIŽAMNAKRBB после чега би резултат дешифровања био SUMNJIVOLICE , или, на пример, BĐUČITENAPAM са резултатом дешифровања SVASTIKINBUT .

Идеја је, да истакнемо још једном, да, пошто је сваки отворени текст једнако вероватан, криптоаналитичар нема начина да установи који отворени текст је послат. Слично, ако се низ случајних бита кључа XOR-ује (сабере по модулу два, бит по бит) са поруком, добија се шифрат потпуно случајаног изгледа, па је криптоаналитичар беспомоћан, без обзира колики су му рачунарски ресурси на располагању.

Проблем са овим системом, и то озбиљан, је да се знакови кључа морају генерисати случајно. Сви напади на овај систем уствари су напади на метод

коришћен за генерисање низа знакова кључа. Ако се у ту сврху користи криптографски слаб алгоритам, то може да изазове невоље. Ако се користи прави извор случајних бројева — што је много теже изводљиво, него што изгледа на први поглед — систем је сигуран.

Коришћење генератора псеудослучајних бројева у општем случају не решава проблем. Постоји много система код којих се случајни бројеви замењују псеудослучајним (генерисаним неким алгоритмом), а који се могу декриптирати. Низови које они генеришу само изгледају случајно, али пажљива анализа открива правилности, које криптоаналитичар може да искористи. Ипак, могуће је генерисати и праве случајне бројеве, чак и помоћу микрорачунара.

Постоји још један проблем са оваквим системом. Дужина низа знакова кључа једнака је дужини поруке. То може да буде практично решење за пренос неколико кратких порука, али тешко може да функционише са каналом за пренос података брзином од милион бита у секунди. Може се на CD-ROM сместити 650 мегабајта низа кључа, и тако практично реализовати случајну шифру за пренос података малом брзином. Тада остаје проблем сигурног чувања CD-ROM -а за време док се не користи, односно његовог уништавања у тренутку кад је његов садржај искоришћен до краја.

Чак и ако се реши проблем преноса кључа до примаоца, мора се обезбедити да пошиљалац и прималац буду потпуно синхронизовани. Ако се шифрат при пријему "помери" за макар један бит, добиће се потпуно бесмислена порука. С друге стране, ако се неки бити погрешно пренесу, само они ће бити дешифровани погрешно (тј. нема тзв. ширења грешака).

Случајна шифра се користи и у данашње време, пре свега за ултрасигурне канале мале брзине. На пример, говорило се да је веза "црвеним телефонима" између бившег СССР и САД (да ли је та веза и данас активна?) била шифрована случајном шифром.

9.4. DES

Алгоритам DES (скраћеница од Data Encryption Standard) коришћен је широм света као стандард више од 20 година. Иако се појављују први симптоми његовог застаревања, он је доста добро издржао вишегодишње покушаје криптоанализе.

До око 1970. године невојна истраживања из криптографије била су ретка, и било је мало објављених радова из ове области. Постојало је више комерцијалних алгоритама и уређаја за шифровање, што је наравно онемогућавало шифровану комуникацију између два различита уређаја. Некадашњи амерички Биро за стандарде покренуо је програм за заштиту рачунарских података, у оквиру кога је било замишљено пројектовање стандардног алгоритма за шифровање, који би штитио податке приликом преноса и смештања. Алгоритам је требало да задовољи прописане критеријуме: да обезбеди високи ниво заштите, да буде комплетно специфициран и разумљив, да његова сигурност буде заснована на тајности кључа (а не алгоритма), да може да буде

доступан свим корисницима, прилагодљив свим применама, да се може ефикасно реализовати у електронским уређајима, да буде ефикасан при коришћењу, да се може верификовати тачност његове реализације и да се може извозити (другим речима, да ипак не буде превише сигуран). Тек на други конкурс јавио се озбиљнији кандидат, фирма IBM са алгоритмом заснованим на сличном алгоритму, званом Луцифер, развијеним око 1970. Предложени алгоритам, иако компликован, састоји се само од једноставних логичких операција са малим групама бита, па се доста ефикасно може реализовати било хардверски, било програмски. Алгоритам је усвојен за стандард, доступан свима, иако је претходно био патентиран. У усвајању коначног решења за овај алгоритам учествовала је и NSA (скраћеница од National Security Agency ; америчка владина установа која се између осталог бави и криптографијом, односно криптоанализом), па неки аутори сумњају да је уствари алгоритам Луцифер ослабљен. Најозбиљнија примедба те врсте односи се на величину кључа (128 бита за Луцифер, 56 бита за DES). Без обзира на све примедбе, алгоритам је усвојен као амерички стандард за шифровање невојних владиних комуникација 1976. године. Касније су уследили други (такође амерички) стандарди за начине коришћења овог алгоритма. У лето 1998. године несигурност DES-а је демонстрирана, тако што је специјализовани рачунар од 250000 долара за 56 сати одредио кључ за DES. Проблем са величином кључа може се решити нпр. троструким шифровањем DES-ом, са три независна кључа. Тренутно је у току конкурс за AES (Advanced Encryption Standard), наследника DES-а.

DES је блоковска шифра, која шифрује блокове од 64 бита и даје блокове од 64 бита шифрата. Дужина кључа је 56 бита. У основи, алгоритам је комбинација две врсте трансформација, "конфузије" и "дифузије". Основни елемент DES-а (такозвана **рунда**) је композиција две такве трансформације (смене и пермутације) текста, одређене кључем. DES има 16 рунди: исту комбинацију трансформација DES примењује на отворени текст 16 пута, видети слику 4. После почетне пермутације IP, блок се раздваја на леву и десну половину, дужине по 32 бита. Постоји укупно 16 идентичних рунди, у којима се користи функција означена са f за комбиновање података са кључем (у свакој рунди користи се други део од 48 бита кључа). После 16-те рунде лева и десна половина се обједињују, па завршна пермутација (инверзна пермутација почетне пермутације) закључује обраду блока.

У свакој рунди се бити кључа циклички померају одређен број пута а онда се увек са истих позиција бира 48 од 56 бита кључа. Десна половина података се проширује на 48 бита "проширујућом пермутацијом" и XOR-ује (сабира по модулу 2, бит по бит) са одговарајућим битима кључа, затим се замењује са 32 бита (читањем по 4 бита из 8 таблица) и још једном пермутује. Ове четири операције чине функцију f. Излаз из блока f комбинује се затим са левом половином података операцијом XOR. Резултат ових операција постаје нова десна половина. Стара десна половина постаје нова лева половина. Ове операције (слика 5) чине рунду алгоритма DES, и понављају се 16 пута.

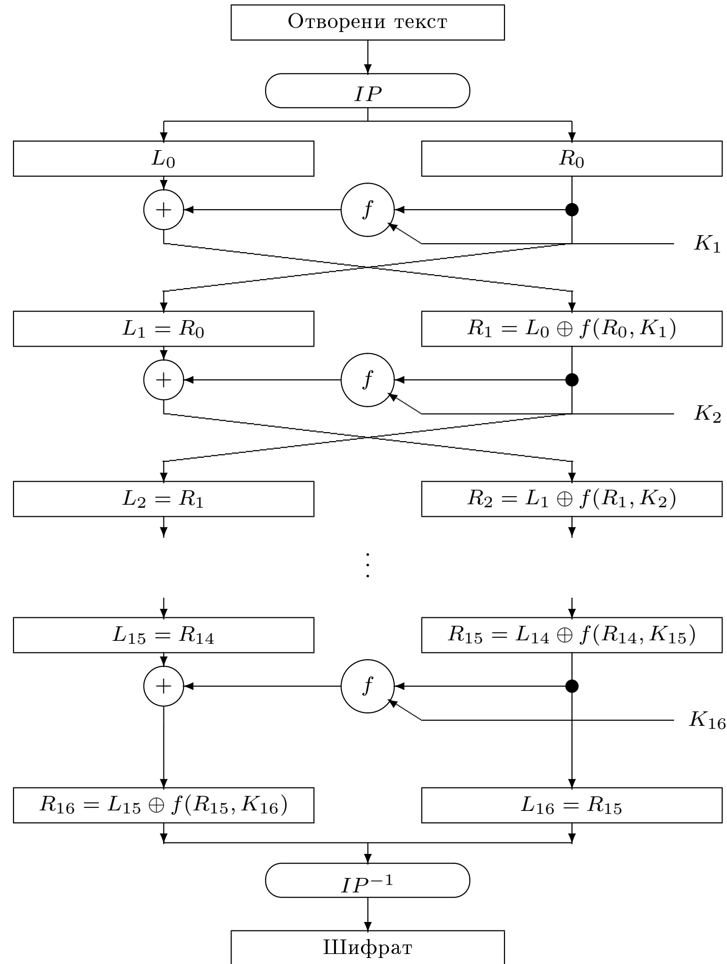


Рис. 4. Алгоритам за шифровање DES.

Ако је B_i резултат i-те рунде, L_i и R_i су лева и десна половина B_i, K_i је 48-битни кључ за рунду i, а f је функција која обухвата смене, пермутовање и XOR-овање са кључем, онда се i-та рунда описује једнакостима

$$
\begin{aligned}
L_i &= R_{i-1} \\
R_i &= L_{i-1} \oplus f(R_{i-1}, K_i)
\end{aligned}
\tag{9.1}
$$

Почетна пермутација која премешта бите у оквиру улазног блока задата је табелом 1. Ова, као и наредне табеле, чита се слева удесно, одозго на доле. На пример, почетна пермутација 1. бит премешта на позицију 58, 2. бит на позицију 50, итд. Јасно је да почетна и завршна пермутација не утичу на сигурност DES-а.

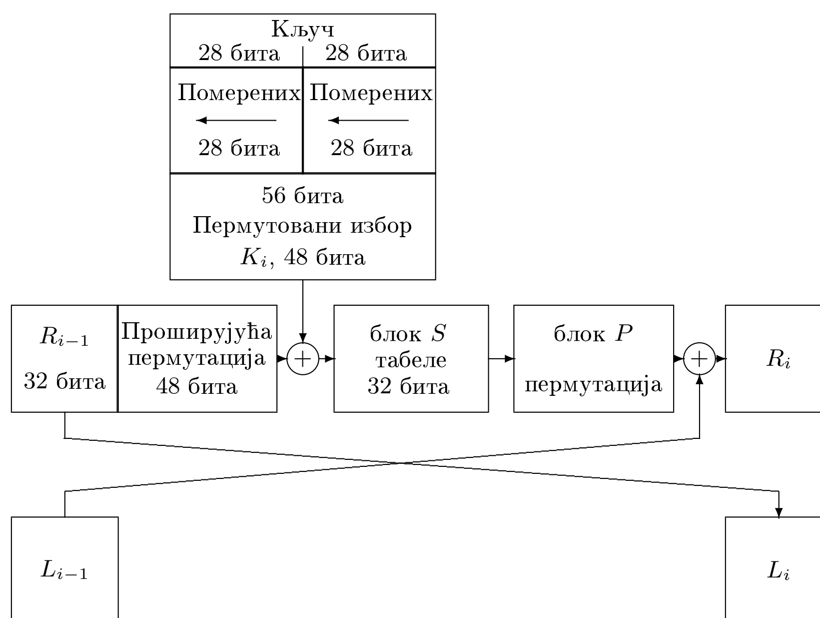


Рис. 5. Једна рунда алгоритма DES.

58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5	63	55	47	39	31	23	15	7

Таблица 1. Почетна пермутација DES-а.

Први корак у трансформисању 64-битног кључа је избацивање сваког осмог бита, видети табелу 2. Изостављени бити користе се за проверу исправности уношења кључа. Од добијеног 56-битног кључа за сваку рунду DES-а генерише се нови 48-битни кључ K_i, $1 \leq i \leq 16$, на следећи начин. Најпре се 56-битни кључ дели на две 28-битне половине. После тога се половине циклички померају улево за једну или две позиције, зависно од редног броја рунде, видети табелу 3. После цикличког померања, бира се 48 од 56 бита кључа. Ова операција бира подскуп бита кључа и истовремено их пермутује, па се зове пермутовани избор; описана је табелом 4. На пример, бит на позицији 14 (17, 33) помереног кључа премешта се на позицију 1 (2, 35) поткључа K_i, а 18-ти бит помереног кључа се игнорише.

57	49	41	33	25	17	9	1	58	50	42	34	26	18
10	2	59	51	43	35	27	19	11	3	60	52	44	36
63	55	47	39	31	23	15	7	62	54	46	38	30	22
14	6	61	53	45	37	29	21	13	5	28	20	12	4

Таблица 2. Пермутација бита кључа за DES.

Проширујућа пермутација је операција која блок R_{i-1} проширује са 32 на 48 бита. Добијени блок се XOR-ује са делом кључа K_i. Затим се дужина поново смањује на 32 бита супституцијом. Тиме што један бит може да утиче на две супституције, зависност од улазних бита простире се брже (такозвани *лавински ефекат*. Једна од основних замисли при пројектовању DES-а је управо постизање циља да један улазни бит утиче на што више излазних бита, и то што раније. Проширујућа пермутација задата је табелом 5, односно сликом 6. Сваки улазни 4-битни блок (b_0, b_1, b_2, b_3) даје 6 бита излаза, тако да се b_0 и b_3 појављују по два пута, а b_1 и b_2 по једном. Табела на месту i, $1 \leq i \leq 48$, садржи индекс улазног бита који се копира на позицију i у излазу. На пример, бит на позицији 3 у улазном блоку копира се на позицију 4 у излазном блоку, а бит на позицији 21 у улазном блоку копира се на позиције 30 и 32 у излазном блоку.

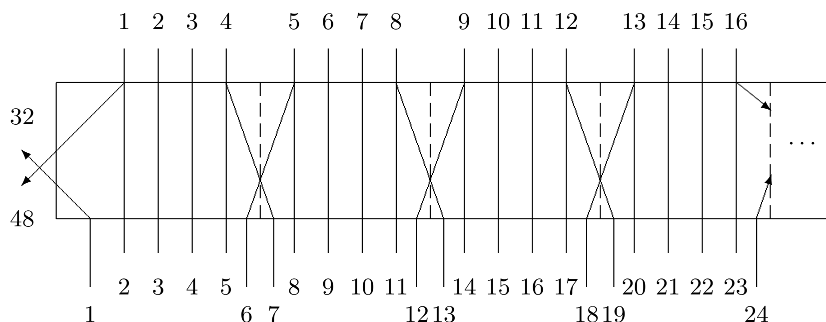


Рис. 6. Проширујућа пермутација DES-а.

рунда	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
број померања	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Таблица 3. Број цикличких померања кључа за поједине рунде DES-а.

14	**17**	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	**33**	48
44	49	39	56	34	53	46	42	50	36	29	32

Таблица 4. Пермутовани избор бита кључа.

32	1	2	3	4	5	4	5	6	7	8	9
8	9	10	11	12	13	12	13	14	15	16	17
16	17	18	19	20	21	20	21	22	23	24	25
24	25	26	27	28	29	28	29	30	31	32	1

Таблица 5. Проширујућа пермутација DES-а.

Пошто се блок K_i XOR-ује са проширеним блоком, добијених 48 бита води се на блок за супституцију. Ова операција изводи се помоћу 8 таблица, **супституционих блокова** $S_1, S_2, \ldots, S_8$, видети табелу 6. Улазних 48 бита дели се на осам 6-битних блокова, адреса за блокове S. Ако улаз блока S представља шест бита $(b_1, b_2, b_3, b_4, b_5, b_6)$, онда је $b_1 b_6$ бинарно записан редни број врсте (од 0 до 3), а $b_2 b_3 b_4 b_5$ је бинарно записан редни број колоне (од 0 до 15). Осам прочитаних елемената (4-битних бројева) из таблица се обједињавају и воде на излаз.

Супституциони блок S_1															
14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
Супституциони блок S_2															
15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
Супституциони блок S_3															
10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
Супституциони блок S_4															
7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
Супституциони блок S_5															
2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
Супституциони блок S_6															
12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
Супституциони блок S_7															
4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
Супституциони блок S_8															
13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Таблица 6. Супституциони блокови DES-а.

Нека је, на пример, улаз за табелу S_6 (другим речима, бити на позицијама од 31 до 36 у блоку добијеном после XOR-овања са K_i) 110011. Први и шести бит дају бинарни број 11, што одговара четвртој врсти табеле. Средња четири бита дају бинарни број 1001, односно 9, што одговара 10-ој колони табеле. Тиме је задат елеменат $(4, 10)$ табеле S_6, односно 14, или бинарно 1110. Дакле, 6-битни блок 110011 замењује се 4-битним блоком 1110.

Ово је критичан корак алгоритма. Све остале операције су линеарне и лако их је анализирати. Супституциони блокови су нелинеарни, и више од осталих елемената доприносе сигурности DES-а.

Резултат фазе супституције је осам 4-битних блокова, који се комбинују у јединствени 32-битни блок. Овај блок обрађује се "пермутационим блоком", који добијене бите пермутује на начин одређен табелом 7 Табела за сваки бит показује на коју позицију ће бити премештен; на пример, бит 4 премешта се на позицију 21, а бит 23 на позицију 3.

На крају се пермутовани блок XOR-ује са левом половином L_{i-1} претходног 64-битног блока. Пошто се замене лева и десна половина, може се почети са наредном рундом.

Излаз из 16-те рунде пермутује се завршном пермутацијом, која је уствари инверзна пермутација полазне пермутације, видети табелу 8. При томе се после последње рунде лева и десна половина не замењују, тј. блок $R_{16}L_{16}$ се доводи на улаз завршне пермутације (видети слику 4).

После свих ових супституција, пермутација, XOR-овања и померања бита, могло би се помислити да је алгоритам за дешифровање потпуно другачији, и исто тако компликован као и алгоритам за шифровање. Напротив, ове операције су комбиноване тако да исти алгоритам изводи и шифровање и дешифровање. Заиста, једнакостима (9.1) еквивалентне су једнакости

$$\begin{aligned} R_{i-1} &= L_i \\ L_{i-1} &= R_i \oplus f(L_i, K_i) \end{aligned}$$

које изражавају улаз у i-ту рунду преко излаза из ње (операција XOR је инверзна самој себи). Једина разлика је у томе што се кључеви K_i морају користити обрнутим редоследом ($K_{16}, K_{15}, \ldots, K_1$ уместо $K_1, K_2, \ldots, K_{16}$). Алгоритам формирања кључева за рунде исти је као код шифровања, изузев што се циклична померања изводе удесно, а број померања се из табеле 3 чита здесна улево, уместо слева удесно (укупан број померања је $4 \times 1 + 12 \times 2 = 28$, па се после 16-те рунде обе половине кључа налазе у почетном положају).

16	7	20	21	29	12	28	17	1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9	19	13	30	6	22	11	4	25

Таблица 7. Пермутациони блок DES-а.

40	8	48	16	56	24	64	32	39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30	37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28	35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26	33	1	41	9	49	17	57	25

Таблица 8. Завршна пермутација DES-а.

Сигурност DES-а у току дужег периода анализирало је много људи. Разматрани су дужина кључа, број итерација и структура супституционих блокова. Посебно су мистериозни супституциони блокови – толико константи, без видљивог разлога зашто су изабране баш такве њихове вредности. Изношене су сумње да је NSA уградила у DES скривену ману, што јој омогућује једноставно декриптирање порука. Те сумње требало је да буду оповргнуте извештајем посебног сенатског комитета 1978. године. Извештај међутим није објављен због тајности; објављен је само његов резиме, у коме се каже да је NSA утицала на развој DES-а тако што је

- IBM убедила да је довољно да дужина кључа буде 56 бита,
- индиректно помогла у развоју супституционих блокова, и
- потврдила да, уз све њихово познавање проблематике, DES нема било какве статистичке или математичке слабости.

Касније, 1992. године један од чланова IBM-овог тима је чак и то оповргао, тврдећи да је DES у потпуности дело људи из IBM-а.

О карактеристикама DES-а објављено је више резултата. У скупу свих 2^{56} могућих кључева, посебно се издвајају четири са особином да поткључеви K_i имају исте вредности у свим рундама (такозвани *слаби кључеви*). Ако се кључеви за DES бирају случајно, онда је вероватноћа да се изабере неки од слабих кључева безначајно мала.

Најозбиљнија примедба на DES односи се на дужину кључа. Још 1979. године процењено је да би се за 20 милиона долара могао направити паралелни рачунар, који би одређивао један кључ дневно. Као што је већ речено, 1998. године је сличан рачунар направљен за 250000 долара.

9.4.1. Начини коришћења DES-а. Поред самог алгоритма DES, стандардизовани су и његови начини коришћења. Основни је ECB (скраћено од Electronic Codebook, електронска кодна књига), а састоји се у подели отвореног текста на блокове од по 64 бита P_i, који се шифрују истим кључем K, $C_i = DES_K(P_i)$, $i = 1, 2, \ldots$. Недостатак овог начина је у томе што ако се у отвореном тексту појаве два иста блока, онда ће и њихови шифрати бити једнаки. Да би се то избегло, предложена су два друга начина, CBC (скраћено од Cipher Block Chaining, уланчавање блокова шифрата) и CFB (скраћено од Cipher Feedback,

У оквиру поступка CBC блок отвореног текста се пре шифровања XOR-ује са претходним блоком шифрата: $C_i = DES_K(P_i \oplus C_{i-1})$, $i = 1, 2, \ldots$ (видети слику 7). Блок C_0 није шифрат, него такозвани **иницијализациони вектор** (на пример, вектор од 64 нуле). Јасно је да шифровање сваког блока зависи од свих претходних блокова. Дешифровање је инверзна операција, $P_i = DES_K^{-1}(C_i) \oplus C_{i-1}$.

Поступак CBC идентичне блокове отвореног текста замењује различитим шифратима само ако су неки претходни блокови отвореног текста различити. Две идентичне поруке и даље дају исти шифрат. Још горе, две поруке са истим почетком имаће исти шифрат, до појаве прве разлике. Начин да се овај проблем избегне је уметање случајног блока бита испред поруке. Шифрат тог блока (XOR-ованог са иницијализационим вектором) је уствари прави иницијализациони вектор, који се преноси као део шифрата, а неопходан је за

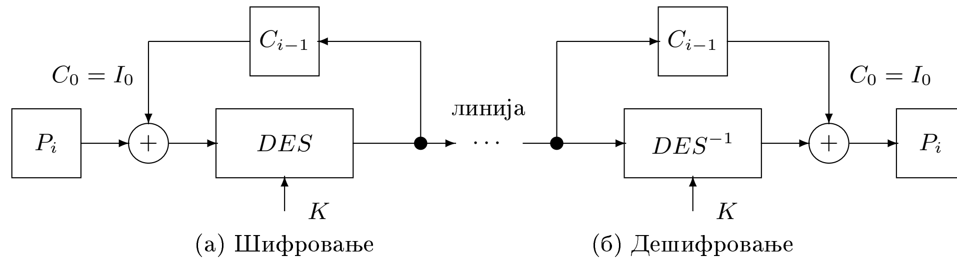


Рис. 7. Употреба DES-а у оквиру поступка CBC.

исправно дешифровање. Важно је да се за сваку поруку користи други иницијализациони вектор. То се може постићи случајним избором иницијализационог вектора за сваку поруку. Други начин је да се искористи редни број поруке. Додавањем иницијализационог вектора за идентичне отворене текстове добијају се различити шифрати. Према томе, нападач не може на основу шифрата да закључи било шта о отвореном тексту.

Поступак CBC решава проблем понављања порука, али има и један недостатак. Претпоставимо да се у току преноса због грешке у преносу променио неки бит шифрата из блока C_{i-1}. Ако промењени блок C_{i-1} означимо са C'_{i-1}, дешифровањем се добија блок $P'_{i-1} = DES_K^{-1}(C'_{i-1}) \oplus C_{i-2}$, који је битно другачији од оригиналног отвореног текста — то је последица особине DES-а да сличне блокове (оне који се разликују на само неколико битских позиција) трансформише шифровањем/дешифровањем у битно различите блокове. Наредни блок $P_i = DES_K^{-1}(C_i) \oplus C'_{i-1}$ има само једну битску грешку, на истој битској позицији где је грешка у C_{i-1}. Сви остали блокови дешифрују се без грешке. Дакле, једна грешка у шифрату квари два узастопна блока при дешифровању. Та особина зове се **простирање грешака**. Да би се овај проблем решио, односно ублажио, потребно је заштитним кодирањем смањити вероватноћу грешке на каналу.

Грешка другог типа, кад се при преносу у шифрат уметне (сувишан) бит, или из њега "испадне" неки бит, фатална је за дешифровање, јер квари све блокове шифрата после грешке. Зато је неопходно да се при преносу очува структура блокова, односно границе између њих.

Са дешифровањем шифрата добијеног поступком CBC, не може се започети пре него што се прими комплетан блок шифрата, што за неке примене представља проблем. Поступак CFB шифрује податке подељене на делове мање од величине блока. Може се, на пример, шифровати један по један ASCII знак отвореног текста (такозвани 8-битни CFB). На сличан начин, може се шифровати бит по бит, коришћењем 1-битног CFB.

На слици 8 приказан је 8-битни CFB. За шифровање i-тог знака отвореног текста, p_i, користи се, на пример, најнижих осам бита z_i резултата шифровања

претходних осам знакова шифрата, $DES_K(c_{i-8}c_{i-7}\dots c_{i-1})$

$$c_i = p_i \oplus z_i = p_i \oplus DES_K(c_{i-8}c_{i-7}\dots c_{i-1}), \quad i \geq 1.$$

Може се замислити да нови знак шифрата улази са десне стране у померачки регистар, са чије леве стране "испада" најстарији знак шифрата; тиме је спреман улаз за DES за шифровање наредног знака p_{i+1} отвореног текста. Да би се процес шифровања могао исправно одвијати од самог почетка, мора се у померачки регистар уписати иницијализациони вектор $(c_{-7}c_{-6}\dots c_0)$. Као и код CBC, за свако наредно шифровање мора се употребити нови, другачији иницијализациони вектор. Његов садржај није тајна, и може се послати отворено, заједно са поруком. Дешифровање је инверзан процес,

$$p_i = c_i \oplus z_i = c_i \oplus DES_K(c_{i-8}c_{i-7}\dots c_{i-1}) \quad i \geq 1.$$

И на пријемној и предајној страни DES извршава само шифровање; у i-том кораку шифровања и дешифровања улази у DES су идентични. Из горњег израза види се да дешифровани знак зависи од претходних девет знакова шифрата. Другим речима, једна грешка у преносу проузрокује грешку код највише девет дешифрованих знакова.

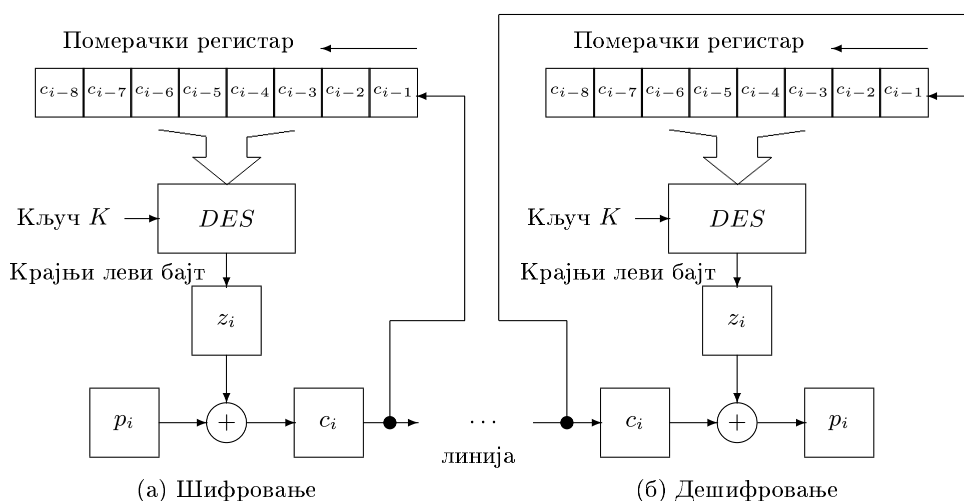


Рис. 8. Употреба DES-а у оквиру 8-битног CFB.

Споменимо да се DES може користити и у тзв. бројачком режиму. Низ улаза за DES су узастопни 64-битни бројеви (узастопна стања бројача). Слично 1-битном CFB, у сваком кораку се од излазних 64 бита узима само један (на пример најнижи бит), који се XOR-ује са наредним битом отвореног текста. Дешифровање се врши XOR-овањем бита шифрата са истим низом излазних бита из DES-а.

9.5. RSA

Систем RSA је најпознатији пример асиметричног шифарског система. Од више асиметричних алгоритама предложених до сада, RSA је најједноставнији за разумевање и реализацију, а такође и најпопуларнији. Име је добио по својим проналазачима (Rivest, Shamir i Adleman), а објављен је 1978. године. Анализа до сада није ни доказала, ни оповргла сигурност овог система.

Сигурност RSA заснива се на сложености факторизације великих бројева. Јавни и тајни кључеви одређени су паром великих простих бројева (од 100 до 200 и више декадних цифара). Сматра се да је одређивање отвореног текста на основу шифрата и кључа за шифровање еквивалентно факторизацији производа два велика проста броја.

Да би се формирао пар кључева, треба најпре изабрати два велика проста броја p и q. Велики прост број са k цифара добија се тако што се редом генеришу случајни бројеви са k цифара, и одбацују — док се не наиђе на прост број. Познато је да је међу бројевима са k декадних цифара "густина" простих бројева приближно $1/\ln 10^k = 1/(k \ln 10)$. То обезбеђује да се после одређеног (не превеликог) броја покушаја мора наићи на прост број. С друге стране, постоје ефикасни алгоритми за проверу да ли је задати број прост (што није у супротности са чињеницом да се не знају ефикасни алгоритми за факторизацију великих бројева). Нека је $n = pq$ производ нађених простих бројева.

Нека је са $\phi(n)$ означена тзв. Ојлерова функција од броја n, једнака броју бројева мањих од n који су узајамно прости са n. Није тешко проверити да за за просте бројеве p и q важи $\phi(pq) = (p-1)(q-1)$. Следећи корак је избор случајног кључа e за шифровање, као броја између 1 и $\phi(n)$, узајамно простог са $\phi(n)$. Поступак за налажење e сличан је оном за налажење бројева p и q: генеришу се случајни бројеви са задатим бројем цифара и за њих се проверава да ли су узајамно прости са $\phi(n)$; добијени бројеви се одбацују све док се не наиђе на неког, који јесте узајамно прост са $\phi(n)$. И у овом случају се зна да се после одређеног броја покушаја са великом вероватноћом мора наићи на број узајамно прост са $\phi(n)$.

Последњи корак је одређивање кључа за дешифровање — таквог броја d за који важи

$$ed \equiv 1 \pmod{(p-1)(q-1)},$$

односно $ed - 1 = l(p-1)(q-1)$ за неки цели број l. Овај проблем еквивалентан је решавању линеарне Диофантове једначине $ed - l(p-1)(q-1) = 1$, у којој су непознате d и l. Ова једначина има решење, јер је највећи заједнички делилац њених коефицијената e и $(p-1)(q-1)$ једнак 1. Као што смо видели у одељку 8.3, за овај проблем постоји ефикасан алгоритам сложености $O(\log n)$, заснован на Еуклидовом алгоритму.

Пар (e, n) је јавни кључ, а број d је тајни кључ. Бројеви p и q више нису потребни и треба их обрисати (треба спречити да их сазна било ко сем власника тајног кључа).

Да би се шифровала порука m, треба је поделити на блокове, тако да се сваки од њих може једнозначно представити бројем између 0 и $n-1$ (за бинарне податке може се изабрати највећи степен двојке мањи од n). Ако су, на пример, p и q бројеви од 100 декадних цифара, онда n има 200 цифара, па сваки блок m_i отвореног текста треба да буде дужине нешто испод 200 цифара. Шифрат ће се састојати од блокова c_i сличне дужине. Поступак шифровања је једноставан:

$$c_i = m_i^e \bmod n.$$

Дешифровање на пријемном крају врши се на сличан начин, степеновањем изложиоцем d по модулу n, што показује следећа теорема.

Теорема 9.1. *Под горњим условима је* $c_i^d \equiv m_i \pmod{n}$.

Доказ теореме директно користи Ојлерову теорему, односно њен специјални случај, Фермаову теорему.

Теорема 9.2 (Ојлерова теорема)**.** *Ако је* $m > 1$ *и број* a *је узајамно прост са* m*, онда је*

$$a^{\phi(m)} \equiv 1 \pmod{m}.$$

Доказательство. Нека су $r_1, r_2, \ldots, r_s$ сви различити позитивни бројеви мањи од m, који су узајамно прости са m. Скуп остатака бројева ar_i за $i = 1, 2, \ldots, s$ при дељењу са m једнак је скупу $\{r_1, r_2, \ldots, r_s\}$, јер су свака два остатка $ar_i \pmod{m}$ и $ar_j \pmod{m}$ за $i \neq j$ — различита: њихова разлика $a(r_i - r_j)$ не може бити дељива са m, јер је a узајмно просто са m, а разлика $r_i - r_j$ по модулу је мања од m. Одатле следи да је $\prod_{i=1}^{s}(ar_i) \equiv \prod_{i=1}^{s} r_i \pmod{m}$. Скраћивањем ове конгруенције производом $\prod_{i=1}^{s} r_i$, узајамно простим са m, добија се тврђење теореме. □

Фермаова теорема, (P. Fermat , 1601-1665.) специјални случај Ојлерове теореме, добија се кад је m једнако простом броју p. Тада је $\phi(m) = p - 1$, па је $a^{p-1} \equiv 1 \pmod{p}$ за свако a које није дељиво са p. Множењем ове конгруенције са a добија се конгруенција $a^p \equiv a \pmod{p}$, која је тачна за сваки цели број a (она је очигледно тачна и ако је a дељиво са p). Другим речима, за прост број p и произвољан цели број a, број $a^p - a$ увек је дељив са p. На сличан начин добија се да је $a^{1+l(p-1)} - a$ дељиво са p за произвољан природан број l. Прелазимо на доказ теореме 9.1.

Доказательство. Заменом се добија

$$c_i^d = m_i^{ed} = m_i^{1+l(p-1)(q-1)},$$

јер је $ed-1$ умножак броја $\phi(n) = (p-1)(q-1)$. Због последице Ојлерове теореме разлика $m_i^{1+l(p-1)(q-1)} - m_i$ је дељива простим бројем p; аналогно, она је дељива и са q. Према томе, ова разлика је дељива производом $pq = n$, односно

$$c_i^d = m_i^{1+l(p-1)(q-1)} \equiv m_i \pmod{n}.$$

□

Преглед елемената система RSA дат је у табели 9. Поступак формирања кључева и шифровања илустроваћемо једним примером.

Пример 9.1. Нека је $p = 47$ и $q = 71$; тада је $n = pq = 3337$. Експонент за шифровање e треба да буде узајамно прост са $\phi(n) = (p-1)(q-1) = 46 \cdot 70 = 3220$. Нека је $e = 79$. Да би се одредио мултипликативни инверз $d = 79^{-1} \pmod{3220}$, најпре се примењује Еуклидов алгоритам на бројеве 3220 и 79:

$$\begin{aligned} \mathbf{3220} &= 40 \cdot \mathbf{79} + \mathbf{60} \\ \mathbf{79} &= 1 \cdot \mathbf{60} + \mathbf{19} \\ \mathbf{60} &= 3 \cdot \mathbf{19} + \mathbf{3} \\ \mathbf{19} &= 6 \cdot \mathbf{3} + \mathbf{1} \\ \mathbf{3} &= 3 \cdot \mathbf{1}. \end{aligned}$$

Одатле се добија представа јединице (највећег заједничког делиоца 3220 и 79) у облику линеарне комбинације два последња члана низа остатака **3220**, **79**, **60**, **19**, **3**: $1 = \mathbf{19} - 6 \cdot \mathbf{3}$. Чланови низа остатака су због прегледности подебљани. Коришћењем горњих једнакости уназад, добијамо израз за јединицу у облику линеарне комбинације прва два елемента низа остатака, односно бројева $e = \mathbf{79}$ и $(p-1)(q-1) = \mathbf{3220}$:

$$\begin{aligned} 1 &= \mathbf{19} - 6 \cdot \mathbf{3} = \mathbf{19} - 6 \cdot (\mathbf{60} - 3 \cdot \mathbf{19}) \\ &= -6 \cdot \mathbf{60} + 19 \cdot \mathbf{19} = -6 \cdot \mathbf{60} + 19 \cdot (\mathbf{79} - \mathbf{60}) \\ &= 19 \cdot \mathbf{79} - 25 \cdot \mathbf{60} = 19 \cdot \mathbf{79} - 25 \cdot (\mathbf{3220} - 40 \cdot \mathbf{79}) \\ &= 1019 \cdot \mathbf{79} - 25 \cdot \mathbf{3220} \end{aligned}$$

Према томе, $d = 1019$. Бројеви e и n представљају јавни кључ, а d је тајни кључ. Бројеви p и q се одбацују (бришу). Да би се шифровала порука

$$m = 6882326879666683,$$

она се прво мора разбити на мање блокове. У овом случају порука се може поделити на троцифрене блокове $m_1 = 688$, $m_2 = 232$, $m_3 = 687$, $m_4 = 966$, $m_5 = 668$ и $m_6 = 3$. Шифровањем првог блока добија се

$$688^{79} \bmod 3337 = 1570 = c_1.$$

Јавни кључ	n	производ два проста броја p и q (чувају се у тајности)
	e	узајамно прост са $(p-1)(q-1)$
Тајни кључ	d	$e^{-1} \pmod{(p-1)(q-1)}$
Шифровање	$c = m^e \pmod{n}$	
Дешифровање	$m = c^d \pmod{n}$	

Таблица 9. Елементи система RSA.

Настављајући на исти начин добија се шифрована порука

$$c = 1570\ 2756\ 2714\ 2276\ 2423\ 0158.$$

Дешифровање се постиже степеновањем кључем $d = 1019$. Дакле,

$$1570^{1019} \pmod{3337} = 688 = m_1.$$

Остатак поруке дешифрује се на исти начин.

Јасно је да се порука може шифровати експонентом d, а дешифровати експонентом e, јер је улога експонената симетрична.

Сигурност система RSA зависи од тежине проблема факторизације великих бројева. Строго говорећи ово није тачно; ради се о хипотези, која никад није доказана, да је израчунавање m, на основу c и e еквивалентно са факторизацијом броја n. Постоји могућност да се пронађе битно другачији поступак криптоанализе RSA. Међутим, ако такав поступак криптоанализе омогућује одређивање d, онда се може искористити за факторизацију. У сваком случају, најприроднији начин напада на RSA је факторизација модула n (видети задатак 9.6). Тренутно се могу растављати на чиниоце бројеви од око 120 декадних цифара. Ако се n изабере тако да има 1024 бита (308 цифара), може се сматрати да овакав напад практично није изводљив. RSA има један практичан недостатак: најбрже реализације RSA су од 100 до 1000 пута спорије од DES-а. Због тога се обично RSA користи само за размену кључева за DES (или неки други блоковски систем), а онда се за сва остала шифровања користи DES.

9.6. Резиме

Криптографија је област у којој ефикасни алгоритми имају значајну примену. Размотрени су примери неких класичних система. Као представници савремених криптографских система приказани су DES и RSA, један симетрични и један асиметрични систем.

Задаци

9.1. Следећи текст добијен је шифровањем Цезаровом шифром са непознатим помаком (абецеда је иста као у одељку 9.3):

EJČDVTCDŽNKVDZDUTSDPTSUJA

Како гласи отворени текст?

9.2. На располагању је један пар $(P, C = DES_K(P))$, при чему се о кључу K зна да се састоји од ASCII знакова из скупа $\{A, B, \ldots, Z\}$. Под претпоставком да се једно шифровање алгоритмом DES може извршити за $1ms$, односно $1\mu s$, колико траје одређивање кључа K провером свих могућих случајева?

9.3. Нека $\bar{x}$ означава негацију блока x, бит по бит. Доказати да из $DES_K(P) = C$ следи $DES_{\bar{K}}(\bar{P}) = \bar{C}$.

9.4. Особа A користи за шифровање својих порука двоструки алгоритам DES: $C = DES_{K_2}(DES_{K_1}(P))$, где су P, C 64-битни блокови поруке и шифрата, а K_1 и K_2 су два независна 56-битна кључа. Конструисати алгоритам за одређивање K_1 и K_2 полазећи од 10 парова (P_i, C_i) таквих да је $C_i = DES_{K_2}(DES_{K_1}(P_i))$. На располагању је меморија величине 2^{64} бајтова, а број шифровања алгоритмом DES треба да буде мањи од 2^{61}.

9.5. Одредити све слабе кључеве алгоритма DES (кључеве за које сви делови кључа K_i имају исте вредности).

9.6. Показати да се, полазећи од јавног кључа (n, e) и тајног кључа d за RSA, може доћи до факторизације броја $n = pq$.

9.7. Корисник A система RSA са параметрима $n = pq$, e и d одлучи да промени кључеве e и d (али не и $n = pq$), јер сумња да је неко сазнао кључ d. Може ли особа B, знајући претходни пар (d, e), да чита наредне поруке за A? Како?

9.8. Показати да се уместо $\phi(n)$ у систему RSA може користити тзв. Кармајклова функција $\chi(n) = \mathrm{NZS}\,(p-1, q-1)$.

9.9. У примеру RSA шифровања 9.1 постоји потенцијални проблем: последњи блок је при шифровању интерпретиран као број 3, исто као и да је последњи блок био нпр. 003. Самим тим, није сигурно да ће се дешифровањем добити оригинална порука. Како се може решити овај проблем?

Глава 10

Редукције

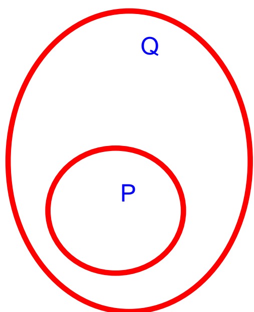
10.1. Увод

Започећемо ово поглавље једном анегдотом. Математичару је објашњено како може да скува чај: треба да узме чајник, напуни га водом из славине, стави чајник на штедњак, сачека да вода проври, и на крају стави чај у воду. А онда су га питали: како може да скува чај ако има чајник, пун вреле воде? Једноставно, каже он; просуће воду и тако свести проблем на већ решен.

У овом поглављу позабавићемо се идејом редукције, односно свођења једног проблема на други. Показаћемо да редукције, иако понекад нерационалне, могу бити и врло корисне. На пример, ако убаците у сандуче поште на Новом Београду писмо за човека који станује одмах поред те поште, писмо ће, без обзира на близину одредишта, прећи пут до Главне поште у Београду, па назад до поште на Новом Београду, и тек онда ће га поштар однети вашем познанику. Чести су случајеви кад није лако препознати специјални случај и за њега скројити ефикасније специјално решење. У пракси је често ефикасније све специјалне случајеве третирати на исти начин. На такву ситуацију често се наилази и при конструкцији алгоритама. Кад наиђемо на проблем који се може схватити као специјални случај другог проблема, користимо познато решење. То решење понекад може бити превише опште или превише скупо. Међутим, у многим случајевима је коришћење општег решења најлакши, најбржи и најелегантнији начин да се проблем реши. Овај принцип се често користи. За неке рачунарске проблеме, на пример рад са неком базом података, обично није неопходно написати програм који решава само тај проблем. Опште решење не мора бити најефикасније, али је много једноставније искористити управо њега.

Претпоставимо да је дат проблем P, који изгледа компликовано, али личи на познати проблем Q. Може се независно (од почетка) решавати проблем P, или се може искористити неки од метода за решавање Q и применити га на P. Постоји, међутим, и трећа могућност. Може се покушати са налажењем **редукције**, односно свођења једног проблема на други. Неформално, редукција је решавање једног проблема коришћењем "црне кутије" која решава други проблем. Редукцијом се може постићи један од два циља, зависно од смера. Решење P, које користи црну кутију за решавање Q, може се трансформисати у алгоритам за решавање P, ако се зна алгоритам за решавање Q. С друге стране, ако се за P зна да је тежак проблем, и зна се доња граница сложености за алгоритме који решавају P, онда је то истовремено и доња

P се своди на Q



Проблем P је специјални случај проблема Q

специјални

Доња граница сложености за ~~општи~~ проблем
не може да буде мања од доње границе сложености за општи проблем

граница сложености за проблем Q. У првом случају је редукција искоришћена за добијање информације о P, а у другом — о Q.

На пример, у одељку 10.4.2 разматрају се проблеми множења и квадрирања матрица. Јасно је да се квадрирање може извести применом алгоритма за множење; према томе, квадрирање матрице може се свести на множење матрица. У одељку 10.4.2 се показује да је могуће помножити две матрице применом алгоритма за квадрирање; другим речима, множење матрица своди се на квадрирање. Сврха ове друге редукције је извођење доказа да се квадрирање матрице не може извести асимптотски брже од израчунавања производа две произвољне матрице (под условима који су размотрени у одељку 10.4.2).

У овом поглављу видећемо неколико примера редукција. Налажење редукције једног проблема на други може бити корисно чак и ако не даје нову доњу или горњу границу сложености проблема. Редукција омогућује боље разумевање оба проблема. Она се може искористити за налажење нових техника за напад на проблем или његове варијанте. На пример, редукција се може искористити за конструкцију паралелног алгоритма за решавање проблема.

Један од ефикасних начина за употребу редукција је дефинисање врло општег проблема, на кога се могу свести многи други проблеми. Такав проблем треба да буде довољно општи, да покрије широку класу проблема, док са друге стране треба да буде довољно једноставан, да би имао ефикасно решење. Линеарно програмирање, један од таквих проблема, размотрићемо у одељку 10.3.

И раније смо већ сретали са примерима редукција — на пример, редукцију проблема налажења транзитивног затворења на проблем налажења свих најкраћих путева (одељак 6.8). Редукције су довољно важне да заслужују да им се посвети посебно поглавље. Оне су поред тога основа и за материјал следећег поглавља.

10.2. Примери редукција

У овом одељку размотрићемо неколико примера употребе редукција као метода за конструкцију ефикасних алгоритама.

10.2.1. Циклично упоређивање стрингова. Почињемо са једноставном варијантом проблема проналажења узорка у тексту.

Проблем. Нека су $A = a_0a_1 \ldots a_{n-1}$ и $B = b_0b_1 \ldots b_{n-1}$ два стринга дужине n. Установити да ли је стринг B циклички померај стринга A.

Проблем је установити да ли постоји индекс k, $0 \leq k \leq n-1$, такав да је $a_i = b_{(k+i) \bmod n}$ за све i, $0 \leq i \leq n-1$. Означимо овај проблем са ЦУС, а основни проблем тражења узорка у тексту са ТУТ (одељак 5.6). Проблем ЦУС може се решити, на пример, изменом алгоритма КМР, описаног у одељку 5.6. Постоји бољи начин да се дође до решења. Идеја је формулисати ЦУС као регуларни пример улаза за проблем ТУТ. Другим речима, тражимо неки *текст* T и *узорак* P, тако да је проналажење P у T еквивалентно утврђивању да је B циклички померај A. Ако нам то пође за руком, онда се решење ТУТ са

petar ciklicki pomeraj arpet?

petar se pronalazi unutar arpetarpet ?

стринговима T и P може искористити за решавање ЦУС са стринговима A и B. Кад се проблем размотри на овај начин, лако је видети решење: за T треба узети AA (односно стринг A надовезан на самог себе). Јасно је да је B циклички померај A ако и само ако је B подстринг стринга AA. Пошто проблем ТУТ умемо да решимо за линеарно време, дошли смо до алгоритма за решавање ЦУС линеарне сложености.

10.2.2. Редукције са упоређивањем низова. Размотримо проблем упоређивања низова из одељка 5.7: дати су низови знакова $A = a_1a_2\dots a_n$, $B = b_1b_2\dots b_m$ су два низа знакова, а задатак је променити A знак по знак, и тако га учинити једнаким низу B. Дозвољене су три едит операције — *уметање*, *брисање* и *замена* знака. Знају се цене сваке операције, а циљ је минимизација цене едитовања. Решење дато у одељку 5.7 заснива се на попуњавању табеле димензија $n\times m$, у којој сваки елеменат одговара *парцијалном* едитовању: елеменат табеле у пресеку i-те врсте и j-те колоне је најмања цена трансформисања првих i знакова A у j првих знакова B. Циљ се постиже израчунавањем елемента у доњем десном углу табеле. Видели смо да се сваки елеменат може израчунати на основу само три "претходна" елемента, што одговара трима могућим варијантама за последњу едит операцију.

Исти проблем може се посматрати на други начин ако табели придружимо усмерени граф. Сваки елеменат табеле одговара чвору графа, односно парцијалном едитовању. Грана (v, w) у графу постоји ако се едитовање које одговара чвору w добија додавањем једне едит операције едитовању које одговара чвору v. Пример таквог графа дат је на слици 1, где је $A = caa$ и $B = aba$. Хоризонталне гране одговарају уметањима, вертикалне брисањима, а дијагоналне заменама знакова. Тако, на пример, пут истакнут на слици 1 одговара брисању c, замени a са a, уметању b и још једној замени a са a. У основној варијанти проблема цене грана су 1, изузев дијагоналних грана код којих се замена врши непромењеним знаком, чија је цена 0. Проблем се на тај начин трансформише у обичан проблем налажења свих растојања од задатог чвора у графу. Свакој грани додељена је тежина, и ми тражимо најкраћи пут од чвора $(0, 0)$ до чвора (n, m). Дакле, проблем налажења едит растојања сведен је на проблем налажења свих растојања од задатог чвора у графу.

Налажење свих растојања од задатог чвора у општем случају није лакше од директног решавања проблема. Ова редукција ипак није бескорисна. Размотримо, на пример, следећу варијанту проблема упоређивања низова. Цене едит операција не морају да зависи само од појединачних знакова. Цена уметања блока знакова унутар другог низа може бити различита од уметања истог броја знакова једног по једног, на различитим местима. Исто може да буде случај и са брисањима. Другим речима, уместо да цене додељујемо појединачним уметањима, брисањима и заменама, цене се могу доделити блоковима уметања, брисања и замена, независно од њихове величине. Или другачије, уметању блока од k знакова може се доделити цена $c_1 + c_2k$, где је c_1 "почетна цена", а c_2 цена за сваки наредни знак. Постоји много других корисних метрика. Оне се много лакше моделирају коришћењем формулације у виду проблема најкраћих

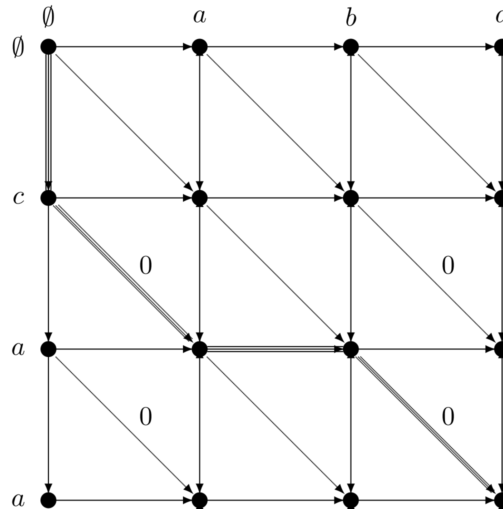
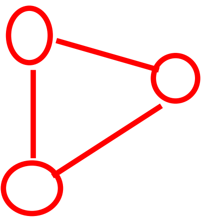


Рис. 1. Граф који одговара низовима $A = caa$ и $B = aba$.

путева, него прилагођавањем полазног проблема. Исто тако, могу се додати нове гране са подгодно изабраним ценама, не мењајући суштину проблема.

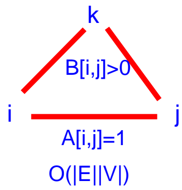
10.2.3. Налажење троуглова у неусмереном графу. Постоји тесна веза између графова и матрица. Граф $G = (V, E)$ са n чворова v_1, v_2, $\dots v_n$, може се представити својом **матрицом повезаности** — квадратном матрицом $A = (a_{ij})$ реда n, таквом да је $a_{ij} = 1$ ако $(v_i, v_j) \in E$, односно $a_{ij} = 0$ у осталим случајевима. Ако је G неусмерени граф, матрица A је **симетрична**. Ако је G тежински граф, онда се такође може представити квадратном матрицом $A = (a_{ij})$ реда n, при чему је елеменат a_{ij} једнак **тежини** гране (v_i, v_j), односно 0, ако те гране нема у графу. Постоје и други начини да се матрица придружи графу. Тако се графу $G = (V, E)$ са n чворова и m грана може придружити $n \times m$ матрица у којој је (i, j) елеменат једнак 1 ако и само ако је i-ти чвор суседан са j-том граном.



Везе графова и матрица не задржавају се само на репрезентацији. Многе особине графова могу се боље разумети анализом одговарајућих матрица. Слично, многе особине матрица откривају се посматрањем одговарајућих графова. Није изненађујуће да се многи алгоритамски проблеми могу решити коришћењем ове аналогије. То ћемо илустровати једним примером.

Проблем. Нека је $G = (V, E)$ неусмерени повезани граф са n чворова и m грана. Потребно је установити да ли у G постоји **троугао**, односно таква три чвора да између свака два од њих постоји грана.

Директно решење обухвата проверу свих трочланих подскупова скупа чворова. Подскупова има $\binom{n}{3} = n(n-1)(n-2)/6$, а пошто се сваки од њих може проверити за константно време, временска сложеност оваквог алгоритма је $O(n^3)$.



Може се конструисати и алгоритам сложености $O(mn)$ (заснован на провери свих парова $(grana, čvor)$ — да ли граде троугао), који је бољи ако граф није густ. Може ли се ово даље побољшати? Приказаћемо сада алгоритам који је асимптотски бржи, и суштински је другачији. Сврха примера је да илуструје везу између графовских и матричних алгоритама.

Нека је A матрица повезаности графа G. Пошто је граф G неусмерен, матрица A је симетрична. Размотримо везу елемената матрице $B = A^2 = AA$ (производ је обичан производ матрица) и графа G. Према дефиницији производа матрица је

$$B[i,j] = \sum_{k=1}^{n} A[i,k] \cdot A[k,j].$$

Из ове једнакости следи да је услов $B[i,j] > 0$ испуњен акко постоји индекс k, такав да су оба елемента $A[i,k]$ и $A[k,j]$ јединице. Другим речима, $B[i,j] > 0$ је испуњено ако и само ако постоји чвор v_k, такав да је $k \neq i$, $k \neq j$ и да су оба чвора v_i и v_j повезана са v_k (претпостављамо да граф нема петљи, односно $A[i,i] = 0$ за $i = 1, 2, \ldots, n$). Према томе, у графу постоји троугао који садржи темена v_i и v_j акко је v_i повезан са v_j и $B[i,j] > 0$. Коначно, у G постоји троугао акко постоје такви индекси i, j да је $A[i,j] = 1$ и $B[i,j] > 0$.

Наведена анализа сугерише алгоритам. Треба израчунати матрицу $B = A^2$ и проверити испуњеност услова $A[i,j] = 1$, $B[i,j] > 0$ за свих n^2 парова (i,j). Сложеност провере овог услова је $O(n^2)$, па преовлађујући део временске сложености алгоритма потиче од множења матрица. Тиме је проблем налажења троугла у графу сведен на проблем множења матрица (прецизније, квадрирања матрице, али као што ћемо видети у одељку 10.4.2, та два проблема су еквивалентна). За множење матрица може се искористити Штрасенов алгоритам и тако добити алгоритам за налажење троугла у графу сложености $O(n^{2.81})$. Прецизније, описана редукција показује да је сложеност овог графовског проблема $O(M(n))$, где је $M(n)$ сложеност множења Булових матрица реда n.

10.3. Редукције на проблем линеарног програмирања

Претходни одељак садржи примере редукција између алгоритама у различитим областима. Покушали смо да трансформишемо један проблем у други, да бисмо могли да искористимо познати алгоритам. У овом одељку ћемо такође разматрати редукције, али са другачијим приступом. Уместо да тражимо кандидата за редукцију сваки пут кад наиђемо на нови проблем, разматрамо "супер–проблеме", на које се могу свести многи други проблеми. Један такав супер–проблем (можда најважнији) је **линеарно програмирање**.

10.3.1. Увод и дефиниције. Садржај многих проблема је максимизирање или минимизирање неке функције, која задовољава задате услове. На пример, код транспортног проблема циљ је максимизирање фукције тока, под условом да су задовољени услови капацитета грана и конзервације тока. Линеарно програмирање је општа формулација таквих проблема, кад су функција и ограничења линеарни. Нека је $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ вектор *променљивих*.

Циљна функција је линеарна функција промењљивих — компоненти вектора $\mathbf{x}$:

$$c(\mathbf{x}) = \sum_{i=1}^{n} c_i x_i, \tag{10.1}$$

где су c_i константе. Циљ линеарног програмирања је пронаћи вредност $\mathbf{x}$ која задовољава задата ограничења (која ће бити наведена) и *максимизира* циљну функцију. Касније ћемо видети да је лако заменити минимизирање циљне функције њеним максимизирањем. Претходно наводимо општи облик задатка линеарног програмирања са три типа ограничења, при чему се у конкретним проблемима не морају појављивати сва три типа ограничења. Показаћемо да се општи проблем може свести на проблем са само два типа ограничења.

Нека су $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_k$ реални вектори врсте једнаке дужине n, и нека су $b_1, b_2, \ldots, b_k$ реални бројеви. **Неједнакосна ограничења** су ограничења облика

$$\mathbf{a}_j \cdot \mathbf{x} \le b_j, \quad 1 \le j \le k, \tag{10.2}$$

при чему су сви симболи сем $\mathbf{x}$ константе. Слична су **једнакосна ограничења**

$$\mathbf{e}_j \cdot \mathbf{x} = d_j, \quad 1 \le j \le m, \tag{10.3}$$

при чему су $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_m$ такође вектори врсте дужине n, а $d_1, d_2, \ldots, d_m$ су реални бројеви.

Обично се додају и посебна **ненегативна ограничења** (иако се она могу схватити као специјални случај претходних ограничења)

$$x_j \ge 0 \quad \text{за све } j \in P, \tag{10.4}$$

где је P задати подскуп скупа $\{1, 2, \ldots, n\}$.

Проблем линеарног програмирања може се формулисати на следећи начин: максимизирати функцију $c(\mathbf{x})$ (10.1), при чему треба да буду задовољена неједнакосна (10.2), једнакосна (10.3) и ненегативна ограничења (10.4). Разуме се да конкретни проблеми не морају да садрже ограничења свих типова.

За овако дефинисан проблем у општем случају постоји више еквивалентних формулација. На пример, једнакосна ограничења типа $\mathbf{e}_j \cdot \mathbf{x} = d_j$ могу се еквивалентно заменити са два неједнакосна ограничења $\mathbf{e}_j \cdot \mathbf{x} \le d_j$ и $\mathbf{e}_j \cdot \mathbf{x} \ge d_j$. Слично, неједнакосно ограничење $\mathbf{a}_i \cdot \mathbf{x} \le b_i$ може се заменити са два еквивалентна, $\mathbf{a}_i \cdot \mathbf{x} + y_i = b_i$ и $y_i \ge 0$, при чему је y_i нова промењљива. У оба случаја замена једног скупа ограничења другим може да повећа број ограничења.

На овом месту нећемо се бавити алгоритмом за решавање проблема линеарног програмирања. Напоменимо само да су постојећи алгоритми довољно ефикасни у пракси, и да свођење неког проблема на линеарно програмирање није само увежбање редукција, него може да буде и добар начин да се тај проблем реши.

10.3.2. Примери редукције на линеарно програмирање. У пракси су проблеми ретко директно задати као проблеми линеарног програмирања. Обично је неопходно увести одговарајуће дефиниције да би се проблем уклопио у ову формулацију.

10.3.2.1. *Транспортни проблем.* Овај проблем разматран је у одељку 6.10. Нека променљиве $x_1, x_2, \ldots, x_n$ представљају вредности тока за свих n грана e_1, e_2, ..., e_n, Циљна функција је вредност укупног тока

$$c(\mathbf{x}) = \sum_{j \in S} x_j,$$

где је S скуп грана које излазе из извора s. Неједнакосна ограничења одговарају ограничењима капацитета

$$x_i \le c_i, \quad 1 \le i \le n,$$

где је c_i капацитет гране e_i. Једнакосна ограничења одговарају условима одржанја тока

$$\sum_{e_i \text{ излази из } v} x_i = \sum_{e_j \text{ улази у } v} x_j, \qquad \text{за све } v \in V \setminus \{s, t\}.$$

На крају, све променљиве треба да задовоље ненегативна ограничења $x_i \ge 0$ за $i \in \{1, 2, \ldots, n\}$. Вредности $\mathbf{x}$ које максимизирају циљну функцију уз ова ограничења одговарају очигледно оптималном току.

10.3.2.2. *Проблем статичког усмеравања.* Нека је $G = (V, E)$ граф који представља комуникациону мрежу. Претпоставимо да сваки чвор $v_i \in V$ може у јединици времена да прими само B_i порука (због једноставности претпоставља се све поруке имају исту дужину). Претпоставимо да нема ограничења на број порука које се могу пренети било којом граном, и да сваки чвор може да чува неограничену залиху порука за слање. Проблем је установити колико порука треба да пренесу поједине гране у јединици времена, да би се максимизирао укупан број порука које мрежа преноси у јединици времена. Ово је проблем статичког усмеравања, јер се претпоставља да чворови увек имају поруке за слање; у пракси потребе за преносом зависе од времена. У графовској формулацији проблем гласи: доделити тежине гранама, тако да је збир тежина грана суседних чвору v_i мањи или једнак од B_i, а да збир тежина свих грана буде максималан.

Овај графовски проблем лако се може формулисати као проблем линеарног програмирања. Свакој грани $e_i = (v, w)$ може се доделити променљива x_i, која представља број порука кроз e_i у јединици времена. Циљна функција је $c(\mathbf{x}) = \sum_i x_i$, а ограничења су

$$\sum_{e_i \text{ је суседна } v} x_i \le B_i, \qquad \text{за све } v \in V,$$

$$x_i \ge 0, \qquad \text{за све } i.$$

10.3.2.3. *Добротворни проблем.* Претпоставимо да n особа желе да упуте помоћ у k установа. Особа i има ограничење s_i на укупан прилог у току године, као и ограничење a_{ij} на износ прилога установи j (на пример, a_{ij} може да буде 0 за неке установе). У општем случају s_i је мање од $\sum_{j=1}^{k} a_{ij}$, па свака особа треба да донесе одлуку о томе коме да упути колики прилог. Претпоставимо даље да свака установа j има ограничење t_j на укупан износ који може да прими (ово ограничење можда није реално, али је ипак интересантно). Циљ је направити алгоритам који максимизира збир прилога.

Овај проблем је уопштење проблема упаривања из одељка 6.9.2. Проблем се може решити поступком сличним поступцима за налажење оптималног упаривања, а може се формулисати и као проблем линеарног програмирања. Укупно има nk променљивих x_{ij}, $1 \le i \le n$, $1 \le j \le k$, где је x_{ij} износ који особа i уплаћује установи j. Циљна функција је

$$c(\mathbf{x}) = \sum_{i,j} x_{ij}.$$

Ограничења су следећа:

$$x_{ij} \le a_{ij} \quad \text{за све } i, j,$$

$$\sum_{j=1}^{k} x_{ij} \le s_i \quad \text{за све } i,$$

$$\sum_{i=1}^{n} x_{ij} \le t_j \quad \text{за све } j.$$

Поред тога, све променљиве наравно морају бити ненегативне.

10.3.2.4. *Проблем придруживања.* Променимо мало добротворни проблем тако да свака особа може да уплати прилог само једној установи, и да свака установа може да прими прилог од само једне особе. Тако добијамо стандардни проблем упаривања, али са тежинама. Сваком могућем упаривању придружен је збир прилога, а циљ је не само пронаћи оптимално упаривање, него и да максимизирати суму прилога. Ово је тежински проблем бипартитног упаривања, или **проблем придруживања**.

Променљиве у овом проблему не могу бити исте као у претходном. Потребно је на неки начин окарактерисати упаривање. Треба обезбедити да је са сваким чвором повезана тачно једна изабрана грана. То се може постићи додељивањем по једне променљиве x_{ij} свакој грани (i, j), тако да је $x_{ij} = 1$ ако је грана изабрана, односно $x_{ij} = 0$ у противном. Циљна функција је

$$c(\mathbf{x}) = \sum_{i,j} a_{ij} x_{ij},$$

а ограничења су

$$\sum_{j=1}^{k} x_{ij} = 1 \quad \text{за све } i,$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \text{за све } j.$$

Поред тога, све променљиве x_{ij} треба да буду ненегативне. Наведена ограничења обезбеђују да је за сваки чвор изабрана највише једна грана.

Са оваквом формулацијом постоји један озбиљан проблем. Променљиве треба да представљају избор типа "да"–"не", а њихове оптималне вредности могу да буду реални (дакле не цели) бројеви! Потребно је додати ограничења да променљиве могу узимати само вредности 0 или 1. У општем случају такав проблем је тешко решити. Линеарни програми чије променљиве морају бити цели бројеви зову се **целобројни линеарни програми**, а њихово решавање је **целобројно линеарно програмирање**. Многи проблеми размотрени у претходним поглављима могу се на природан начин формулисати као проблеми целобројног линеарног програмирања. Међутим, иако се линеарни програми могу ефикасно решавати, целобројни линеарни програми су обично (али не увек) врло тешки. На овај проблем вратићемо се и у следећем поглављу.

10.4. Примена редукција на налажење доњих граница

A - општи

B - специјални

доња граница за специјални
је истовремено и
доња граница за општи

Ако покажемо да се произвољан алгоритам за решавање проблема A може модификовати (не повећавајући му при томе значајно временску сложеност) тако да решава проблем B, онда је доња граница за проблем B истовремено и доња граница за проблем A. Заиста, сваки алгоритам за A је истовремено и алгоритам за B, па је доња граница за B већа или једнака од доње границе за A. Приказаћемо три примера доказа доње границе сложености заснована на редукцији.

10.4.1. Доња граница за конструкцију простог многоугла. Посматрајмо проблем повезивања скупа тачака у равни простим многоуглом (видети одељак 7.3). Видели смо како се тај проблем може решити коришћењем сортирања. Испоставља се да, под одређеним претпоставкама, овај проблем не може да се реши брже од сортирања. Због тога се алгоритам који смо видели за налажење простог многоугла не може побољшати ако се не побољша сортирање (под "побољшањем" се подразумева побољшање за више од константног фактора).

Теорема 10.1. *Нека је познат алгоритам за конструкцију простог многоугла временске сложености $O(T(n))$. Тада постоји алгоритам за сортирање сложености $O(T(n) + n)$.*

Доказательство. Размотримо n тачака на кружници, слика 2. Једини начин да се те тачке повежу простим многоуглом је да се свака тачка повеже са својим суседима на кружници. У противном, ако две тачке које су суседи не би биле повезане, дуж која садржи једну од те две тачке раздвајала би преостале тачке на две групе, које се не могу повезати не пресецајући ову дуж. Посматрајмо сада улаз $x_1, x_2, \ldots, x_n$ за проблем сортирања. Ако бисмо имали алгоритам ("црну кутију") за решавање проблема простог многоугла,

бројеве бисмо могли да сортирамо на следећи начин. Улаз $x_1, x_2, \ldots, x_n$ најпре пресликавамо у углове $y_1, y_2, \ldots, y_n$, из опсега од 0 до 2π, са истим међусобним односима као и $x_1, x_2, \ldots, x_n$. То се може постићи линеарном функцијом $y = 2\pi(x_i - x_{min})/(x_{max} - x_{min})$ која интервал (x_{min}, x_{max}) пресликава у интервал $(0, 2\pi)$; овде је (x_{min}, x_{max}) произвољан интервал који садржи све тачке x_i. Углови се затим на природан начин пресликавају у тачке на јединичној кружници: броју x_i одговара тачка на кружници, чији њен полупречник са фиксном полуправом заклапа угао y_i. Ово пресликавање бројева у тачке изводи се за линеарно време. Сада се може искористити црна кутија за повезивање овог скупа тачака простим многоуглом, за време $O(T(n))$. Као што смо већ видели, многоугао мора да спаја сваку тачку са њеним суседима на кружници. Због тога, пролазећи тачке оним редом којим су оне сложене да би чиниле темена многоугла, добијамо низ тачака сортиран по угловима, а тиме и сортиран полазни низ бројева. Сложеност оваквог сортирања је $O(T(n) + n)$. □

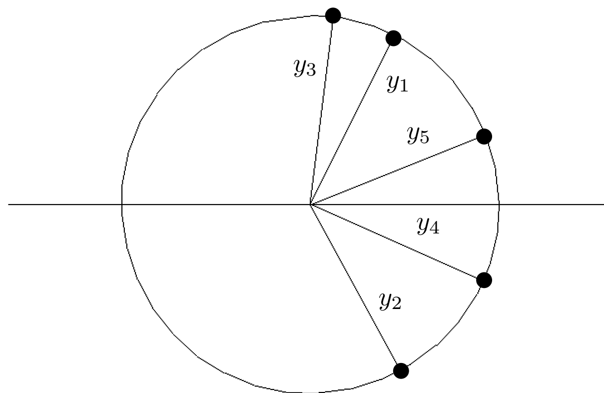


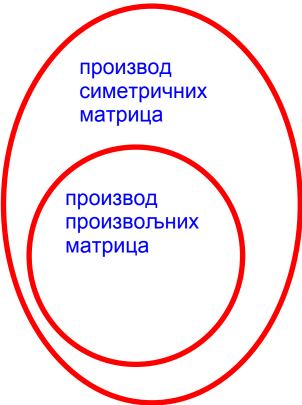
Рис. 2. Придруживање тачака на кружници бројевима.

Да бисмо добили доњу границу за проблем налажења простог многоугла, морамо бити пажљиви у погледу подразумеваног модела рачунања. Доња граница $\Omega(n \log n)$ за проблем сортирања доказана је у одељку 5.3.6 под претпоставком да се ради о моделу стабла одлучивања. Да би ова граница могла да се искористи за проблем налажења простог многоугла, мора се користити исти модел. Другим речима, мора се претпоставити да црна кутија која решава проблем налажења простог многоугла користи $O(T(n))$ упоређивања, на начин који је у складу са моделом стабла одлучивања. Теорема дакле мора да садржи ту претпоставку. Затим се мора показати да је и редукција у складу са моделом стабла одлучивања. У овом случају редукција је регуларна јер приликом доказа доње границе за сортирање није било никаквих ограничења на тип питања дозвољених у оквиру стабла одлучивања. Дакле, упоређивање које ради са x и y координатама које одговарају углу y_i рачуна се као једно упоређивање у стаблу одлучивања. Стабло одлучивања које решава проблем налажења простог

многоугла може се трансформисати у стабло одлучивања за сортирање, без значајне промене висине.

Последица 10.2. *Ако се претпостави модел стабла одлучивања, конструкција простог многоугла који повезује скуп од задатих n тачака у равни захтева у најгорем случају $\Omega(n \log n)$ упоређивања.*

Ова редукција установљава чињеницу да је сортирање централни део решавања проблема конструкције простог многоугла.



10.4.2. Једноставне редукције са матрицама. У одељку 8.5 размотрен је необичан, али асимптотски брз поступак множења матрица. На симетричне матрице (оне којима је елеменат (i, j) једнак елементу (j, i) за сваки пар индекса (i, j)) често се наилази у пракси. Природно је упитати се да ли је једноставније множити симетричне матрице. Није нелогично да симетрија омогућује проналажење бољих израза за множење нпр. матрица реда 3. То би могло да произведе асимптотски бољи алгоритам за множење симетричних матрица. Показаћемо да то ипак није могуће, односно да је множење две симетричне матрице, са тачношћу до на константни фактор, исте сложености као и множење две произвољне матрице.

Означимо проблем израчунавања производа две произвољне матрице са *PrM*, а проблем израчунавања производа две симетричне матрице са SimM. Јасно је да проблем SimM није *тежи* од PrM, јер је SimM специјални случај PrM). Претпоставимо сада да имамо алгоритам који решава SimM. Показаћемо да се тај алгоритам може искористити као црна кутија за решавање општијег проблема PrM. Нека су A и B две произвољне квадратне матрице реда n. Означимо са A^T **транспоновану матрицу** матрице A (тј. матрицу добијену пребацивањем сваког елемента (i, j) матрице на позицију (j, i)). Непосредно се проверава да је тачан следећи израз у коме се појављује производ две *симетричне* $2n \times 2n$ матрице

$$\begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix} \begin{pmatrix} 0 & B^T \\ B & 0 \end{pmatrix} = \begin{pmatrix} AB & 0 \\ 0 & A^T B^T \end{pmatrix}. \tag{10.5}$$

Овде 0 означава $n \times n$ матрицу чији су сви елементи нуле. Редукција је последица чињенице да су матрице са леве стране симетричне. Њихов производ може се израчунати коришћењем алгоритма за решавање проблема SimM. Међутим, горњи леви блок овог производа је управо производ AB. Према томе, проблем PrM може се решити применом алгоритма за решавање SimM на матрице двоструко веће димензије. Тако долазимо до следећег тврђења.

Теорема 10.3. *Ако постоји алгоритам за множење две реалне симетричне $n \times n$ матрице за време $O(T(n))$, при чему је $T(2n) = O(T(n))$, онда постоји алгоритам за множење две произвољне реалне $n \times n$ матрице за време $O(T(n) + n^2)$.*

n^3
(2n)^3=8 n^3

Доказательство. Ако су дате две произвољне $n \times n$ матрице A и B, њихов производ налазимо коришћењем алгоритма за множење симетричних

матрица на основу једнакости (10.5). Потребно је извршити $O(n^2)$ операција за израчунавање A^T и B^T и формирање две симетричне матрице, и $T(2n)$ операција да се помноже две добијене симетричне матрице, из чега непосредно следи тврђење теореме. □

Претпоставка да је $T(2n) = O(T(n))$ није прејака, јер је, на пример, свака полиномијална функција задовољава. Наведена редукција је интересантна само у оквиру доказа доње границе. Теорема тврди да је немогуће искористити симетрију приликом множења матрица, и тако добити асимптотски бржи алгоритам. Наводимо још једну сличну редукцију.

квадрирање матрице

множење произвољне две матрице

Теорема 10.4. *Ако постоји алгоритам који израчунава квадрат реалне* $n \times n$ *матрице за време* $O(T(n))$*, при чему је* $T(2n) = O(T(n))$*, онда постоји алгоритам за множење две произвољне реалне* $n \times n$ *матрице за време* $O(T(n) + n^2)$.

Доказательство. Као и у доказу теореме 10.3, треба пронаћи матрицу чији квадрат садржи довољно информација за израчунавање производа две задате матрице. Решење се заснива на следећем изразу:

$$\begin{pmatrix} 0 & A \\ B & 0 \end{pmatrix}^2 = \begin{pmatrix} AB & 0 \\ 0 & BA \end{pmatrix}.$$

Из њега непосредно следи тврђење теореме. □

10.5. Уобичајене грешке

Са редукцијама треба бити опрезан. Навешћемо примере уобичајених грешака које се могу направити при покушају редукције. Најчешћа грешка је да се редукција изведе у погрешном смеру, а до ње долази углавном при редукцијама за добијање доњих граница. У том случају редукција треба да покаже да је неки проблем P тежак бар колико други проблем Q, чију сложеност знамо. Тада треба поћи од *произвољног* улаза за проблем Q, и показати да се он може решити црном кутијом за решавање проблема P. Размотримо, на пример, следећи покушај редукције проблема сортирања на проблем компресије података Хофмановим кодом (одељак 5.5). Циљ је показати да је $\Omega(n \log n)$ доња граница сложености за Хофманово кодирање.

Полази се од запажања да, ако су учестаности знакова јако различите, онда кодно стабло постаје толико неуравнотежено да се може искористити за сортирање учестаности, видети пример на слици 3. У том случају су знакови у кодном стаблу поређани по опадајућим учестаностима (са најчешћим знаком најближим корену стабла). То управо значи да се Хофманово кодирање може искористити за сортирање учестаности. Према томе, формирање кодног стабла је тешко бар толико колико и сортирање, па је тиме изгледа доказана доња граница сложености $\Omega(n \log n)$.

Грешка у доказу крије се у чињеници да смо пошли од специјалног улаза за проблем сортирања: разматрали смо само такве учестаности, које су значајно

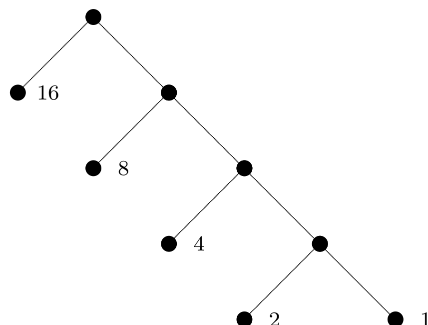


Рис. 3. Хофманово кодно стабло кад се учестаности знакова драстично разликују.

раздвојене. Да би се дошло до доње границе сложености за проблем кодирања, мора се поћи од *произвољног улаза* за проблем сортирања: треба доказати да се произвољни бројеви могу сортирати помоћу алгоритма за Хофманово кодирање. Овакву грешку размотрићемо још једном у наредном поглављу.

Испоставља се да се у овом случају грешка може исправити. Идеја је да се утроши још мало времена да би се променио (произвољни!) улаз за проблем сортирања, тако да се он може искористити као улаз за кодирање. Нека је улаз низ различитих природних бројева $X = x_1, x_2, \ldots, x_n$. Може се претпоставити да су бројеви различити, јер доња граница за сортирање важи и за улазе од различитих бројева (шта више, доња граница је доказана за различите бројеве). Кодно стабло Хофмановог кода за ове учестаности може бити произвољног облика, па је претходно разматрање непримењљиво. Међутим, сваки број x_i може се заменити учестаношћу $y_i = 2^{x_i}$. Пошто за сваки природни број m важи $2^m > \sum_{i<m} 2^i$, кодно стабло имаће облик као на слици 3 (свака учестаност у низу већа је од збира свих учестаности мањих од ње). Према томе, алгоритам за Хофманово кодирање може се искористити за сортирање бројева y_i. Потребно је још уверити се да број операција изведених при редукцији није недозвољено велики. Степеновање може да садржи велики број операција, али то овде није битно, јер доња граница за сортирање обухвата само упоређивања. У одељку 5.3.6 нису учињене никакве претпоставке о броју осталих операција. Према томе, доказано је да, ако се претпостави модел стабла одлучивања, формирање Хофмановог кодног стабла у најгорем случају захтева $\Omega(n \log n)$ упоређивања (потенцијално је могуће брже формирати стабло алгоритмом који није обухваћен моделом стабла одлучивања).

Код алгоритама добијених редукцијом важно је да сама редукција не унесе значајну неефикасност. Размотримо проблем ранца из одељка 4.10 и његово уопштење кад се копије сваког од предмета могу у ранац ставити неограничени број пута. Директна редукција уопштеног проблема на полазни (у коме се сваки предмет може искористити само једном) је следећа. Нека је величина ранца K. У ранац може да стане највише K/s_i копија предмета величине s_i. Према томе,

може се сваки предмет заменити са K/s_i предмета исте величине у полазном проблему. Иако је ова редукција коректна, она није ефикасна, јер је величина проблема значајно повећана. Проблем се може се решити знатно ефикасније (видети задатак 4.16).

10.6. Резиме

Увек је корисно разматрати сличности између проблема. Анализом разлика и сличности између два проблема, обично се стиче боља представа о оба проблема. Кад се наиђе на нови проблем, у већини случајева је корисно упитати се да ли је он сличан неком већ познатом проблему. Понекад сличност између два проблема постаје видљива тек по извршењу компликоване редукције. Посебно су интересантне редукције између матричних и графовских алгоритама. У овом поглављу видели смо више примера редукција, а нове ћемо размотрити и у следећем поглављу.

Линеарно и целобројно програмирање су овде изложени врло кратко. То су веома важни проблеми, и корисно је да се са њима детаљније упозна свако кога интересују алгоритми.

Задаци

10.1. Решити следећу варијанту проблема упоређивања низова. Тражи се низ едит операција које трансформишу дати низ знакова $A = a_1a_2 \ldots a_n$ у други дати низ знакова $B = b_1b_2 \ldots b_m$. Едит операције су уобичајене — уметање, брисање и замена знака, али су њихове цене промењене. Цена уметања знака на позицију i је ci (c је константа), а цена брисања j-тог знака је cj. Цена замене другачијим знаком је и даље 1. Алгоритам треба да одреди низ едит операција који трансформише A у B и има минималну цену.

10.2. Наћи редукцију (у једном од смерова) између проблема налажења максималних тачака у равни (задатак 7.15) и проблема налажења интервала на правој садржаних у другим интервалима (задатак 7.22).

10.3. Варијанта проблема упаривања. Укупно $2n$ студената конкурисало је на n факултета. Размотримо бипартитни граф формиран од два скупа чворова — студената и факултета, при чему грана између неког студента и неког факултета постоји акко студент задовољава услове за пријем на тај факултет. Конструисати алгоритам за максимизирање броја примљених студената, уз услов да ни један факултет не прими више од два студента. Проблем решити свођењем на обичан проблем бипартитног упаривања.

10.4. У школи има n курсева и n наставника. Посматра се бипартитни граф са два скупа по n чворова (курсеви и наставници), при чему грана између курса и наставника постоји ако је наставник квалификован за одржавање курса. Сваки курс могу да држе највише два наставника, а сваки наставник може да држи највише два курса. Конструисати алгоритам (свођењем на познати проблем) за максимизирање укупног броја курсева.

10.5. Нека је $G = (V, E)$ неусмерени граф чијем је сваком чвору v придружен природни број $b(v) \leq d(v)$. **b-упаривање** у графу G дефинише се као подскуп грана $M \subset E$ такав да је сваки чвор $v \in V$ суседан са највише $b(v)$ грана из M (у случају кад је $b(v) = 1$ за свако $v \in V$, M је обично упаривање у G). *Оптимално* b-упаривање је b-упаривање са највећим могућим бројем грана. Свести проблем налажења оптималног b-упаривања на проблем налажења оптималног упаривања.

10.6. У датом графу $G = (V, E)$ издвојен је чвор v и сваком чвору $w \in V$ придружена је цена $c(w)$. Цена усмереног пута $v, x_1, x_2, \ldots, x_k, u$ дефинише се изразом $\sum_{i=1}^{k} c(x_i)$; специјално, ако $(v, u) \in E$, онда је цена пута v, u нула. Конструисати ефикасан алгоритам за налажење путева минималне цене од v до свих осталих чворова у G.

10.7. Општија формулација проблема линеарног програмирања од оне из одељка 10.3 дозвољава оба типа неједнакосних ограничења: први тип је са релацијама "$\leq$", а други са релацијама "$\geq$" (са различитим коефицијентима, наравно). Показати да се ова формулација проблема може свести на првобитну.

10.8. Дат је неусмерен граф $G = (V, E)$ и константа $h > 0$; потребно је придружити гранама $e \in E$ тежине $w(e)$ тако да сума тежина грана суседних произвољном чвору буде мања или једнака од h, а да сума тежина свих грана буде максимална. Формулисати овај проблем као проблем линеарног програмирања.

10.9. Размотримо још једном добротворни проблем из одељка 10.3.2.3, уз претпоставку да није ограничен износ који установе могу да приме. Свести ову варијанту проблема на линеарно програмирање.

10.10. Нека је S скуп од n темена произвољног конвексног многоугла, задатих произвољним редоследом. Доказати да је временска сложеност уређења тачака тако да одређују уобичајену репрезентацију многоугла $\Omega(n \log n)$.

10.11. У одељку 8.5 видели смо да је довољно 7 уместо 8 множења да се израчуна производ две произвољне 2×2 матрице, и да то води асимптотски бољем алгоритму за множење матрица. Квадрат реалне 2×2 матрице може се израчунати помоћу само 5 множења (задатак 8.15). Објаснити зашто ово запажање не противречи теореми 10.4.

10.12. **Доња (горња) троугаона матрица** је квадратна матрица у којој су елементи изнад (испод) главне дијагонале једнаки нули. Доказати да ако постоји алгоритам сложености $O(T(n))$ за множење $n \times n$ доње троугаоне матрице са $n \times n$ горњом троугаоном матрицом, онда постоји алгоритам сложености $O(T(n) + n^2)$ за множење две произвољне $n \times n$ матрице. При томе се може користити претпоставка да је $T(cn) = O(T(n))$ за произвољну константу $c > 0$.

10.13. Доказати да ако постоји алгоритам сложености $O(T(n))$ за множење две $n \times n$ доње троугаоне матрице, онда постоји алгоритам сложености $O(T(n) + n^2)$ за множење две произвољне $n \times n$ матрице. Може се претпоставити да је $T(cn) = O(T(n))$ за произвољну константу $c > 0$.

10.14. Нека је S скуп од n тачака у равни. Тачке из S дефинишу тежински неусмерени граф на следећи начин. Граф је комплетан (свака два његова чвора повезана су граном), а тежине грана једнаке су еуклидском растојању између одговарајућих тачака. Показати да је доња граница сложености алгоритма за налажење минималног повезујућег стабла (MCST) у овом графу $\Omega(n \log n)$.

10.15. Нека је S скуп од n тачака у равни. Дијаметар S је највеће растојање неких двеју тачака из S. Означимо са ДМ проблем одређивања дијаметра скупа од n тачака, а са ДС проблем утврђивања да ли су дисјунктна два скупа A и B од укупно n реалних бројева. Доказати да ако постоји алгоритам који проблем ДМ решава за време $O(T(n))$, онда постоји алгоритам сложености $O(T(n) + n)$ за решавање проблема ДС.

Глава 11

NP-комплетност

11.1. Увод

Ово поглавље је битно другачије од осталих. У претходним поглављима разматране су технике за решавање алгоритамских проблема и њихове примене их на конкретне проблеме. Било би лепо кад би сви проблеми имали елегантне ефикасне алгоритме, до којих долази коришћењем малог скупа техника. Међутим, стварни живот није тако једноставан. Постоји много проблема који се не покоравају до сада размотреним техникама. Могуће је да приликом њиховог решавања није уложено довољно напора, али се са доста разлога може претпоставити да постоје проблеми који *немају* ефикасно опште решење. У овом поглављу описаћемо технике за препознавање неких таквих проблема.

Временска сложеност већине до сада разматраних алгоритама ограничена је неким полиномом од величине улаза. За такве алгоритме кажемо да су **ефикасни** алгоритми, а за одговарајуће проблеме да су **решиви**. Другим речима, за алгоритам кажемо да је ефикасан ако је његова временска сложеност $O(P(n))$, где је $P(n)$ полином од величине проблема n. Подсетимо се да је величина улаза дефинисана као број бита потребних за представљање улаза. Ово није строго формална дефиниција: могуће је исте податке представити битима на више начина; међутим, дужине свих довољно ефикасних представљања истих података не могу се значајно разликовати. Класу свих проблема који се могу решити ефикасним алгоритмом означаваћемо са P (полиномијално време). Ова дефиниција може да изгледа чудно на први поглед. Тако, на пример, алгоритам сложености $O(n^{10})$ се ни по којим мерилима не може сматрати ефикасним; слично, алгоритам са временом извршавања $10^7 n$ тешко је сматрати ефикасним, иако је линеарне сложености. Ипак, ова дефиниција има смисла из два разлога. Прво, она омогућује развој теорије којом ћемо се сада позабавити; друго, и важније, ову дефиницију је лако применити. Испоставља се да огромна већина решивих проблема има практично употребљива решења. Другим речима, временска сложеност полиномијалних алгоритама на које се у пракси наилази је најчешће полином малог степена, ретко изнад квадратног. Обрнуто је обично такође тачно: алгоритми са временском сложеношћу већом од произвољног полинома обично се не могу практично извршавати за велике улазе.

Постоји доста проблема за које се не зна ни један алгоритам полиномијалне сложености. Неки од тих проблема ће једном можда бити решени ефикасним

алгоритмима. Међутим, има разлога за веровање да се многи проблеми не могу решити ефикасно. Волели бисмо да будемо у стању да препознамо такве проблеме, да не бисмо губили време на тражење непостојећег алгоритма. У овом поглављу размотрићемо како приступати проблемима за које се не зна да ли су у класи P. Специјално, размотрићемо једну поткласу таквих проблема, класу такозваних **NP-комплетних проблема**. Ови проблеми могу се груписати у једну класу јер су сви међусобно строго еквивалентни — *ефикасни алгоритам за неки* NP-*комплетан проблем постоји ако и само ако за сваки* NP-*комплетан проблем постоји ефикасни алгоритам.* Широко је распрострањено веровање да не постоји ефикасан алгоритам за било који NP-комплетан проблем, али се не зна доказ оваквог тврђења. Чак и кад би постојали ефикасни алгоритми за решавање NP-комплетних проблема, они би сигурно били врло компликовани, јер се таквим проблемима много истраживача бавило током дугог низа година. До овог тренутка се за стотине (можда хиљаде) проблема зна да су NP-комплетни, што ову област чини врло значајном.

Ово поглавље састоји се из два дела. У првом се дефинише класа NP-комплетних проблема и наводе примери доказа припадности неких проблема тој класи. Затим се разматра неколико техника за *приближно* решавање NP-комплетних проблема. Та решења нису увек оптимална, или се не могу применити на произвољан улаз, али је боље имати њих него ништа.

11.2. Редукције полиномијалне временске сложености

У овом одељку ограничићемо се на **проблеме одлучивања**, тј. разматраћемо само оне проблеме на које се после извршавања одређеног алгоритма може одговорити са "да" или "не". Ово ограничење упрошћава разматранја. Већи део проблема лако се може превести у проблеме одлучивања. На пример, уместо да тражимо оптимално упаривање у задатом графу, можемо да поставимо питање да ли за задато k постоји упаривање величине бар k. Ако умемо да решимо проблем одлучивања, обично можемо да решимо и полазни проблем — бинарном претрагом, на пример.

Проблем одлучивања може се посматрати као **проблем препознавања језика**. Нека је U скуп могућих улаза за проблем одлучивања. Нека је $L \subseteq U$ скуп свих улаза за које је решење проблема "да". За L се каже да је **језик** који одговара проблему, па појмови *проблем* и *језик* могу да се користе равноправно. Проблем одлучивања је установити да ли задати улаз припада језику L. Сада ћемо увести појам редукције полиномијалне сложености између језика, као основни алат у овом поглављу.

Дефиниција 11.1. Нека су L_1 и L_2 два језика, подскупа редом скупова улаза U_1 и U_2. Кажемо да је L_1 **полиномијално сводљив** на L_2, ако постоји алгоритам полиномијалне временске сложености, који дати улаз $u_1 \in U_1$ преводи у улаз $u_2 \in U_2$, тако да $u_1 \in L_1$ ако и само ако $u_2 \in L_2$. Алгоритам је полиномијалан у односу на величину улаза u_1. Претпостављамо да је појам

величине добро дефинисан у просторима улаза U_1 и U_2, тако да је, на пример, величина u_2 ограничена полиномом од величине u_1.

Алгоритам из дефиниције своди један проблем на други. Ако знамо алгоритам за препознавање L_2, онда га можемо *суперпонирати* са алгоритмом редукције и тако добити алгоритам за решавање L_1. Означимо алгоритам редукције са AR, а алгоритам за препознавање L_2 са AL_2. Произвољни улаз $u_1 \in U_1$ може се применом AR трансформисати у улаз $u_2 \in U_2$, и применом AL_2 установити да ли $u_2 \in L_2$, а тиме и да ли $u_1 \in L_1$. Последица специјалног случаја овог разматрања је следећа теорема.

Теорема 11.1. *Ако је језик L_1 полиномијално сводљив на језик L_2, и постоји алгоритам полиномијалне временске сложености за препознавање L_2, онда постоји алгоритам полиномијалне временске сложености за препознавање L_1.*

Доказательство. Доказ следи из претходног разматрања. □

Релација полиномијалне сводљивости није симетрична: полиномијална сводљивост L_1 на L_2 не повлачи увек полиномијалну сводљивост L_2 на L_1. Ова асиметрија потиче од чињенице да дефиниција сводљивости захтева да се *произвољан* улаз за L_1 може трансформисати у еквивалентан улаз за L_2, али не и обрнуто. Могуће је да улази за L_2, добијени на овај начин, представљају само мали део свих могућих улаза за L_2. Према томе, ако је L_1 полиномијално сводљив на L_2, онда можемо сматрати да је проблем L_2 *тежи*.

Два језика L_1 и L_2 су **полиномијално еквивалентни**, или једноставно еквивалентни, ако је сваки од њих полиномијално сводљив на други. Специјално, сви нетривијални проблеми из класе P су еквивалентни (видети задатак 11.1). Релација "полиномијалне сводљивости" је транзитивна, као што показује следећа теорема.

Теорема 11.2. *Ако је L_1 полиномијално сводљив на L_2 и L_2 је полиномијално сводљив на L_3, онда је L_1 полиномијално сводљив на L_3.*

Доказательство. Нека су језици L_1, L_2 и L_3 подскупови скупова могућих улаза редом U_1, U_2 и U_3. Суперпонирањем алгоритама редукције L_1 на L_2, односно L_2 на L_3 добија се алгоритам редукције L_1 на L_3. Произвољан улаз $u_1 \in U_1$ конвертује се најпре у улаз $u_2 \in U_2$, који се затим конвертује у улаз $u_3 \in U_3$. Пошто су редукције полиномијалне сложености, а композиција две полиномијалне функције је полиномијална функција, резултујући алгоритам редукције је такође полиномијалне сложености (то је један од разлога зашто су за меру сложености решивих проблема изабрани полиноми). □

Суштина метода који ћемо примењивати у овом поглављу је да ако се за неки проблем не може наћи ефикасан алгоритам, онда покушавамо да установимо да ли је он еквивалентан неком од проблема за које знамо да су тешки. Класа NP-комплетних проблема садржи стотине таквих међусобно еквивалентних проблема.

11.3. Недетерминизам и Кукова теорема

Теорија NP-комплетности заснована је значајном Куковом (S. A. Cook) теоремом 1971. године. Пре него што формулишемо ову теорему, потребно је да уведемо још неке појмове. Покушаћемо да избегнемо техничке детаља и да дискусију задржимо на интуитивном нивоу. Књига Герија и Џонсона [**2**] (M. R. Garey, D. S. Johnson) добро покрива ову област. Теорија NP-комплетности је део шире области која се зове **сложеност израчунавања**, чији највећи део није предмет ове књиге. Разматрање ћемо ограничити на оне делове који омогућују коришћење те теорије.

До сада се нисмо бавили прецизнијим дефинисањем појма алгоритма. То није важно све док се користе уобичајене операције које подржавају сви рачунари (на пример сабирање, упоређивање, приступ меморији). Прецизна дефиниција алгоритма је битна нпр. за доказивање доњих граница сложености (тако је модел стабла одлучивања коришћен приликом доказа доње границе сложености сортирања, одељак 5.3.6; међутим, тај модел је врло ограничен). Основни модел израчунавања је **Тјурингова машина**. Други често коришћени модел је **машина са униформним приступом**. На срећу, ови и други разумни модели еквивалентни су за наше сврхе, јер се алгоритам за један модел увек може трансформисати у алгоритам за други модел, тако да се време извршавања не мења више него за полиномијални фактор. Кукова теорема је, на пример, доказана коришћењем Тјурингове машине као модела израчунавања, али је тачна и за друге моделе. На овом месту нећемо експлицитно користити ни један специјални модел. Међутим, због јаснијег разумевања појма недетерминистичког алгоритма, ослонићемо се на дефиницију Тјурингове машине и недетерминистичке Тјурингове машине.

Као модел израчунавања може се користити **детерминистичка Тјурингова машина** (ДТМ), која је шематски приказана на слици 1. ДТМ се састоји од *управљачког блока* са коначним бројем стања, *главе за читање/писање* и двострано неограничене *траке*, подељене на бесконачни број једнаких ћелија, нумерисаних целим бројевима $\dots, -2, -1, 0, 1, 2, 3, \dots$.

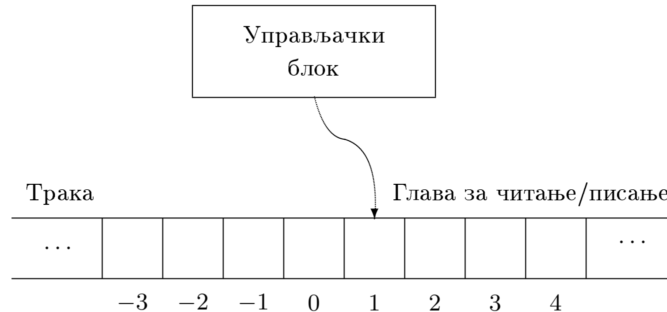


Рис. 1. Шематски приказ детерминистичке Тјурингове машине (ДТМ).

Програм за ДТМ састоји се од следећих компоненти:

(1) коначног скупа Γ *симбола* који се записују на траку, подскупа $\Sigma \subset \Gamma$ *улазних симбола*, и посебно издвојеног *празног симбола* $b \in \Gamma \setminus \Sigma$;
(2) коначног скупа *стања* Q, чији су елементи поред осталих *почетно стање* q_0 и два *завршна стања* q_{da} и q_{NE};
(3) *функције прелаза* $\delta : (Q \setminus \{q_{da}, q_{ne}\}) \times \Gamma \to Q \times \Gamma \times \{-1, 1\}$.

Програм се извршава на следећи начин. Улаз за програм је реч $x \in \Sigma^*$ (састављена од знакова из скупа Σ). Реч се записује на траци у ћелијама са редним бројевима $1, 2, \ldots, |x|$, по један симбол у свакој ћелији (овде $|x|$ означава дужину речи x). Све друге ћелије у почетном тренутку садрже празан симбол, па се зато каже да су празне. Програм почиње са радом налазећи се у стању q_0, при чему се глава за читање/писање налази над ћелијом са бројем 1. Затим се процес израчунавања наставља корак по корак. Ако је тренутно стање q једнако било q_{da}, било q_{ne}, онда се процес израчунавања завршава, и то са резултатом "да" ако је $q = q_{da}$, односно "не" ако је $q = q_{ne}$. У противном, тренутно стање припада скупу $Q \setminus \{q_{da}, q_{ne}\}$, а глава за читање/писање чита са траке неки симбол $s \in \Gamma$, па је дефинисана вредност $\delta(q, s)$. Претпоставимо да је $\delta(q, s) = (q', s', \Delta)$. У том случају глава брише знак s, уместо њега пише знак s' и помера се за једну ћелију улево ако је $\Delta = -1$, односно удесно ако је $\Delta = +1$. Истовремено, управљачки блок прелази из стања q у стање q'. Тиме се завршава један корак процеса израчунавања, и програм је спреман за извршавање следећег корака (ако он постоји).

У табели 1 приказан је пример једноставног програма M за ДТМ. Функција прелаза δ дефинисана је таблицом, у којој се у пресеку врсте q и колоне s налази вредност $\delta(q, s)$. На слици 2 приказано је извршавање програма M са улазом $x = 10100$; приказани су стање, положај главе и садржаји непразних ћелија траке пре и после сваког корака.

Запажамо да се израчунавање завршава после осам корака у стању q_{da}, па се за улаз 10100 добија одговор "да". У општем случају кажемо да програм M, који има улазни алфабет Σ, *прихвата* реч $x \in \Sigma^*$, ако и само ако се, примењен на улаз x, зауставља у стању q_{da}. Језик L_M који *препознаје* програм M, задаје се изразом

$$L_M = \{x \in \Sigma^* | M \text{ прихвата } x\}.$$

Није тешко видети да програм из табеле 1 препознаје језик

$$\{x \in \{0,1\}^* | \text{ два последња знака } x \text{ су нуле}\},$$

односно, ако x схватимо као бинарне цифре улазног броја N, онда M проверава да ли је N дељиво са 4.

$\Gamma = \{0, 1, b\}, \ \Sigma = \{0, 1\}$
$Q = \{q_0, q_1, q_2, q_3, q_{da}, q_{ne}\}$

q	0	1	b
q_0	$(q_0, 0, +1)$	$(q_0, 1, +1)$	$(q_1, b, -1)$
q_1	$(q_2, b, -1)$	$(q_3, b, -1)$	$(q_{ne}, b, -1)$
q_2	$(q_{da}, b, -1)$	$(q_{ne}, b, -1)$	$(q_{ne}, b, -1)$
q_3	$(q_{ne}, b, -1)$	$(q_{ne}, b, -1)$	$(q_{ne}, b, -1)$

$\delta(q, s)$

Таблица 1. Пример програма за ДТМ $M = (\Gamma, Q, \delta)$.

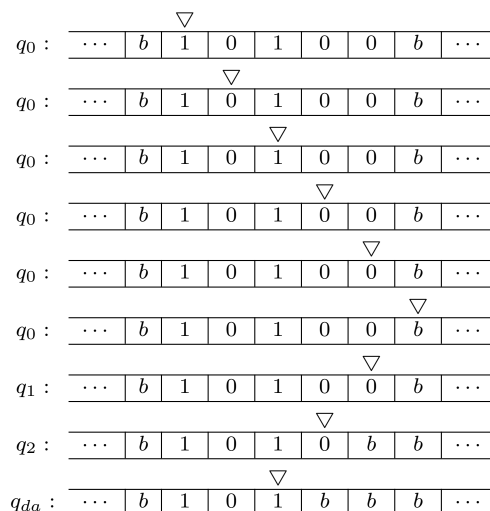


Рис. 2. Извршавање програма M из табеле 1 са улазом 10100 на ДТМ.

Приметимо да дефиниција препознавања језика не захтева да се програм зауставља за све улазе из Σ^*; довољно је да се M зауставља за улазе из L_M. Ако x припада скупу $\Sigma^* \setminus L_M$, онда се извршавање програма M на улазу x или завршава у стању q_{ne}, или траје неограничено. С друге стране, програм за ДТМ који одговара нашем схватању алгоритма, требало би да се зауставља на свим речима улазног алфабета. У том смислу програм из табеле 1 јесте алгоритамски, јер се, почевши извршавање са било којом речи од симбола $0, 1$, зауставља (долази до неког од завршних стања).

Имајући у виду модел ДТМ, лако је прецизно дефинисати временску сложеност. *Време* потребно програму M за ДТМ за израчунавање са улазом x је број корака које ДТМ извршава до заустављања. Нека је $\mathbf{Z}^+$ скуп целих ненегативних бројева. Ако се програм M зауставља за *све* улазе $x \in \Sigma^*$, онда се *временска сложеност* $T_M : \mathbf{Z}^+ \to \mathbf{Z}^+$ тог програма дефинише као

$$T_M(n) = \max\left\{ m \,\middle|\, \begin{array}{l} \text{постоји таква реч } x \in \Sigma^*,\ |x| = n, \text{ да је време} \\ \text{извршавања } M \text{ са улазом } x \text{ једнако } m \end{array} \right\},$$

Програм M је *полиномијални програм за ДТМ* ако постоји полином p, такав да је за све $n \in \mathbf{Z}^+$ испуњено $T_M(n) \le p(n)$.

Сада се може и формално дефинисати класа језика P:

$$\mathrm{P} = \{L \mid \text{постоји полиномијални програм } M \text{ за ДТМ, за који је } L_M = L\}.$$

Термин "полиномијални алгоритам" користићемо углавном неформално. Његов формални еквивалент је полиномијални програм за ДТМ. Међутим, због онога што је речено о еквивалентности "реалних" модела рачунара у односу на појам "полиномијално време рада", ако се формално класа P дефинише за произвољни од тих модела, добиће се иста класа језика.

Због тога, доказујући да се неки проблем може решити полиномијалним алгоритмом, нема потребе улазити у детаље одговарајуће ДТМ. Као и до сада, разматраћемо алгоритме у машински независном стилу, као да они раде директно са компонентама самог проблема (скуповима, графовима, бројевима и слично), а не са кодираним улазом за проблем. У случају постојања жеље и стрпљења, за сваки полиномијални алгоритам може конструисати одговарајући полиномијални програм за ДТМ. Наша неформална разматрања алгоритама треба схватати као упутство за такву кострукцију.

Сада ћемо дефинисати другу важну класу језика (односно проблема одлучивања) — класу NP. Пре него што пређемо на формалну дефиницију засновану на Тјуринговим машинама, покушаћемо да разјаснимо смисао појмова на којима се заснива класа NP. Размотримо као пример проблем трговачког путника.

Проблем. Задат је тежински граф $G = (V, E)$ са $|V| = n$ чворова (тако да је за произвољне чворове – градове $v_i, v_j \in V$, тежина гране $(v_i, v_j) \in E$ једнака $d(v_i, v_j)$) и број $B \in \mathbf{Z}^+$. Установити да ли у G постоји Хамилтонов циклус са збиром тежина грана мањим или једнаким од B. Другим речима, постоји ли такав низ чворова $(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)})$ да је

$$\sum_{i=1}^{n-1} d(v_{\pi(i)}, v_{\pi(i+1)}) + d(v_{\pi(n)}, v_{\pi(1)}) \leq B?$$

Полиномијални алгоритам за решавање овог проблема није познат. Претпоставимо, међутим, да је за неки конкретан улаз за овај проблем добијен одговор "да". Ако у то посумњамо, можемо захтевати "доказ" тог тврђења — низ чворова који има тражену особину. Кад имамо овакав низ чворова, лако можемо да проверимо да ли је то Хамилтонов циклус, да израчунамо његову дужину и упоредимо је са задатом границом B. Поред тога, очигледно је временска сложеност оваквог алгоритма провере полиномијална у односу на величину улаза.

Управо појам полиномијалне проверљивости карактерише класу NP. Приметимо да проверљивост за полиномијално време не повлачи и могућност решавања за полиномијално време. Утврђујући да се за полиномијално време може проверити одговор "да" за проблем трговачког путника, ми не узимамо у обзир време које нам може бити потребно за проналажење жељеног циклуса у експоненцијалном броју свих могућих циклуса у графу. Ми само тврдимо да се за сваки задати низ чворова и за конкретан улаз u, за полиномијално време може проверити да ли тај низ "доказује" да је за улаз u одговор "да".

Класа NP се неформално може дефинисати помоћу појма *недетерминистичког алгоритма.* Такав алгоритам састоји се од две различите фазе: *фазе погађања* и *фазе провере*. За задати улаз u, у првој фази се изводи просто "погађање" неке структуре S. Затим се u и S заједно предају као улаз фази провере, која се изводи на обичан (детерминистички) начин, па се завршава

одговором "да" или "не", или се извршава бесконачно дуго. Недетерминистички алгоритам "решава" проблем одлучивања Π, ако су за произвољни улаз $u \in U_\Pi$ за овај проблем испуњена следећа два услова:

(1) Ако $u \in L_\Pi$, онда постоји таква структура S, чије би погађање за улаз u довело до тога да се фаза провере са улазом (u, S) заврши одговором "да".

(2) Ако $u \notin L_\Pi$, онда *не постоји* таква структура S, чије би погађање за улаз u обезбедило завршавање фазе провере са улазом (u, S) одговором "да".

На пример, недетерминистички алгоритам за решавање проблема трговачког путника могао би се конструисати користећи као фазу погађања избор произвољног низа градова (чворова), а као фазу провере — горе поменути алгоритам "провере доказа" за проблем трговачког путника. Очигледно је да за произвољан конкретан улаз u постоји такво погађање S, да се као резултат рада фазе провере са улазом (u, S) добија "да", онда и само онда, ако за улаз u постоји Хамилтонов циклус тражене дужине.

Каже се да недетерминистички алгоритам који решава проблем одлучивања Π ради за "полиномијално време", ако постоји полином p такав да за сваки улаз $u \in U_\Pi$ постоји такво погађање S, да се фаза провере са улазом (u, S) завршава са одговором "да" за време $p(|u|)$. Из тога следи да је величина структуре S обавезно ограничена полиномом од величине улаза, јер се на проверу погађања S може утрошити највише полиномијално време.

Класа NP, дефинисана неформално, — то је класа свих проблема одлучивања који при разумном кодирању могу бити решени недетерминистичким (N – nondeterministic) алгоритмом за полиномијално (P – polynomial) време. На пример, проблем трговачког путника припада класи NP.

У сличним неформалним дефиницијама термин "решава" треба опрезно користити. Треба да буде јасно да је основни смисао "полиномијалног недетерминистичког алгоритма" у томе да објасни појам "провериљивости за полиномијално време", а не у томе да буде реални метод решавања проблема одлучивања. За сваки конкретан улаз такав алгоритам има не једно, него неколико могућих извршавања — по једно за свако могуће погађање.

На крају ћемо формализовати наведену дефиницију у терминима језика и Тјурингових машина. Формални еквивалент недетерминистичког алгоритма је програм за *недетерминистичку Тјурингову машину са једном траком* (НДТМ). Због једноставности користићемо мало нестандардни модел НДТМ. Модел НДТМ има исту структуру као и ДТМ; разлика је само у томе што НДТМ има *погађачки блок* са својом *главом*, која може само да пише по траци (видети слику 3). Погађачки блок контролише записивање "погађања", и то му је једина сврха.

Програм за НДТМ дефинише са на исти начин као и програм за ДТМ, и у ту сврху користе се: алфабет симбола траке Γ; улазни алфабет Σ; празан знак b; скуп стања Q, чији су елементи поред осталих почетно стање q_0, завршна стања q_{da} и q_{ne}; функција прелаза $\delta : (Q \setminus \{q_{da}, q_{ne}\}) \times \Gamma \to Q \times \Gamma \times \{-1, 1\}$. Извршавање програма за НДТМ са улазом $x \in \Sigma^*$ за разлику од извршавања програма за ДТМ има две различите фазе. У првој фази изводи се "погађање". У почетном тренутку реч x записана је на траци у ћелијама са редним бројевима $1, 2, \ldots, |x|$ (остале ћелије су празне), глава за читање/писање "стоји" над ћелијом са бројем

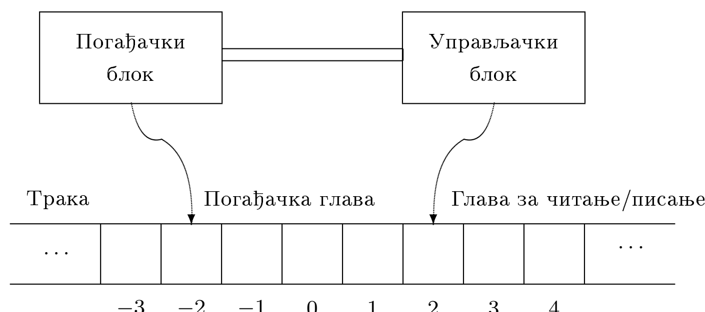


Рис. 3. Шематски приказ недетерминистичке Тјурингове машине са једном траком (НДТМ).

1, глава за писање погађачког блока стоји код ћелије са бројем -1, а управљачки блок је "пасиван". Затим погађачки блок почиње да управља погађачком главом, која извршава по један корак у сваком тренутку, па или пише један знак из алфабета Γ у ћелију која се налази наспрам ње и помера се за једну ћелију улево, или се зауставља. У другом случају погађачки блок прелази у пасивно стање, а управљачки блок почиње рад из стања q_0. Погађачки блок (док ради) одлучује да ли да настави са радом (а онда и који знак из Γ да напише на траци), или да пређе у пасивно стање, при чему то ради потпуно произвољно. На тај начин погађачки блок до тренутка завршетка свог рада може да испише произвољну реч из Γ^*, па је чак могуће и да се никад не заустави.

Фаза провере почиње оног тренутка кад управљачки блок пређе у стање q_0. Почевши од тог тренутка, извршавање програма за НДТМ остварује се по истим правилима као извршавање програма за ДТМ. Погађачки блок и његова глава постају неактивни, пошто су извршили своју улогу записујући на траци реч–погађање. Наравно, реч–погађање може бити учитана (и обично јесте) од стране главе за читање/писање у процесу провере. Израчунавање се завршава онда кад управљачки блок пређе у једно од два завршна стања (q_{da} или q_{ne}); оно се зове *прихватајуће израчунавање* ако се зауставља у стању q_{da}. Сва остала израчунавања, без обзира да ли се завршавају или не, зову се *неприхватајућа израчунавања*.

Приметимо да произвољни програм M за НДТМ може да има бесконачно много могућих израчунавања за дати улаз x, по једно за сваку реч–погађање из Γ^*. Кажемо да програм M за НДТМ *прихвата* x, ако је бар једно од његових израчунавања са улазом x прихватајуће. Језик који програм M *препознаје* је језик

$$L_M = \{x \in \Sigma^* | M \text{ прихвата } x\}.$$

Време потребно НДТМ програму M за прихватање речи $x \in L_M$ — то је минимални број корака који се извршавају у фази погађања и фази провере до тренутка заустављања у стању q_{da}, при чему се минимум тражи по свим прихватајућим израчунавањима програма M са улазом x. Временска сложеност програма M за НДТМ — то је функција $T_M : \mathbf{Z}^+ \to \mathbf{Z}^+$ дефинисана на следећи начин

$$T_M(n) = \max\left(\{1\} \cup \left\{ m \,\middle|\, \begin{array}{l} \text{постоји таква реч } x \in L_M,\ |x| = n, \text{ да је време прих-} \\ \text{ватања } x \text{ програмом } M \text{ једнако } m \end{array} \right\}\right).$$

Приметимо да временска сложеност програма M зависи само од броја корака који се извршавају у прихватајућим израчунавањима, а такође да стављамо $T_M(n) = 1$ ако не постоји ни један улаз величине n кога прихвата програм M.

НДТМ програм је *НДТМ програм са полиномијалним временом извршавања* ако постоји полином p, такав да је $T_M(n) \le p(n)$ за све $n \ge 1$. На крају, класа NP формално се

дефинише са

$$\mathrm{NP} = \left\{ L \,\middle|\, \begin{array}{l} \text{постоји НДТМ програм } M \text{ са полиномијалним временом} \\ \text{извршавања, за који је } L_M = L \end{array} \right\}.$$

Није тешко установити везу између ових формалних и претходних неформалних дефиниција. Ма један детаљ треба посебно обратити пажњу Док недетерминистички алгоритам погађа структуру S која на неки начин зависи од конкретног улаза за проблем, погађачки блок НДТМ потпуно игнорише улазну реч. Иако се *произвољна* реч из Γ^* може добити као резултат рада погађачког блока, то није битно, јер се програм за НДТМ увек може конструисати тако да фаза провере почиње провером да ли погађање (при имплицитној интерпретацији коју програм даје речима) одговара погађајућој структури за задати улаз. У противном програм може одмах да пређе у завршно стање q_{ne}.

Пошто се сваки реалистички модел рачунара може допунити аналогом размотреног "погађачког блока са главом за писање", наше формалне дефиниције могле би се парафразирати за сваки други стандардни модел рачунара. Сви такви модели еквивалентни су са тачношћу до детерминистичких полиномијалних израчунавања, па ће све на тај начин добијене верзије класе NP бити идентичне. Према томе, исправно је поистовећивање формално дефинисане класе NP са класом свих проблема одлучивања који се могу "решити" недетерминистичким алгоритмима са полимомијалним временом извршења.

Недетерминистички алгоритми су врло моћни, али њихова снага није неограничена. Постоје проблеми који се не могу ефикасно решити недетерминистичким алгоритмом. На пример, посматрајмо следећи проблем: да ли је величина оптималног упаривања у задатом графу једнака тачно k? Недетерминистичким алгоритмом лако се може пронаћи упаривање величине k, ако оно постоји, али се не може лако установити (чак ни недетерминистички) да не постоји веће упаривање.

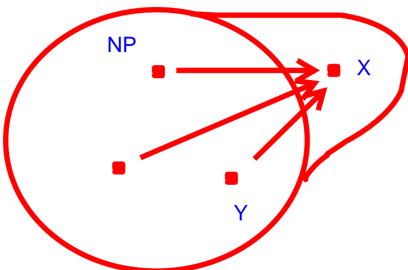
Класа проблема за које постоји недетерминистички алгоритам полиномијалне временске сложености зове се NP. Изгледа разумно сматрати да су недетерминистички алгоритми моћнији од детерминистичких, али да ли је то тачно? Један начин да се то докаже био би да се пронађе неки NP проблем који није у P. То до сада никоме није пошло за руком. У противном, да би се доказало да су ове две класе једнаке (односно P=NP), требало би показати да сваки проблем из класе NP може да буде решен детерминистичким алгоритмом полиномијалне временске сложености. Ни овакво тврђење нико није успео да докаже (и мало је оних који верују у његову тачност). Проблем утврђивања односа између P и NP познат је као **проблем** P=NP.

Сада ћемо дефинисати две класе, које не само да садрже бројне важне проблеме за које се не зна да ли су у P, него садрже **најтеже** проблеме у NP.

Дефиниција 11.2. Проблем X је **NP-тежак проблем** ако је сваки проблем из класе NP полиномијално сводљив на X.

Дефиниција 11.3. Проблем X је **NP-комплетан проблем** ако (1) припада класи NP, и (2) X је NP-тежак.

Последице дефиниције класе NP-тешких проблема је да, ако се за било који NP-тежак проблем докаже да припада класи P, онда је P=NP.



Кук је 1971. године доказао да *постоје* NP-комплетни проблеми. Специјално, навео је пример једног таквог проблема, који ћемо ускоро приказати. Када се зна један NP-комплетан проблем, докази да су други проблеми NP-комплетни постају једноставнији. Ако је дат нови проблем X, довољно је доказати да је Куков проблем (или било који други NP-комплетан проблем) полиномијално сводљив на X. То следи из следеће леме.

Лема 11.1. *Проблем X је* NP-*комплетан ако (1) X припада класи* NP*, и (2') постоји* NP-*комплетан проблем Y који је полиномијално сводљив на X.*

Доказательство. Проблем Y је према услову (2) NP-тежак, па је сваки проблем у класи NP је полиномијално сводљив на Y. Пошто је Y полиномијално сводљив на X, а полиномијална сводљивост је транзитивна релација, сваки проблем у класи NP је полиномијално сводљив и на проблем X. □

Много је лакше доказати да су два проблема полиномијално сводљива, него директно доказати да је испуњен услов (2). Због тога је Куков резултат камен темељац целе теорије. Даље, са појавом нових проблема за које се зна да су NP-комплетни, расте број могућности за доказивање услова (2'). Убрзо пошто је Куков резултат постао познат, Карп (R. M. Karp) је за 24 важна проблема показао да су NP-комплетни. Од тог времена је за стотине проблема (можда хиљаде, зависно од начина бројања варијација истог проблема) доказано да су NP-комплетни. У следећем одељку приказаћемо пет примера NP-комплетних проблема, са доказима њихове NP-комплетности. Навешћемо такође (без доказа) више NP-комплетних проблема. Најтежи део доказа је обично (не увек) доказ испуњености услова (2').

Изложићемо сада проблем за који је Кук доказао да је NP-комплетан, са идејом доказа. Проблем је познат као **проблем задовољивости** (SAT, скраћеница од satisfiability). Нека је S Булов израз у **конјуктивној нормалној форми** (КНФ). Другим речима, S је конјункција више *клауза* (дисјункција група *литерала* — симбола променљивих или њихових негација). На пример, $S = (x + y + \bar{z}) \cdot (\bar{x} + y + z) \cdot (\bar{x} + \bar{y} + \bar{z})$, где сабирање, односно множење одговарају дисјункцији, односно конјункцији (*или*, односно *и*), а свака променљива има вредност 0 (нетачно) или 1 (тачно). Познато је да се свака Булова функција може представити изразом у КНФ. За Булов израз се каже да је **задовољив**, ако постоји такво додељивање нула и јединица променљивим, да израз има вредност 1. Проблем SAT састоји се у утврђивању да ли је задати израз задовољив (при чему није неопходно пронаћи одговарајуће вредности променљивих). У наведеном примеру израз S је задовољив, јер за $x = 1$, $y = 1$ и $z = 0$ има вредност $S = 1$.

Проблем SAT је у класи NP јер се за (недетерминистички) изабране вредности променљивих за полиномијално време (од величине улаза — укупне дужине формуле) може проверити да ли је израз тачан. Проблем SAT је NP-тежак јер се извршавање програма НДТМ за алгоритам који решава произвољан изабрани проблем из класе NP може описати Буловим изразом (доказ ове чињенице није једноставан). Под "описати" се овде подразумева да ће израз

бити задовољив ако и само ако Тјурингова машина са задатим улазом завршава рад и долази до прихватајућег стања. То није лако урадити и такав израз може да буде дугачак и компликован, али његова величина је ограничена полиномом од броја корака које извршава Тјурингова машина. Према томе, улаз за сваки проблем, за који постоји недетерминистички полиномијални алгоритам, може се трансформисати у улаз за проблем SAT.

Теорема 11.3 (**Кукова теорема**)**.** *Проблем* SAT *је* NP-*комплетан.*

11.4. Примери доказа NP-комплетности

У овом одељку доказаћемо да су NP-комплетни следећих пет проблема: покривач грана, доминирајући скуп, 3 SAT, 3-обојивост и проблем клика. Сваки од ових проблема биће детаљније изложен. Доказивање NP-комплетности заснива се на техникама које ће бити резимиране на крају одељка. Да би се доказала NP-комплетност неког проблема, мора се најпре доказати да он припада класи NP, што је обично (али не увек!) лако, а затим треба пронаћи редукцију полиномијалне временске сложености на наш проблем неког проблема за који се зна да је NP-комплетан. Редослед редукција у доказима NP-комплетности пет наведених проблема приказан је на слици 4. Да би се ови докази лакше разумели, наведени су редоследом према тежини, а не према растојању од корена стабла на слици 4; редослед доказа приказан је бројевима уз гране.

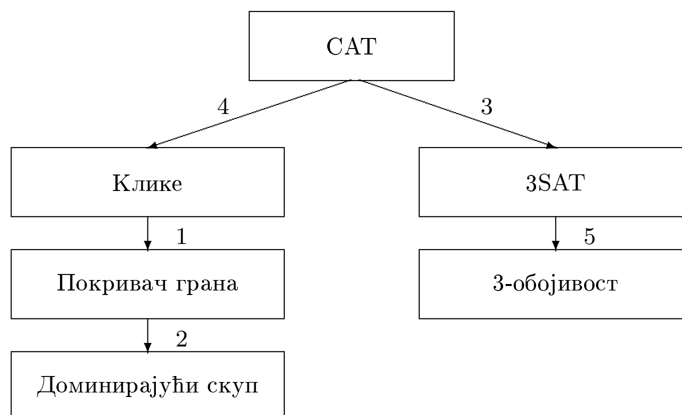


Рис. 4. Редослед доказа NP-комплетности у тексту.

11.4.1. Покривач грана. Нека је $G = (V, E)$ неусмерени граф. **Покривач грана** графа G је такав скуп чворова, да је свака грана G суседна бар једном од чворова из скупа.

Проблем. Задат је неусмерени граф $G = (V, E)$ и природни број k. Установити да ли у G постоји покривач грана са $\leq k$ чворова.

Теорема 11.4. *Проблем покривач грана је* NP-*комплетан.*

Доказательство. Проблем покривач грана је у класи NP, јер се за претпостављени подскуп од $\leq k$ чворова лако проверава за полиномијално време да ли је покривач грана графа. Да бисмо доказали да је проблем покривач грана NP-комплетан, треба да на њега сведемо неки NP-комплетан проблем. У ту сврху искористићемо проблем клика (доказ да је проблем клика NP-комплетан биће дат у одељку 11.4.4). У неусмереном графу $G = (V, E)$ **клика** је такав подграф C графа G у коме су сви чворови међусобно повезани гранама из G. Другим речима, клика је комплетан подграф. Проблем клика гласи: за задати граф G и природни број k, установити да ли G садржи клику величине k. Потребно је трансформисати произвољан улаз за проблем клика у улаз за проблем покривач грана, тако да је решење проблема клика "да" акко је "да" решење одговарајућег проблема покривач грана. Нека је $G = (V, E)$, k произвољан улаз за проблем клика. Нека је $\bar{G} = (V, \bar{E})$ **комплемент графа** G, односно граф са истим скупом чворова као и G, и комплементарним скупом грана у односу на G (односно између произвољна два чвора у $\bar{G}$ грана постоји акко између та два чвора у G *не* постоји грана). Нека је $n = |V|$. Тврдимо да је проблем клика (G, k) сведен на проблем покривач грана графа, са улазом $\bar{G}$, $n - k$ (видети пример на слици 5, где су испрекиданим линијама приказане гране из G). Претпоставимо да је $C = (U, F)$ клика у G. Скуп чворова $V \setminus U$ покрива све гране у $\bar{G}$, јер у $\bar{G}$ нема грана које повезују чворове из U (све те гране су у G). Према томе, $V \setminus U$ је покривач грана за $\bar{G}$. Другим речима, ако G има клику величине k, онда $\bar{G}$ има покривач грана величине $n - k$. Обрнуто, нека је D покривач грана у $\bar{G}$. Тада D покрива све гране у $\bar{G}$, па у $\bar{G}$ не може да постоји грана која повезује нека два чвора из $V \setminus D$. Према томе, $V \setminus D$ је клика у G. Закључујемо да ако у $\bar{G}$ постоји покривач грана величине $n - k$, онда у G постоји клика величине k. Ова редукција се очигледно може извршити за полиномијално време: потребно је само конструисати граф $\bar{G}$ полазећи од графа G и израчунати разлику $n - k$. □

11.4.2. Доминирајући скуп.

Нека је $G = (V, E)$ неусмерени граф. **Доминирајући скуп** је скуп $D \subset V$, такав да је сваки чвор G у D, или је суседан бар једном чвору из D.

Проблем. Дат је неусмерени граф $G = (V, E)$ и природни број k. Установити да ли у G постоји доминирајући скуп са највише k чворова.

Теорема 11.5. *Проблем доминирајући скуп је* NP-*комплетан.*

Доказательство. Проблем доминирајући скуп је у класи NP, јер се за претпостављени подскуп од највише k чворова лако за полиномијално време проверава да ли је доминирајући скуп. Извешћемо редукцију проблема покривач грана на проблем доминирајући скуп. Ако је задат произвољан улаз (G, k)

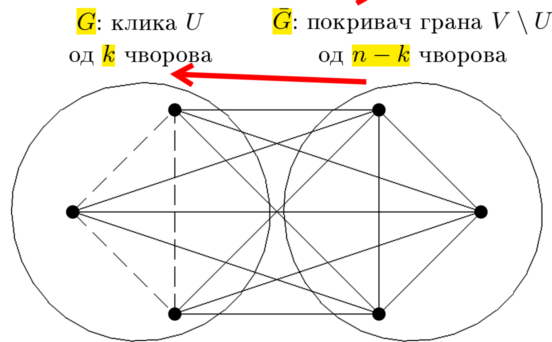


Рис. 5. Редукција проблема клика на проблем покривач грана.

за проблем покривач грана, циљ је конструисати нови граф G' који има доминирајући скуп одређене величине акко G има покривач грана величине највише k. При томе се, без смањења општости, може претпоставити да G нема изолованих чворова (они не утичу на покривач грана, али морају бити укључени у доминирајући скуп). Полазећи од графа G, додајемо му $|E|$ нових чворова и $2|E|$ нових грана на следећи начин. За сваку грану (v, w) из G додајемо нови чвор vw и две нове гране (v, vw) и (w, vw) (слика 6). Другим речима, сваку грану трансформишемо у троугао. Означимо нови граф са G'. Граф G' је лако конструисати за полиномијално време.

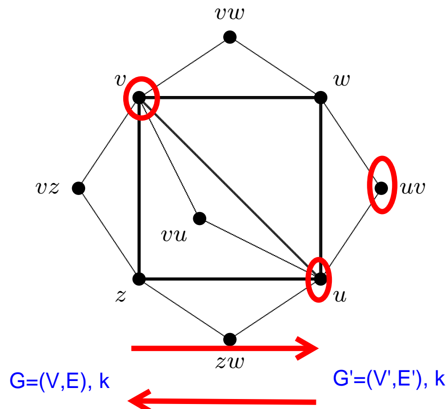


Рис. 6. Редукција проблема покривач грана на проблем доминирајући скуп.

Тврдимо да G има покривач грана величине m акко G' има доминирајући скуп величине m. Нека је D доминирајући скуп графа G'. Ако D садржи било који од нових чворова vw, онда се такав чвор може заменити било чвором v, било чвором w, после чега ће и нови скуп бити доминирајући скуп (v и w покривају све чворове које покрива vw). Према томе, без смањења општости може се претпоставити да D садржи само чворове из G. Међутим, пошто D

"доминира" над свим новим чворовима, он мора за сваку грану из G да садржи бар један њен крај, па је због тога D истовремено и покривач грана графа G. Обрнуто, ако је C покривач грана у G, онда је свака грана G суседна неком чвору из C, па је и сваки нови чвор из G' суседан неком чвору из C. Стари чворови су такође покривени чворовима из C, јер по претпоставци чворови из C покривају све гране. □

11.4.3. 3 SAT. Проблем 3 SAT је упрошћена верзија обичног проблема SAT. Улаз за проблем 3 SAT је КНФ у којој свака клауза има тачно три литерала.

Проблем. Задат је Булов израз у КНФ, у коме свака клауза садржи тачно три литерала. Установити да ли је израз задовољив.

Теорема 11.6. *Проблем* 3 SAT *је* NP-*комплетан.*

Доказательство. Овај проблем је на први поглед лакши од обичног проблема SAT, због допунског ограничења да свака клауза има по три променљиве. Показаћемо да алгоритам који решава 3 SAT може да се искористи да реши обичан проблем SAT (односно да се SAT може свести на 3 SAT). Пре тога, јасно је да 3 SAT припада класи NP. Могу се изабрати ("погодити") вредности променљивих и за полиномијално време проверити да ли је израз тачан. Нека је E произвољан улаз за SAT. Сваку клаузу у E заменићемо са неколико клауза од по тачно три литерала. Нека је $C = (x_1 + x_2 + \cdots + x_k)$ произвољна клауза из E, таква да је $k \geq 4$. Овде је због удобности са x_i означен литерал, односно било променљива, било негација променљиве. Показаћемо како се C може еквивалентно заменити са неколико клауза од по тачно три литерала. Идеја је увести нове променљиве $y_1, y_2, \ldots, y_{k-3}$, које клаузу трансформишу у део улаза за 3 SAT, не мењајући задовољивост израза. За сваку клаузу из E уводе се нове, различите променљиве; C се замењује конјункцијом клауза C', тако да је

$$C' = (x_1+x_2+y_1)(x_3+\bar{y}_1+y_2)(x_4+\bar{y}_2+y_3)\cdots(x_{k-2}+\bar{y}_{k-4}+y_{k-3})(x_{k-1}+x_k+\bar{y}_{k-3}).$$

Тврдимо да је израз, добијен од E заменом C са C', задовољив акко је задовољив израз E. Ако је израз E задовољив, онда бар један од литерала x_i мора имати вредност 1. У том случају се могу изабрати вредности променљивих y_i у C' тако да све клаузе у C' буду тачне. На пример, ако је $x_3 = 1$, онда се може ставити $y_1 = 1$ (што чини тачном прву клаузу), $y_2 = 0$ (друга клауза је тачна због $x_3 = 1$), и $y_i = 0$ за све $i > 2$. Уопште, ако је $x_i = 1$, онда стављамо $y_1 = y_2 = \cdots = y_{i-2} = 1$ и $y_{i-1} = y_i = \cdots = y_{k-3} = 0$, што обезбеђује да буде $C' = 1$. Обрнуто, ако израз C' има вредност 1, тврдимо да бар један од литерала x_i мора имати вредност 1. Заиста, ако би сви литерали x_i имали вредност 0, онда би израз C' имао исту тачност као и израз $C'' = (y_1) \cdot (\bar{y}_1 + y_2) \cdot (\bar{y}_2 + y_3) \cdots (\bar{y}_{k-4} + y_{k-3}) \cdot (\bar{y}_{k-3})$, који очигледно није задовољив (да би било $C'' = 1$, морало би да буде редом $y_1 = 1$, па $y_2 = 1$, итд, $y_{k-3} = 1$, $y_{k-3} = 0$, и $y_{k-3} = 1$, што је контрадикција).

Помоћу ове редукције све клаузе са више од три литерала могу се заменити са неколико клауза од по тачно три литерала. Остаје да се трансформишу клаузе са једним или два литерала. Клауза облика $C = (x_1 + x_2)$ замењује се еквивалентном изразом

$$C' = (x_1 + x_2 + z)(x_1 + x_2 + \bar{z}),$$

где је z нова променљива. Коначно, клауза облика $C = x_1$ може се заменити изразом

$$C' = (x_1 + y + z)(x_1 + \bar{y} + z)(x_1 + y + \bar{z})(x_1 + \bar{y} + \bar{z}),$$

у коме су y и z нове променљиве.

Према томе, произвољни улаз за проблем SAT може се свести на улаз за проблем 3 SAT, тако да је први израз задовољив акко је задовољив други. Јасно је да се ова редукција изводи за полиномијално време. □

11.4.4. Клике. Проблем клика дефинисан је у одељку 11.4.1, у коме је разматран проблем покривач грана.

Проблем. Дат је неусмерени граф $G = (V, E)$ и природни број k. Установити да ли G садржи клику величине бар k.

Теорема 11.7. *Проблем клика је* NP*-комплетан.*

Доказательство. Проблем клика је у класи NP, јер се за сваки претпостављени подскуп од k чворова за полиномијално време може проверити да ли је клика. Показаћемо сада да се проблем SAT може свести на проблем клика. Нека је E произвољни Булов израз у КНФ, $E = E_1 \cdot E_2 \cdots E_m$. Посматрајмо, на пример, клаузу $E_i = (x + y + z + w)$. Њој придружујемо "колону" од четири чвора, означена литералима из E_i (без обзира што се неки од њих можда појављују и у другим клаузама). Другим речима, граф G који конструишемо имаће по један чвор за сваку појаву било које променљиве. Остаје питање како повезати ове чворове, тако да G садржи клику величине бар k акко је израз E задовољив. Приметимо да се вредност k може изабрати произвољно, јер је потребно свести проблем SAT на проблем клика, односно решити проблем SAT користећи алгоритам за решавање проблема клика. Наравно, алгоритам за решавање проблема клика мора да ради за сваку вредност k. Ово је важна флексибилност, која се често користи у доказима NP-комплетности. У овом случају за k ћемо изабрати вредност једнаку броју клауза m.

Гране у графу G могу се задати на следећи начин. Чворови из исте колоне (односно чворови придружени литералима из исте клаузе) не повезују се гранама. Чворови из различитих колона су скоро увек повезани: изузетак је случај два чвора од којих један одговара променљивој, а други комплементу те исте променљиве. Пример графа који одговара изразу

$$E = (x + y + \bar{z}) \cdot (\bar{x} + \bar{y} + z) \cdot (y + \bar{z})$$

приказан је на слици 7. Јасно је да се G може конструисати за полиномијално време.

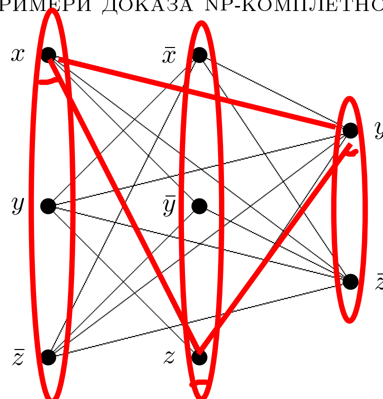


Рис. 7. Редукција проблема SAT са улазом $(x + y + \bar{z}) \cdot (\bar{x} + \bar{y} + z) \cdot (y + \bar{z})$ на проблем клика.

Тврдимо да G има клику величине бар m, акко је израз E задовољив. Најпре запажамо да због конструкције максимална клика не може имати више од m чворова, независно од E. Претпоставимо да је израз E задовољив. Тада постоји такво додељивање вредности променљивим, да у свакој клаузи постоји бар један литерал са вредношћу 1. Чвор који одговара том литералу прикључује се клики (ако има више таквих литерала, бира се произвољан од њих). Добијени подграф јесте клика, јер једини начин да два чвора из различитих колона не буду повезана је да један од њих буде комплемент другог — што је немогуће, јер је свим изабраним литералима додељена вредност 1. Обрнуто, претпоставимо да G садржи клику величине бар m. Клика мора да садржи тачно један чвор из сваке колоне (јер по конструкцији чворови из исте колоне нису повезани). Одговарајућим литералима додељујемо вредност 1. Ако на овај начин некој променљивој није додељена вредност, то се може учинити на произвољан начин. Изведено додељивање вредности променљивима је непротивречно: ако би некој променљивој x и њеном комплементу $\bar{x}$, укљученим у клику, истовремено била додељена вредност 1, то би значило да одговарајући чворови (по конструкцији) нису повезани — супротно претпоставци да су оба чвора у клики. □

11.4.5. 3-обојивост. Нека је $G = (V, E)$ неусмерени граф. **Исправно бојење** (или само бојење) графа G је такво придруживање боја чворовима, да је сваком чвору придружена нека боја, а да су суседним чворовима увек придружене различите боје.

Проблем (3-обојивост). Дат је неусмерени граф $G = (V, E)$. Установити да ли се G може обојити са три боје.

Теорема 11.8. *Проблем 3-обојивост је* NP-*комплетан.*

Доказательство. Проблем 3-обојивост припада класи NP, јер се може претпоставити произвољно бојење графа са 3 боје, а затим за полиномијално

време проверити да ли је претпостављено бојење исправно. Извешћемо редукцију проблема 3 SAT на проблем 3-обојивост. Доказ је нешто компликованији из два разлога. Најпре, проблеми се односе на различите објекте — Булове изразе у КНФ, односно графове. Друго, не може се просто заменити један објекат (на пример чвор или грана) другим (на пример клаузом); мора се водити рачуна о комплетној структури. Идеја је да се искористе саставни елементи, који се повезују у целину. Нека је E произвољан улаз за проблем 3 SAT. Треба конструисати граф G, тако да је израз E задовољив акко је G 3-обојив. Најпре конструишемо *основни троугао* M. Пошто је M троугао (комплетни граф са три чвора), за његово бојење потребне су тачно три боје. Означимо те боје са T (тачно), F (нетачно) и A, видети доњи троугао на слици 8. Поред тога, за сваку променљиву x додајемо нови троугао M_x, чије чворове означавамо са x, $\bar{x}$ и A, при чему се чвор означен са A поклапа са чвором из M са истом ознаком. Према томе, ако се у изразу појављује k променљивих, имамо $k+1$ троуглова са заједничким чвором A (слика 8). Ово обезбеђује да, ако је, на пример, чвор x обојен бојом T, онда чвор $\bar{x}$ мора бити обојен бојом F (јер су оба чвора суседна чвору обојеном бојом A), и обрнуто. Ово је у складу са значењем $\bar{x}$.

Рис. 8. Први део конструкције за редукцију 3 SAT на 3-обојивост графа.

Потребно је додати услов, који би обезбеђивао да у свакој клаузи бар један литерал има вредност 1. То се може обезбедити на следећи начин. Претпоставимо, на пример, да се ради о клаузи $(x+y+z)$. Овде се x, y, z могу сматрати литералима, тј. недостатак негација над симболима x, y, z не смањује општост разматрања. За ову клаузу уводимо шест нових чворова и повезујемо их са постојећим чворовима на начин приказан на слици 9. Назовимо три нова чвора повезана са T и x, y или z *спољашњим чворовима* (означени су са O на слици), а три нова чвора у троуглу — *унутрашњим чворовима* (означени су са I на слици). Тврдимо да ова конструкција обезбеђује (ако је могуће бојење са три боје) да бар један од чворова x, y или z мора бити обојен бојом T. Ни један од чворова x, y, z не може бити обојен бојом A, јер су сви они повезани са чвором A (видети слику 8). Ако би сва три чвора x, y, z били обојени бојом F, онда би три нова спољашња чвора повезана са њима морали бити обојени

бојом A, па се унутрашњи троугао не би могао обојити са три боје! Комплетан граф који одговара изразу $(\bar{x}+y+\bar{z})\cdot(\bar{x}+\bar{y}+z)$ приказан је на слици 10.

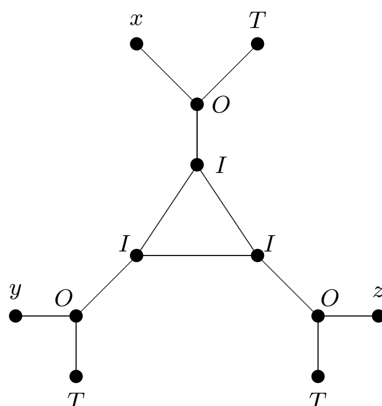


Рис. 9. Подграфови који одговарају клаузама у редукцији 3 SAT на 3-обојивост.

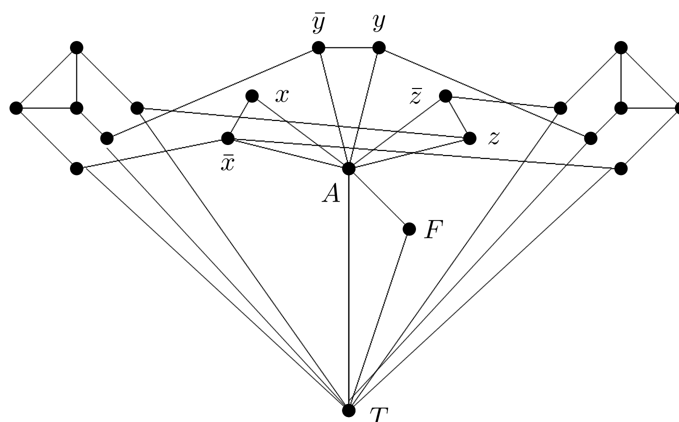


Рис. 10. Граф који одговара изразу $(\bar{x}+y+\bar{z})\cdot(\bar{x}+\bar{y}+z)$ у редукцији 3 SAT на 3-обојивост.

Сада можемо да комплетирамо доказ, и то у два смера: (1) ако је израз E задовољив, G се може обојити са три боје, и (2) ако се G може обојити са три боје, онда је израз E задовољив. Ако је израз E задовољив, онда постоји такво додељивање вредности променљивим, да у свакој клаузи бар један литерал има вредност 1. Обојимо чворове графа у складу са њиховим вредностима (са T ако је вредност 1, односно са F у противном). Троугао M обојен је бојама T, F и A на описани начин. У подграфу који одговара клаузи бар један литерал има вредност 1; одговарајући спољашњи чвор бојимо са F, а остале спољашње

чворове са A, после чега је бојење унутрашњих чворова лако извести. Према томе, G се може обојити са три боје. Обрнуто, ако се G може обојити са три боје, назовимо боје у складу са бојењем троугла M (који мора бити обојен са три боје). Због троуглова на слици 8, боје променљивих омогућују непротивречно додељивање вредности променљивим. Конструкција са слике 9 обезбеђује да је бар један литерал у свакој клаузи обојен са T. Коначно, јасно је да се граф G може конструисати за полиномијално време, чиме је доказ завршен. □

11.4.6. Општа запажања. Размотрићемо укратко неколико општих метода за доказивање NP-комплетности неког проблема Q. Први услов — да Q припада класи NP — обично се лако доказује (не увек). После тога треба изабрати неки проблем за који се зна да је NP-комплетан, а за који изгледа да је повезан, или сличан са Q. Тешко је дефинисати ову "сличност", јер понекад изабрани проблем и Q изгледају јако различито (на пример, проблем клика и SAT). Избор правог проблема који ће бити сведен на Q је понекад изузетно тежак, и захтева велико искуство. Мора се покушати са неколико редукција, док се не дође до погодног проблема.

Важно је да се изведе редукција на Q, полазећи од *произвољног улаза* познатог NP-комплетног проблема. Најчешћа грешка у оваквим доказима је извођење редукције у обрнутом смеру. Прави смер обезбеђује нпр. следеће правило: треба обезбедити да се познати NP-комплетан проблем може решити црном кутијом, која извршава алгоритам за решавање Q. То можда мало противречи интуицији. Природно би било, покушати са решавањем проблема Q, пошто је то задати проблем. Овде пак покушавамо да решимо други проблем (познати NP-комплетан проблем), користећи решење проблема Q. Ми и не покушавамо да решимо Q!

Постоји више "степени слободе" који се могу користити при редукцији. На пример, ако Q садржи неки параметар, онда се његова вредност може фиксирати на произвољан начин (супротно од параметара у проблему који се своди на Q, који се не смеју фиксирати!). Пошто је Q само алат за решавање познатог NP-комплетног проблема, може се искористити на произвољан начин. Поред фиксирања параметара, могу се користити рестрикције Q на специјалне случајеве добијене и на друге начине. На пример, могу се користити само неки типови улаза за Q (ако се ради о графовима — бипартитни графови, стабла и слично). Други важан извор флексибилности је чињеница да је ефикасност редукције небитна — довољно је да се редукција може извести за полиномијално време. Могу се игнорисати не само константе, тако што би се, на пример, дуплирала величина проблема: исто тако може се и квадрирати величина проблема! Може се увести полиномијално много нових променљивих, може се заменити сваки чвор у графу новим великим графом и слично. Не постоји потреба за ефикасношћу (све док се остаје у границама полиномијалног), јер сврха редукције није да се трансформише у алгоритам.

Постоје неке уобичајене технике за добијање редукција. Најједноставнија је доказати да је познати NP-комплетан проблем специјалан случај проблема

специјални случај

покривање скупова → покривач грана

Q. Ако је тако, доказ је директан, јер је решавање Q истовремено и решавање познатог NP-комплетног проблема. Посматрајмо, на пример, проблем **покривање скупова** покривање скупова. Улаз је колекција подскупова $S_1, S_2, \ldots, S_n$ задатог скупа U и природни број k. Проблем је установити да ли постоји подскуп $W \subseteq U$ са највише k елемената, који садржи бар по један елеменат сваког од подскупова S_i. Запажамо да је проблем покривач грана специјални случај проблема покривања скупова у коме U одговара скупу чворова V, а сваки скуп S_i одговара једној грани и садржи два чвора којима је та грана суседна. Према томе, ако знамо да решимо проблем покривања скупова за произвољне скупове, онда можемо да решимо и проблем покривач грана.

Морамо, међутим, бити пажљиви кад користимо овај приступ. У општем случају није тачно да додавање нових захтева чини проблем тежим. Посматрајмо проблем покривач грана. Претпоставимо да смо додали ограничење да покривач грана не сме да садржи два суседна чвора. Другим речима, тражимо мали скуп чворова који је истовремено и покривач грана и независан скуп (независан скуп је подскуп чворова графа, такав да између његова два произвољна елемента не постоји грана). Овај проблем је на први поглед тежи и од проблема покривач грана и од проблема независан скуп, јер треба бринути о више захтева. Испоставља се, међутим, да је ово лакши проблем, и да се може решити за полиномијално време (видети задатак 6.65). Разлог овој појави је у томе што допунски захтеви толико сужавају фамилију подскупова кандидата, да се минимум лако налази.

Друга релативно једноставна техника користи **локалне редукције**. Објекат, улаз за један проблем, пресликава се у објекат који је улаз за други проблем. Пресликавање се изводи на локални начин, независно од осталих објеката. Доказ NP-комплетности проблема доминирајућег скупа изведен је на овај начин. Свака грана у једном графу замењена је троуглом у другом графу. Потешкоћа са овом техником је како на најбољи начин дефинисати одговарајуће објекте.

Најкомпликованија техника је употреба саставних елемената – блокова, као што је то учињено при доказу NP-комплетности проблема 3-обојивост. Блокови обично зависе један од другог, па је њихово независно пројектовање неизводљиво. Морају се имати на уму сви циљеви проблема, да би се могло координирати пројектовање различитих блокова.

11.4.7. Још неки NP-комплетни проблеми. Следи списак садржи још неколико NP-комплетних проблема, који могу бити корисни као полазна основа за нове редукције. Проналажење правог проблема за редукцију обично је више од половине доказа NP-комплетности.

Хамилтонов циклус: Хамилтонов циклус у графу је прост циклус који садржи сваки чвор графа тачно једном. Проблем је установити да ли задати граф садржи Хамилтонов циклус. Проблем је NP-комплетан и за неусмерене и за усмерене графове. (Редукција проблема покривач грана.)

Хамилтонов пут: Хамилтонов пут у графу је прости пут који садржи сваки чвор графа тачно једном. Проблем је установити да ли задати граф

садржи Хамилтонов пут. Проблем је NP-комплетан и за неусмерене и усмерене графове. (Редукција проблема покривач грана.)

Проблем трговачког путника: Нека је задат тежински комплетан граф G и број W. Установити да ли у G постоји Хамилтонов циклус са збиром тежина грана $\leq W$. (Директна редукција проблема Хамилтонов циклус.)

Независан скуп: Независан скуп у графу је подскуп чворова графа, такав да између његова два произвољна елемента не постоји грана. Ако је задат граф G и природни број k, установити да ли G садржи независни скуп са бар k чворова. (Директна редукција проблема клика.)

3-димензионално упаривање: Нека су X, Y и Z дисјунктни скупови од по k елемената, и нека је M задати скуп тројки (x, y, z) таквих да је $x \in X$, $y \in Y$ и $z \in Z$. Проблем је установити да ли постоји такав подскуп скупа M који сваки елеменат садржи тачно једном. Одговарајући дводимензионални проблем упаривања је обичан проблем бипартитног упаривања. (Редукција проблема 3SAT.)

Партиција: Улаз је скуп X чијем је сваком елементу x придружена његова величина $s(x)$. Проблем је установити да ли је могуће поделити скуп на два дисјунктна подскупа са једнаким сумама величина. (Редукција проблема 3-димензионално упаривање.)

Приметимо да се овај и следећи проблем могу ефикасно решити алгоритмом *Ranac* из одељка 4.10, ако су величине мали цели бројеви. Међутим, пошто је величина улаза сразмерна броју бита потребних да се представи улаз, овакви алгоритми (који се зову **псеудополиномијални алгоритми**) су уствари експоненцијални у односу на величину улаза.

Проблем ранца: Улаз су два броја S, V и скуп X чијем је сваком елементу x придружена величина $s(x)$ и вредност $v(x)$. Проблем је установити да ли постоји подскуп $B \subseteq X$ са укупном величином $\leq S$ и укупном вредношћу $\geq V$. (Редукција проблема партиције.)

Проблем паковања: Улаз је низ бројева $a_1, a_2, \ldots, a_n$ и два броја b, k. Проблем је установити да ли се овај скуп може разложити у k подскупова, тако да је сума бројева у сваком подскупу $\leq b$. (Редукција проблема партиције.)

11.5. Технике за рад са NP-комплетним проблемима

Појам NP-комплетности је основа за елегантну теорију која омогућује препознавање проблема за које највероватније не постоји полиномијални алгоритам. Међутим, доказивањем да је проблем NP-комплетан, сам проблем није елиминисан! И даље је потребно решити га. Технике за решавање NP-комплетних проблема су понекад другачије од техника које смо до сада разматрали. Ни један NP-комплетан проблем се (највероватније) не може решити тачно и комплетно алгоритмом полиномијалне временске сложености. Због тога смо принуђени на компромисе. Најчешћи компромиси односе се на оптималност, гарантовану ефикасност, или комплетност решења. Постоје и друге алтернативе, од којих свака понешто жртвује. Исти алгоритам се може користити у различитим ситуацијама, примењујући различите компромисе.

Алгоритам који не даје увек оптималан (или тачан) резултату зове се **приближан алгоритам**. Посебно су занимљиви приближни алгоритми који могу да гарантују границу за степен одступања од тачног решења. Нешто касније видећемо три примера таквих алгоритама.

У одељку 5.8 разматрали смо пробабилистичке алгоритме који могу да погреше. Најпознатији у тој класи је алгоритам за препознавање простих бројева. За проблем препознавања простих бројева се не зна да ли је у P, али се сматра да није NP-комплетан. Овде се нећемо бавити алгоритмима за препознавање простих бројева, јер захтевају предзнање из теорије бројева. Раширено је веровање да се NP-комплетни проблеми не могу решити алгоритмом полиномијалне временске сложености, код којих је вероватноћа грешке мала за све улазе. Такви алгоритми су по свему судећи ефикасни за проблеме, за које се не зна да ли су у класи P, али се не верује да су NP-комплетни. Такви проблеми нису чести. Пробабилистички алгоритми се могу користити као део других стратегија — на пример, као део приближних алгоритама.

Други компромис могућ је у вези са захтевом да полиномијално буде време извршавања *у најгорем случају*. Може се покушати са решавањем NP-комплетних проблема за полиномијално *средње време*. Потешкоћа са овим приступом је како дефинисати *просечно време*. На пример, тешко је искључити улазе за које је конкретан проблем тривијалан (као што је граф који има само изоловане чворове) из израчунавања просека. Такви тривијални улази могу знатно да утичу на просек. Алгоритми предвиђени за поједине типове случајних улаза могу да буду корисни ако је стварна расподела вероватноћа улаза у складу са претпостављеном. Међутим, обично је налажење тачне расподеле је обично врло тешко. Најтежи део посла при конструкцији алгоритама, који у просеку добро раде, је најчешће њихова анализа.

Коначно, могу се правити компромиси у вези са комплетношћу алгоритама; наиме, може се дозволити да алгоритам ради ефикасно само за неке специјалне улазе. На пример, проблем покривач грана може се решити за полиномијално време за бипартитне графове. Према томе, кад се формулише апстрактни проблем полазећи од ситуације из реалног живота, треба обезбедити да сви допунски услови које улаз задовољава буду укључени у апстрактну дефиницију. Други пример су алгоритми са експоненцијалном временском сложеношћу, који се ипак могу извршавати за мале улазе, што је често потпуно задовољавајуће.

У овом одељку описаћемо више оваквих техника и илустровати их примерима. Започињемо са две опште и корисне технике које се зову **претрага** и **гранање са одсецањем**. Ове технике су сличне. Могу се користити као основа за приближни алгоритам, или као тачан алгоритам за мале улазе. На крају наводимо неколико примера приближних алгоритама.

11.5.1. Претрага и гранање са одсецањем. Ове технике описаћемо на једном примеру. Размотримо проблем 3-бојење, односно додељивања боја чворовима графа уз извесна ограничења. Пример је из класе проблема налажења оптималне *вредности* (броја боја у овом случају) за n параметара (овде боја

чворова). У случају проблема 3-бојење сваки параметар има три могуће вредности, што одговара трима бојама. Према томе, број потенцијалних решења је 3^n, јер је то укупан број начина на које се n чворова могу обојити са три боје. Наравно, сем ако граф нема грана, број *исправних* бојења може да буде знатно мањи од 3^n, јер гране намећу ограничења на могућа бојења. Да би се испитали сви могући начини бојења чворова, може се започети додељивањем произвољне боје једном од чворова, а затим наставити са бојењем осталих чворова, водећи рачуна о ограничењима које намећу гране — да суседни чворови морају бити обојени различитим бојама. При бојењу чвора, покушава се са свим могућим бојама, које су конзистентне са претходно обојеним чворовима. Овај процес може се обавити алгоритмом обиласка *стабла*, који представља суштину претраге и гранања са одсецањем. Да бисмо разликовали чворове графа и стабла, чворове стабла зваћемо *теменима*.

Корен стабла одговара почетном стању проблема, а свака грана одговара некој одлуци о вредности неког параметра. Означимо три боје са Ц(рвено), П(лаво) и З(елено). На почетку се могу изабрати нека два суседна чвора v и w, и обојити рецимо са П и З. Пошто се они ионако морају обојити различитим бојама, није битно које ће боје бити изабране (коначно бојење се увек може испермутовати), па се зато може започети са бојењем два уместо једног чвора. Бојење ова два чвора одговара почетном стању проблема, које је придружено корену. Стабло се конструише у току самог обиласка. У сваком темену t стабла бира се следећи чвор графа за бојење, и додаје један, два или три сина темену t, зависно од броја боја којима се може обојити чвор u. На пример, ако је u први изабрани чвор (после v и w), а u је суседан чвору w (који је већ обојен бојом З), онда постоје две могућности за бојење u, П и Ц, па се корену додају два одговарајућа сина. Затим се бира један од ова два сина и процес се наставља. Кад се један чвор обоји, смањује се број могућности за бојење осталих чворова; према томе, број синова показује тенденцију опадања како се напредује у дубину стабла.

У случају да су обојени сви чворови графа, проблем је решен. Вероватније је, међутим, да ћемо наићи на чвор који се не може обојити (јер има три суседна чвора који су већ обојени различитим бојама). У том тренутку чинимо **корак назад** — враћамо се уз стабло (ка нижим нивоима) и покушавамо са другим синовима. Пример графа, и стабла које одговара решавању проблема 3-бојење на њему, приказан је на слици 11. Приметимо да у овом случају, кад се фиксирају боје чворова 1 и 2, за бојење осталих чворова постоји само један начин, до кога се долази крајњим десним путем кроз стабло на слици.

Алгоритам за обилазак стабла може се формулисати индукцијом. Потребно је појачати индуктивну хипотезу, тако да обухвати бојење графова чији су неки чворови већ обојени. Другим речима, индуктивна хипотеза треба да обухвати бојење не само необојених графова, него и довршавање започетог бојења.

Индуктивна хипотеза. Умемо да завршимо 3-бојење графа који има мање од k необојених чворова, или да за такве графове установимо да се не могу обојити са три боје.

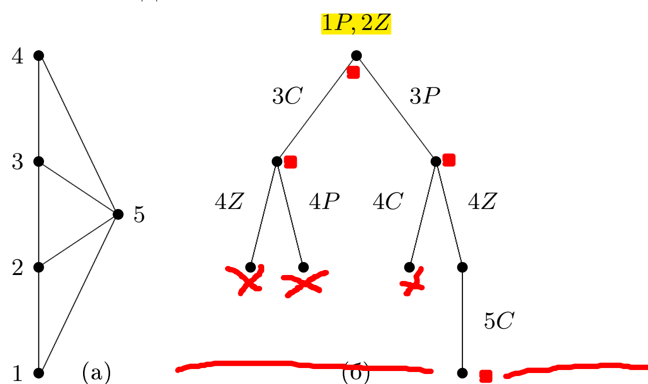
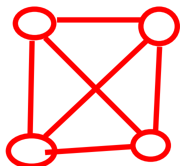


Рис. 11. Пример примене претраге на 3-бојење графа.

Ако је дат граф са k необојених чворова, бирамо један од необојених чворова и проналазимо све могуће боје којима се он може обојити. Ако су све боје већ искоришћене за суседе тог чвора, започето 3-бојење се не може комплетирати. У противном, чвор бојимо једном од могућих боја (једном по једном) и решавамо преостале проблеме (који имају по $k-1$ необојени чвор) индукцијом. Алгоритам је приказан на слици 12.



Алгоритам 3 − bojenje(G, **var** U);
Улаз: $G = (V, E)$ (неусмерени граф) и U (скуп већ обојених чворова).
{U је обично на почетку празан скуп}
Излаз: Придруживање једне од три боја сваком чвору G.
begin
  if $U = V$ **then** *print* "бојење је завршено"; *halt*
    {прекид свих рекурзивних позива}
  else
    изабрати неки чвор v који није у скупу U;
    for $C := 1$ **to** 3 **do**
      if ни један сусед v није обојен бојом C **then**
        додај чвор v обојен бојом C скупу U;
        $3 - bojenje(G, U)$
end

Рис. 12. Алгоритам $3 - bojenje$.

Није тешко наћи пример графа и редослед његових чворова, такав да се при његовом бојењу добија стабло са експоненцијалним бројем темена (видети задатак 11.11). Ово је честа ситуација у алгоритмима претраге. Једино се можемо надати да ћемо, ако стабло будемо обилазили на погодан начин, на решење брзо наићи. Алгоритам који смо описали не прецизира начин избора

следећег чвора. Пошто се као следећи може изабрати произвољан чвор, имамо извесну флексибилност, која се може искористити за формирање одговарајуће хеуристике. Нешто касније вратићемо се на ово питање.

Гранање са одсецањем је варијација претраге, која се може применити ако се тражи минимум (или максимум) неке циљне функције. Посматрајмо општи проблем бојења графа: циљ је пронаћи најмањи број боја потребан за бојење задатог графа, а не само установити да ли је граф могуће обојити са три боје. Може се формирати стабло слично као за 3-бојење, али број синова сваког чвора може да буде врло велики. Сваки нови чвор може се обојити неком од већ коришћених боја (сем ако је неки од његових суседа већ обојен том бојом), или новом бојом. Према томе, алгоритам за 3-бојење мења се на два места: (1) константа 3 замењује се бројем до сада коришћених боја, и (2) алгоритам се не завршава у тренутку кад је $V = U$, јер је могуће да постоји бољи начин да се обоји граф.

Потешкоће изазива чињеница да у алгоритму долази до враћања уназад само кад се дође до листа у стаблу (односно кад је $V = U$), јер се нова боја увек може доделити чвору. Према томе, скоро је сигурно да ће овакав алгоритам бити врло неефикасан (сем ако је граф врло густ). Ефикасност алгоритма може се побољшати помоћу следећег запажања, које је у основи метода гранања са одсецањем. Претпоставимо да смо прошли део стабла до неког листа и тако пронашли исправно бојење са k боја. Претпоставимо даље да смо после враћања уназад кренули другим путем, и тако дошли до чвора који захтева увођење $(k + 1)$-е боје. У том тренутку може се направити корак назад, јер је већ познато боље решење. Према томе, k служи као **граница** за претрагу. У сваком темену стабла израчунавамо доњу границу за најбоље решење на које се може наићи међу следбеницима тог темена. Ако је та доња граница већа од неког већ нађеног решења, чинимо корак назад. Један од начина да се алгоритам гранања са одсецањем учини ефикасним је израчунавање добрих доњих граница (или горњих граница, ако је циљ максимизирати циљну функцију). Други важан елеменат је налажење доброг редоследа обиласка, који омогућује брзо проналажење добрих решења, а тиме и раније напуштање неперспективних подстабала.

Илустроваћемо ову идеју проблемом целобројног линеарног програмирања, споменутог у одељку 10.3. Проблем је сличан линеарном програмирању, али има допунско ограничење да променљиве могу узимати само целобројне вредности. Нека је $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ вектор колона променљивих, нека су $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_k$ вектори врсте реалних бројева једнаке дужине n, и нека су $b_1, b_2, \ldots, b_k$, $c_1, c_2, \ldots, c_n$ реални бројеви. Проблем је максимизирати вредност **линеарне циљне функције**

$$z = \sum_{i=1}^{n} c_i x_i$$

под условом да су координате вектора $\mathbf{x}$ цели бројеви и

$$\mathbf{a}_j \cdot \mathbf{x} \leq b_j, \quad j = 1, 2, \ldots, k,$$

при чему су сви симболи сем **x** константе. Многи NP-комплетни проблеми могу се лако формулисати као проблеми целобројног линеарног програмирања (у наставку ћемо видети један такав пример). Према томе, целобројно линеарно програмирање је NP-тежак проблем. Прецизније, проблем је NP-комплетан, али је доказ да је тај проблем у класи NP нешто компликованији.

Показаћемо сада како се проблем клика може формулисати као проблем целобројног линеарног програмирања При томе разматрамо варијанту у којој се тражи максимална клика, уместо да се проверава да ли постоји клика задате величине. Уводимо n променљивих $x_1, x_2, \dots, x_n$, које одговарају чворовима, тако да је $x_i = 1$ ако v_i припада максималној клики, односно $x_i = 0$ у противном. Циљна функција је

$$z = x_1 + x_2 + \cdots + x_n,$$

јер је потребно укључити у клику што већи број чворова. Постоји по једно ограничење за сваки чвор

$$0 \le x_i \le 1, \quad i = 1, 2, \dots, n,$$

и по једно ограничење за сваки пар несуседних чворова

$$x_i + x_j \le 1 \text{ за сваки пар чворова } v_i, v_j \text{ таквих да } (v_i, v_j) \notin E.$$

Први скуп ограничења ограничава вредности променљивих на 0 или 1. Други скуп ограничења обезбеђује да се два несуседна чвора не могу истовремено изабрати у клику; према томе, чворови који су изабрани чине клику.

Овакав проблем целобројног линеарног програмирања може се решити алгоритмом гранања са одсецањем, коришћењем одговарајућег линеарног програма (који решава исти проблем, али без ограничења целобројности) за израчунавање граница. Решење линеарног програма може се састојати само од целих бројева; у том случају полазни проблем је решен. Међутим, вероватније је да ће у решењу неке променљиве имати нецелобројне вредности. Претпоставимо, на пример, да је решење линеарног програма придруженог проблему клика $(0.1, 1, \dots, 0.5)$ и $z = 7.8$. Пошто линеарни програм максимизира циљну функцију са мање ограничења од целобројног линеарног програма, максимум који он пронађе је горња граница за максимум који може да пронађе целобројни линеарни програм. Према томе, не може се очекивати проналажење клике величине веће од 7. Ова информација може бити корисна приликом претраге. Као и код обичне претраге, бирамо неке вредности параметара и напредујемо дуж стабла, при чему теме ниже у стаблу одговара потпроблему полазног проблема. На пример, потпроблем може да одговара прикључивању чворова v, w клики, и елиминисању из ње чворова u, x, тј. покушава се са налажењем највеће клике са v, w, а без u, x. Ако се том приликом добије решење линеарног програма које је *мање* од величине већ пронађене клике, онда чинимо корак назад, и напуштамо ту варијанту. То је суштина метода гранања и одсецања. Покушавамо да нађемо горње границе (или доње, ако се циљна функција минимизира) које ће омогућити одсецање неперспективних подстабала на што нижем нивоу (што ближе корену стабла).

Резултат линеарног програма може се искористити и при избору редоследа претраге. На пример, ако је $x_2 = 1$ у нецелобројном решењу, може се претпоставити да је $x_2 = 1$ и у целобројном решењу. Та претпоставка не мора бити тачна, али је пример врсте хеуристика које тражимо. Покушавамо да повећамо вероватноћу брзог налажења оптималног решења; при томе је јасно да не могу све такве одлуке да буду "исправне", јер је проблем NP-комплетан. Можемо да ставимо $x_2 = 1$, изменимо у складу са тим ограничења (нпр. променимо вредности променљивих за све чворове несуседне са v_2 на 0), и решимо резултујући линеарни програм. Ако у неком тренутку модификовани линеарни програм има максималну вредност $z = a$, при чему је a мање од величине највеће пронађене клике, та грана се може напустити.

Према томе, линеарни програм користи се на два начина: за добијање горњих граница и тиме за напуштање неперспективиних подстабала (одсецање), односно за доношења одлука о усмеравању претраге. Очекује се да решавање потпроблема који "највише обећава", учини непотребним решавање великог дела других потпроблема. Учестаност одсецања, односно еифкасност целог алгоритма, зависи од хеуристике за формирање потпроблема и избора наредног потпроблема за испитивање. Хеуристика зависи од конкретног проблема, и у овој области спроводе се широка истраживања.

Алгоритми са гранањем и одсецањем гарантују проналажење оптималног решења ако се сви потпроблеми истраже или "одсеку". Ако извршавање траје предуго, може се прекинути, и тако добити апроксимацију — најбоље решење пронађено до тог тренутка. Обилазак стабла може се извести претрагом у дубину, претрагом у ширину или неком њиховом комбинацијом. Екстремни пример ранијег завршавања претраге је узимање првог пута (изабраног у складу са неком хеуристиком) која води прихватљивом решењу (обично оно које одговара листу стабла) као излаза алгоритма. На пример, у алгоритму за бојење графа, чворови се могу бојити редоследом према растућим степенима; идеја је да се мање губи на флексибилности фиксирањем боје чворова малог степена. То је једноставан похлепни алгоритам.

11.5.2. Приближни алгоритми са гарантованим квалитетом решења. У овом одељку размотрићемо приближне алгоритме за три NP-комплетна алгоритма: покривач грана, паковање и еуклидски проблем трговачког путника. Сва три алгоритма имају гарантовани квалитет решења; другим речима, може се доказати да решења која они дају нису предалеко од оптималних решења.

11.5.2.1. *Покривач грана.* Најпре ћемо размотрити једноставан приближни алгоритам за налажење покривача грана датог графа. Алгоритам гарантује налажење покривача са највише два пута више чворова него у минималном покривачу. Нека је G граф, и нека је M максимално упаривање у G. Пошто је M упаривање, његове гране немају заједничких тачака, а пошто је M максимално упаривање, све остале гране имају бар један заједнички чвор са неком граном из M.

uparivanje - skup disjunktnih grana

Теорема 11.9. *Скуп чворова суседних гранама максималног упаривања M је покривач грана, са највише два пута више чворова него што их има минимални покривач.*

Доказательство. Скуп чворова који припадају M чини покривач грана, јер је M максимално упаривање. Сваки покривач грана мора да покрије све гране — специјално, све гране упаривања M. Пошто је M упаривање, било који чвор из M може да покрије највише једну грану упаривања M. Према томе, бар половина чворова из M мора да припада покривачу грана. □

Максимално упаривање може се наћи простим додавањем грана све дотле док је то могуће (односно док постоје нека два неупарена, а суседна чвора).

11.5.2.2. *Једнодимензионално паковање*. **Једнодимензионални проблем паковања** односи се на паковање објеката различите величине у *кутије* тако да се искористи најмањи могући број кутија. На пример, приликом селидбе је циљ пренети све ствари, користећи камион најмањи могући број пута, пакујући ствари што је могуће боље. То је наравно тродимензионални проблем; овде ћемо се позабавити његовом једнодимензионалном верзијом. Због једноставности се претпоставља да сви сандуци имају величину 1.

Проблем. Нека је $x_1, x_2, \ldots, x_n$ скуп реалних бројева између 0 и 1. Поделити их у најмањи могући број подскупова, тако да сума бројева у сваком подскупу буде највише 1.

На једнодимензионални проблем паковања наилази се, на пример, у проблемима управљања меморијом, кад постоје захтеви за доделом меморијских блокова различите величине, а додељивање се врши из неколико једнаких великих блокова меморије. Једнодимензионални проблем паковања је NP-комплетан.

Једна од могућих хеуристика за решавање овог проблема је ставити x_1 у прву кутију, а затим, за свако i, ставити x_i у прву кутију у којој има довољно места, или започети са новом кутијом, ако нема довољно места ни у једној од коришћених кутија. Овај алгоритам зове се **први одговарајући** и, као што показује следећа теорема, довољно је добар у најгорем случају.

Теорема 11.10. *Алгоритам први одговарајући захтева највише $2m$ кутија, где је m најмањи могући број кутија.*

Доказательство. По завршетку алгоритма први одговарајући не постоје две кутије са искоришћењем мањим од $1/2$. Према томе, ако са k означимо број употребљених кутија, биће $m \geq \sum_{i=1}^{n} x_i > (k-1)/2$, одакле је $k < 2m+1$, односно $k \leq 2m$. 2m > k-1 □

Испоставља се да је граница дефинисана овом теоремом прилично груба. Константа 2 из теореме може се смањити на 1.7 после нешто компликованије анализе. Константа 1.7 не може се даље смањити, јер постоје примери у којима алгоритам први одговарајући захтева 1.7 пута више кутија од оптималног алгоритма.

Описани алгоритам може се једноставно побољшати. До најгорег случаја долази кад се много малих бројева појављује на почетку. Уместо да се бројеви пакују у редом којим наилазе, они се најпре сортирају у нерастући низ. Промењени алгоритам зове се **опадајући први одговарајући**, и у најгорем случају троши највише око 1.22 пута више кутија од оптималног алгоритма (теорему дајемо без доказа).

Теорема 11.11. *Алгоритам опадајући први одговарајући захтева највише* $\frac{11}{9}m + 4$ *кутија, где је* m *најмањи могући број кутија.*

Константа 11/9 је такође најбоља могућа. Први одговарајући и опадајући први одговарајући су једноставне хеуристике. Постоје други методи који гарантују још мање константе. Њихова анализа је у већини случајева компликована.

Стратегије које смо описали типичне су за хеуристичке алгоритме. Оне одражавају природне приступе, односно одговарају начину на који би неко ручно решавао ове проблеме. Међутим, видели смо много случајева у којима се директни приступи понашају лоше за велике улазе. Због тога је веома важно анализирати понашање таквих алгоритама.

11.5.2.3. *Еуклидски проблем трговачког путника*. **Проблем трговачког путника** (TSP, скраћеница од traveling salesman problem) је важан проблем са много примена. Размотрићемо овде варијанту TSP са допунским ограничењем да тежине грана одговарају еуклидским растојањима.

Проблем. Нека је $C_1, C_2, \ldots, C_n$ скуп тачака у равни које одговарају положајима n градова. Пронаћи Хамилтонов циклус минималне дужине (маршруту трговачког путника) међу њима.

Проблем је и даље NP-комплетан (ово тврђење наводимо без доказа), али видећемо да претпоставка да су растојања еуклидска омогућује конструкцију приближног алгоритма за његово решавање. Прецизније, ова претпоставка може се уопштити, замењујући је претпоставком да растојања задовољавају **неједнакост троугла**, односно да је растојање између два чвора увек мање или једнако од дужине произвољног пута између њих преко осталих чворова.

Најпре се конструише минимално повезујуће стабло (MCST; овде су цене грана њихове дужине), што је много лакши проблем (видети одељак 6.6). Тврдимо да цена стабла није већа од дужине најбољег циклуса TSP. Заиста, циклус TSP садржи све чворове, па га уклањање једне гране чини повезујућим стаблом, чија цена је већа или једнака од цене MCST.

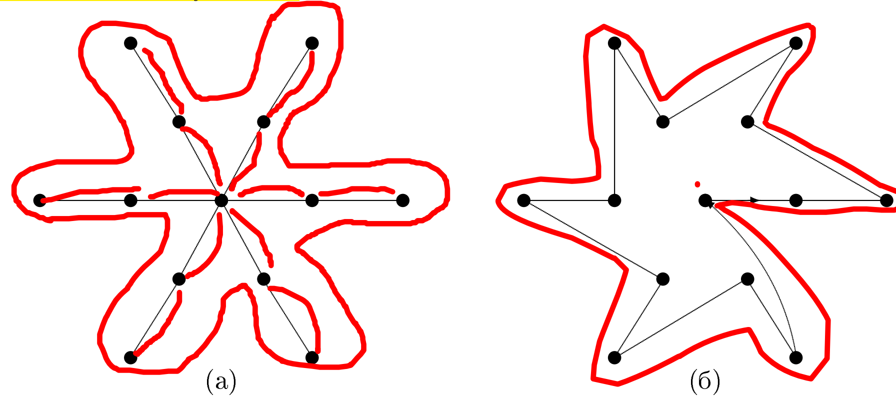
Оjлеров циклус

Од повезујућег стабла се, међутим, не добија директно циклус TSP. Посматрајмо најпре циклус који се добија претрагом у дубину овог стабла (полазећи од произвољног чвора), и садржи грану у обрнутом смеру увек кад се грана приликом враћања пролази у супротном смеру (овај циклус одговара, на пример, обиласку галерије у облику стабла, са сликама на оба зида сваког ходника, идући увек удесно). Свака грана се тако пролази тачно два пута, па је цена овог циклуса највише два пута већа од цене MCST. На крају се овај циклус преправља у циклус TSP, идући пречицом увек кад треба проћи кроз грану већ укључену у циклус (видети слику 13). Другим речима, уместо повратка

1. Минимално повезујуће стабло
2. Дуплирање грана
3. Ојлеров циклус
4. Пречице - Хамилтонов циклус

2. Додавање грана између чворова непарног степена, са најмањим збиром дужина

старом граном, идемо директно до првог непрегледаног чвора. Претпоставка да су растојања еуклидска је важна, јер обезбеђује да је увек директни пут између два града бар толико добар, колико било који заобилазни пут. Према томе, дужина добијеног циклуса TSP је још увек мања од двоструке дужине минималног циклуса TSP.



HC <= Ojlerov ciklus <= 2m

1.5m

Рис. 13. (а) Минимално повезујуће стабло (MCST) (б) циклус TSP добијен од MCST полазећи од средњег чвора и идући увек удесно.

Сложеност. Временска сложеност овог алгоритма одређена је бројем корака у алгоритму за конструкцију MCST.

11.5.2.4. *Побољшање.* Алгоритам који смо управо описали може се побољшати на следећи начин. Његов "најгрубљи" део је претварање обиласка стабла у циклус TSP. Та конверзија може се посматрати и на други начин: она формира Ојлеров циклус од стабла у коме је свака грана удвостручена. После тога се конструише циклус TSP коришћењем пречица у Ојлеровом циклусу. Конверзија стабла у Ојлеров циклус може се извести много рационалније. Ојлеров граф може да садржи само чворове парног степена. Посматрајмо све чворове непарног стапена у стаблу. Број таквих чворова мора бити паран (у противном би сума степена чворова била непарна, што је немогуће, јер је та сума једнака двоструком броју грана). Ојлеров граф се од стабла може добити додавањем довољног броја грана, чиме се постиже да степени свих чворова постану парни. Пошто се циклус TSP састоји од Ојлеровог циклуса (са неким пречицама), волели бисмо да минимизирамо збир дужина додатих грана. Размотримо сада тај проблем.

Дато је стабло у равни, а циљ је додати му неке гране, са минималном сумом дужина, тако да добијени граф буде Ојлеров. Сваком чвору непарног степена мора се додати бар једна грана. Покушајмо да постигнемо циљ додавањем тачно једне гране сваком таквом чвору. Претпоставимо да има $2k$ чворова непарног степена. Ако додамо k грана, тако да свака од њих спаја два

чвора непарног степена, онда ће степени свих чворова постати парни. Проблем тако постаје проблем *упаривања*. Потребно је пронаћи упаривање минималне дужине које покрива све чворове непарног степена. Налажење оптималног упаривања може се извести за $O(n^3)$ корака за произвољни граф (ово тврђење наводимо без доказа). Коначни циклус TSP се затим добија од Ојлеровог графа (који обухвата минимално повезујуће стабло и упаривање минималне дужине) коришћењем пречица. Циклус TSP добијен овим алгоритмом од стабла са слике 13 приказан је на слици 14.

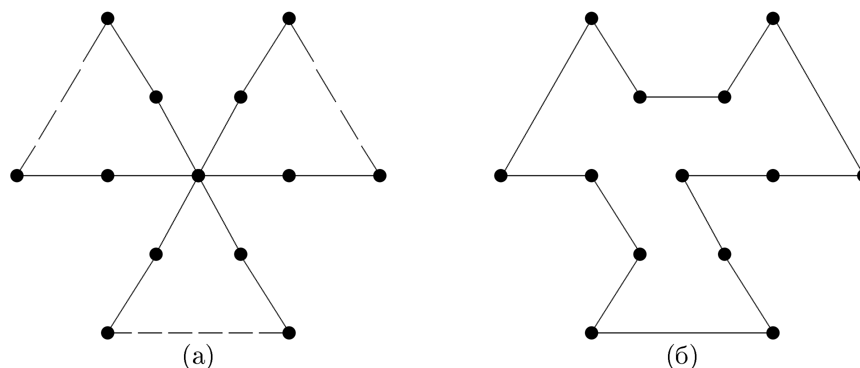


Рис. 14. Минимални Ојлеров циклус и одговарајући циклус TSP (а) минимално повезујуће стабло са упаривањем, (б) циклус TSP добијен од Ојлеровог циклуса.

Теорема 11.12. *Побољшани алгоритам даје циклус* TSP *чија је дужина највише* 1.5 *пута већа од дужине минималног циклуса* TSP.

Доказательство. Потребно је оценити дужину Ојлеровог циклуса, јер се дужина циклуса после увођења евентуалних пречица може само смањити. Ојлеров циклус се састоји од стабла и упаривања. Означимо са Q минимални циклус TSP, а са $|Q|$ његову дужину. Видели смо већ да је дужина стабла мања или једнака од $|Q|$; према томе, довољно је доказати да дужина упаривања не прелази $|Q|/2$. Циклус Q садржи све чворове. Нека је D скуп чворова непарног степена у полазном стаблу. За скуп D могу се формирати два дисјунктна упаривања са збиром дужина $\leq |Q|$ на следећи начин (видети слику 15, на којој су заокружени чворови из D). Започињемо са произвољним чвором $v \in D$ и упарујемо га са са најближим чвором (у смеру казаљке на сату) дуж циклуса Q. После тога наставља се са упаривањем у истом смеру. Ако упарени чворови нису суседи у Q, онда је растојање између њих мање или једнако од дужине пута који их повезује у Q (због неједнакости троугла). Овај процес даје једно упаривање. Друго упаривање добија се понављањем истог процеса у смеру супротном од казаљке на сату. Сума дужина оба упаривања је највише $|Q|$, видети слику 15. Међутим, пошто је M упаривање минималне тежине у

D, његова дужина је мања или једнака од дужине бољег од два споменута упаривања (са збиром $|Q|$), па је дакле мања или једнака од $|Q|/2$. □

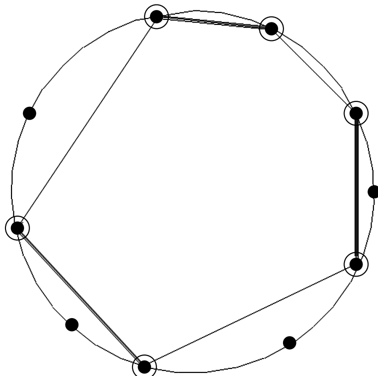


Рис. 15. Два упаривања чија сума дужина не прелази дужину минималног циклуса TSP.

Налажење упаривања минималне тежине знатно је сложеније од налажења минималног повезујућег стабла, али се тиме добија боља граница. Наведени пример илуструје основну карактеристику овог типа алгоритама: уопштава се лакши проблем — или се ослабљују (релаксирају) неки елементи полазног проблема — и тако се формира хеуристика.

11.6. Резиме

Претходна поглавља су можда изазвале оптимизам у вези са могућношћу конструкције добрих алгоритама. Ово поглавље треба да нас приближи реалности. Има много важних проблема који се, нажалост, не могу решити елегантним, ефикасним алгоритмима. Потребно је да препознајемо такве проблеме и да им проналазимо неоптимална, приближна решења. При нападу на неки проблем на располагању су два приступа. Може се покушати са техникама из претходних поглавља, да би се проблем решио, или могу искористити технике уведене у овом поглављу, и показати да је проблем NP-комплетан. Да би се избегли многобројни безуспешни покушаји пре избора исправног приступа, потребно је развити интуицију за оцену тежине проблема.

Задаци

11.1. Показати да су свака два нетривијална проблема из класе P полиномијално еквивалентна.

11.2. Размотрите следећи алгоритам за утврђивање да ли граф има клику величине k. Најпре се генеришу скупови од по тачно k чворова (има их $O(n^k)$). Затим се за сваки подграф индукован тим подскупом проверава да ли је комплетан. Зашто ово није алгоритам полиномијалне временске сложености (што би имало за последицу P=NP)?

11.3. Написати 3SAT израз који се добија применом поступка свођења SAT на 3SAT (одељак 11.4.3) на израз

$$(x + y + \bar{z} + w + u + \bar{v})(\bar{x} + \bar{y} + z + \bar{w} + u + v)(x + \bar{y} + \bar{z} + w + u + \bar{v})(x + \bar{y}).$$

11.4. Доказати да проблем покривач грана остаје NP-комплетан чак и ако се уведе ограничење да сви чворови у графу морају имати паран степен.

11.5. Доказати NP-комплетност следећег проблема. За дати неусмерени повезан граф $G = (V, E)$ и природни број k установити да ли G садржи клику величине k и независни скуп величине k.

11.6. Доказати да је следећи проблем NP-комплетан. Дат је неусмерени граф G и природни број k. Утврдити да ли G садржи подскуп од k чворова, чији је индуковани подграф ациклички.

11.7. Нека је E израз у конјуктивној нормалној форми (КНФ), у коме се свака променљива x појављује тачно једном, и њена негација $\bar{x}$ појављује се тачно једном. Конструисати алгоритам полиномијалне временске сложености за утврђивање да ли је израз E задовољив, или доказати да је овај проблем NP-комплетан.

11.8. Доказати да је следећа варијанта проблема 3SAT NP-комплетна. Улаз је исти као и за проблем 3SAT. Потребно је утврдити да ли постоји придруживање вредности променљивим, такво да је дата формула тачна, и да у свакој клаузи *тачно* један литерал има вредност 1.

11.9. Задат је *регуларни* граф G (граф чији сви чворови имају исти степен) и природни број k. Треба утврдити да ли G садржи клику величине k. Доказати да је овај проблем NP-комплетан.

11.10. Нека је $G = (V, E)$ неусмерени граф, чијем је сваком чвору придружен неки посао. Два чвора су повезана граном, ако се одговарајући послови *не могу* извршавати истовремено (јер нпр. захтевају исти ресурс). Ово су једина ограничења истовремености послова: сваки скуп послова, таквих да било која два посла из скупа нису повезани, може се извршити у једном кораку. Доказати да је следећи проблем NP-комплетан. Дат је граф $G = (V, E)$ и природни број k. Установити могу ли се сви послови који одговарају чворовима извршити за k корака.

11.11. Навести пример фамилије графова, за коју трајање бојења са три боје алгоритмом претраге са слике 12 расте експоненцијално са бројем чворова n.

Глава 12

Паралелни алгоритми

12.1. Увод

Паралелно израчунавање више није егзотична област, и већ дуже време је на главном правцу развоја рачунарства. Развија се врло брзо, чак и у односу на друге рачунарске области. У употреби је више типова паралелних рачунара, са бројем процесора у опсегу од 2 до 65536, и већим. Разлике између различитих постојећих рачунара, чак и са аспекта необавештеног корисника, врло су велике. Немогуће је усвојити један општи модел израчунавања који би обухватао све паралелне рачунаре

У овом поглављу нису покривене све (па чак ни већина) области повезаних са паралелним израчунавањем. Приказани су примери коришћења појединих модела израчунавања и различите технике. Циљ је стицање представе о паралелним алгоритмима и упознавање са потешкоћама везаним за њихову конструкцију. Почиње се са заједничким карактеристикама паралелних алгоритама. Затим се укратко описују неки основни модели паралелног израчунавања, а на крају се наводе примери алгоритама и техника.

Основне мере сложености за секвенцијалне алгоритме су време извршавања и величина коришћене меморије. Ове мере су важне и код паралелних алгоритама, али се мора водити рачуна и о другим ресурсима, посебно о броју процесора. Постоје проблеми који су суштински секвенцијални, који се не могу "паралелизовати" чак ни ако је на располагању неограничени број процесора. Ипак, већина осталих проблема може се до неког степена паралелизовати. Што више процесора се користи — до неке границе — алгоритам се брже извршава. Важно је проучавати ограничења паралелних алгоритама, и бити у стању окарактерисати проблеме за које постоје врло брза паралелна решења. Пошто је број процесора ограничен, исто тако је важно да се процесори ефикасно користе. Следећи важан елеменат је комуникација између процесора. Често је више времена потребно да два процесора размене податке, него да се изврше једноставне операције са подацима. Поред тога, трајање размене података може да зависи од "удаљености" два процесора у рачунару. Према томе, важно је минимизирати комуникацију и организовати је на ефикасни начин. Следеће важно питање је синхронизација, која је велики проблем код паралелних алгоритама кад се извршавају на независним машинама, повезаним неком мрежом

за комуникацију. Такви алгоритми се обично зову **дистрибуирани алгоритми**. Њих због недостатка простора овде нећемо разматрати; ограничићемо се на моделе са потпуном синхронизацијом.

Неки модели паралелног израчунавања садрже ограничење да сви процесори у једном кораку извршавају једну исту инструкцију (над евентуално различитим подацима). Паралелни рачунари са оваквим ограничењем зову се SIMD (скраћеница од Single–Instruction Multiple–Data) рачунари. Паралелни рачунари код којих сваки процесор може да извршава различити програм зову се MIMD (скраћеница од Multiple–Instruction Multiple–Data) рачунари. Уколико се не нагласи другачије, претпоставља се да су рачунари о којима је реч MIMD рачунари.

12.2. Модели паралелног израчунавања

Детаљан преглед модела паралелних рачунара захтевао би више простора. Споменућемо само основне моделе, са нагласком на оне који се користе у овом поглављу. У овом одељку изложићемо нека општа разматрања и дефиниције који се односе на многе моделе. Сваки од следећих одељака покрива један од типова модела, садржи његов детаљнији опис и примере алгоритама.

Време извршавања алгоритма означаваћемо са $T(n,p)$, где је n величина улаза, а p број процесора. Однос

$$S(p) = T(n,1)/T(n,p)$$

зове се **убрзање** алгоритма. Паралелни алгоритам је најефикаснији кад је $S(p) = p$, кад алгоритам достиже **савршено убрзање**. За вредност $T(n,1)$ треба узети *најбољи* познати секвенцијални алгоритам. Важна мера искоришћености процесора је **ефикасност** паралелног алгоритма, која се дефинише изразом

$$E(n,p) = \frac{S(p)}{p} = \frac{T(n,1)}{pT(n,p)}.$$

Ефикасност је однос времена извршавања на једном процесору (кад извршава секвенцијални алгоритам) и *укупног времена* извршавања на p процесора (укупно време је стварно протекло време помножено бројем процесора). Ефикасност указује на удео процесорског времена, које се ефективно користи у односу на секвенцијални алгоритам. Ако је $E(n,p) = 1$, онда је количина рачунања обављеног на свим процесорима у току извршавања алгоритма једнака количини рачунања коју захтева секвенцијални алгоритам. У том случају постигнуто је оптимално искоришћење процесора. Постизање оптималне ефикасности је ретко, јер се у паралелним алгоритмима морају извршити нека допунска израчунавања, која нису потребна код секвенцијалног алгоритма. Један од основних циљева је максимизирање ефикасности.

При конструкцији паралелног алгоритма могло би се фиксирати p, у складу са бројем процесора на располагању, и покушати са минимизирањем $T(n,p)$. Али недостатак оваквог приступа је у томе што би он могао да захтева нови алгоритам, кад год се промени број процесора. Згодније би било конструисати

алгоритам који ради за што је могуће више различитих вредности p. Размотрићемо сада како трансформисати алгоритам који ради за неку вредност p, у алгоритам за мању вредност p, без значајне промене ефикасности. У општем случају, алгоритам са $T(n,p) = X$ може се трансформисати у алгоритам са $T(n,p/k) \simeq kX$, за произвољну константу $k > 1$. Другим речима, може се користити за фактор k мање процесора, чије је време рада онда дуже за фактор k. Модификовани алгоритам може се конструисати заменом сваког корака полазног алгоритма са k корака, у којима један процесор *емулира* (паралелно) извршавање једног корака на k процесора. Овај принцип није увек применљив. На пример, могуће је да p није дељиво са k, или да алгоритам зависи од начина повезивања процесора (о чему ће бити речи у одељку 12.4), или да доношење одлуке о томе које процесоре емулирати захтева такође утрошак неког времена. Ипак, овај принцип, такозвани **принцип имитирања паралелизма**, принцип имитирања паралелизма врло је општи и користан. Он показује да се може смањити број процесора, не мењајући битно ефикасност. Ако, на пример, полазни алгоритам (који је конструисан за велике p) има велико убрзање, онда се могу добити алгоритми који постижу приближно исто убрзање за било коју мању вредност p. Према томе, треба конструисати алгоритам са што бољим убрзањем за максимални број процесора, при чему ефикасност треба да буде добра (тј. блиска јединици). Затим, ако је на располагању мањи број процесора, и даље се може користити исти алгоритам. С друге стране, паралелни алгоритми са малом ефикасношћу су корисни само ако је на располагању велики број процесора. Претпоставимо, на пример, да имамо алгоритам са $T(n,1) = n$ и $T(n,n) = \log_2 n$, односно са убрзањем $S(n) = n/\log_2 n$ и ефикасношћу $E(n) = 1/\log_2 n$. Претпоставимо да нам је на располагању $p = 256$ процесора и да је $n = 1024$. Време извршавања паралелног алгоритма је $T(1024, 256) = 4 \log_2 1024 = 40$ (уз претпоставку да је могуће имитирање паралелизма биће $T(n,p) = (n \log_2 n)/p$), што је убрзање за фактор око 25 у односу на секвенцијални алгоритам. С друге стране, за $p = 16$ време извршавања је 640, што даје недовољно убрзање (мање од 2 са 16 процесора).

Модели паралелног израчунавања разликују се међусобно углавном по начину комуникације и синхронизације процесора. Разматраћемо само моделе који подразумевају потпуну синхронизацију и различите начине повезивања. Модели са **заједничком меморијом** претпостављају да постоји заједничка меморија са равномерним приступом, тако да сваки процесор може да приступи свакој променљивој за јединично време. Ова претпоставка о времену приступа независном од броја процесора и величине меморије није баш реална, али је добра апроксимација. Модели са заједничком меморијом разликују се по начину на који обрађују конфликте приликом приступа меморији. Поједине алтернативе размотрићемо у одељку 12.3.

Заједничка меморија је обично најједноставнији начин за моделирање комуникације, али начин који је најтеже хардверски реализовати. Други модели претпостављају да су процесори међусобно повезани посредством **мреже**.

Мрежа рачунара се може представити графом, при чему чворови одговарају процесорима, а два чвора су повезана ако између одговарајућих процесора постоји директна веза. Сваки процесор обично има локалну меморију, којој може да приступа брзо. Комуникација се остварује порукама, које морају да прођу више директних веза да би дошле до одредишта. Према томе, брзина комуникације зависи од растојања између процесора који размењују поруке. Неколико различитих графова је анализирано у улози скелета мреже рачунара. Више популарних конфигурација наведено је у одељку 12.4.

Следећи модел који ћемо размотрити је модел **систоличког рачунања**. Систоличка архитектура подсећа на покретну траку у фабрици. Подаци се крећу кроз процесоре равномерно, и том приликом се над њима изводе једноставне операције. Уместо да приступају заједничкој (или локалној) меморији, процесори добијају улазне податке од својих суседа, обрађују их, и прослеђују даље. Неки систолички алгоритми наведени су у одељку 12.5.

Коло је основни теоријски модел, који ће бити коришћен само за потребе илустрације. Коло је усмерени ациклички граф, у коме чворови одговарају једноставним операцијама, а гране показују кретање операнада. На пример, Булово коло је оно у коме су улазни степени чворова највише два, а све операције су Булове операције (дисјункција, конјункција или негација). Посебно су издвојени улазни (са улазним степеном нула) и излазни чворови (са излазним степеном нула). Дубина кола је дужина најдужег пута од неког улазног до неког излазног чвора. Дубина одговара времену извршавања паралелног алгоритма.

12.3. Алгоритми за рачунаре са заједничком меморијом

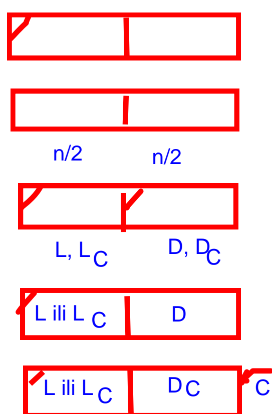
Рачунар са заједничком меморијом састоји се од више **процесора** и **заједничке меморије**. У овом одељку бавићемо се само потпуно синхронизованим алгоритмима. Претпостављамо да се израчунавање састоји од *корака*. У сваком кораку сваки процесор извршава неку операцију над подацима којима располаже, чита из заједничке меморије или пише у заједничку меморију (у пракси сваки процесор може да има и локалну меморију). Модели са заједничком меморијом разликују се по томе како обрађују меморијске конфликте. Модел EREW (Exclusive–Read Exclusive–Write) не дозвољава да два процесора истовремено приступају истој меморијској локацији. Модел CREW (Concurent–Read Exclusive–Write) дозвољава да више процесора истовремено читају са исте меморијске локације, али не дозвољава да два процесора истовремено пишу на исту локацију. На крају, модел (Concurent–Read Concurent–Write CRCW) не намеће никаква ограничења на приступ процесора меморији.

Модели EREW и CREW су добро дефинисани, али није јасно шта је резултат истовременог писања од стране два процесора на једну исту меморијску локацију. Има више начина за обраду истовремених писања. Најслабији CRCW модел, једини који ће бити овде разматран, дозвољава да више процесора истовремено пишу на исту локацију само ако записују исту вредност. Ако два процесора покушају да упишу истовремено различите вредности на исту

локацију, прекида се са извршавањем алгоритма. Иако је то можда неочекивано, видећемо у одељку 12.3.2 да је овакав модел врло моћан. Друга могућност је претпоставити да су процесори нумерисани, и да, ако више процесора покушају истовремени упис на исту локацију, реализује се упис процесора са највећим редним бројем.

12.3.1. Паралелно сабирање. Започињемо са једноставним примером паралелног алгоритма за решавање проблема, који је на први поглед суштински секвенцијалан.

Проблем. Израчунати суму два n-битна бинарна броја.



Обичан секвенцијални алгоритам најпре сабира два бита најмање тежине, а онда у сваком кораку сабира по два бита, и збиру евентуално додаје пренос са ниже позиције. На први поглед немогуће је предвидети исход i-тог корака, све док се не саберу $i-1$ бита најмање тежине, јер пренос може, а не мора да постоји. Ипак, могуће је конструисати паралелни алгоритам.

Користимо индукцију по n. Прелаз са $n-1$ на n не може да буде од велике користи, јер води итеративном секвенцијалном алгоритму. Приступ заснован на декомпозицији често даје добре резултате код паралелних алгоритама, па и у овом случају, јер може да реши све мање потпроблеме паралелно. Претпоставимо да смо поделили проблем на два потпроблема величине $n/2$, тј. на сабирање левих и десних сабирака од по $n/2$ бита (због једноставности претпостављамо да је n степен двојке). Суме два пара бројева могу се израчунати паралелно. Ипак, још увек остаје проблем преноса. Ако сума делова мање тежине има пренос, мора се променити сума делова веће тежине.

Проблем решава запажање да постоје само две могућности — пренос постоји или не. Захваљујући томе, може се *појачати индуктивна хипотеза*, тако да обухвати оба случаја. Модификовани проблем гласи: пронаћи суму два броја, са и без преноса на најнижу позицију. Претпоставимо да смо решили модификовани проблем за оба пара сабирака, и тако добили четири броја L, L_C, D и D_C, који представљају суму нижег пара без преноса, исту суму са преносом, и одговарајуће суме за виши пар бројева, редом. За сваку од тих сума такође знамо да ли је при њеном израчунавању дошло до преноса. Укупну суму S (без преноса на најнижи бит) чине L са D или D_C, зависно од тога да ли је сума L произвела пренос. Укупна сума S_C са преносом добија се на исти начин, само се L замењује са L_C.

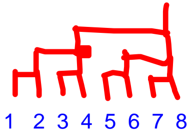
Проблем величине n своди се на решавање два потпроблема величине $n/2$ и на извршавање константног броја корака за обједињавање два резултата. Пошто се оба потпроблема могу решити паралелно — претпоставка је да процесори могу да приступе различитим битима независно — добија се диференцна једначина $T(n,n) = T(n/2,n/2) + O(1)$, чије је решење $T(n,n) = O(\log n)$. Поред тога, пошто су два потпроблема потпуно независна, овај алгоритам подразумева само модел EREW. Овај алгоритам није најбољи за паралелно сабирање али је добар пример једноставне паралелизације алгоритма. Кад за неки проблем

постане јасно да се може ефикасно решити паралелно, решење се може даље побољшавати.

12.3.2. Алгоритми за налажење максимума.

Проблем. Пронаћи највећи од n различитих бројева, задатих у вектору.

Овај проблем решићемо за два различита модела са заједничком меморијом, EREW и CRCW. Алгоритми за оба модела користе технике које се користе при решавању многих других проблема.



12.3.2.1. EREW *модел.* Директни секвенцијални алгоритам за налажење максимума захтева $n-1$ упоређивање. Упоређивање се може схватити као партија коју играју два броја, у којој побеђује већи. Проблем налажења максимума је тада еквивалентан организовању турнира, у коме је победник највећи број у целом скупу. Ефикасан начин да се турнир организује паралелно је да се искористи стабло. Играчи се деле у парове за прво коло (при чему евентуално један играч не учествује, ако је укупан број играча непаран), победници се поново деле у парове, и тако даље до финала. Број кола је $\lceil \log_2 n \rceil$. Турнир се може трансформисати у паралелни алгоритам тако што се свакој партији додели процесор (процесор игра улогу судије у партији). Међутим, треба одбезбедити да сваки процесор зна бројеве – такмичаре. То се може постићи копирањем победника у партији на позицију са већим индексом од позиција два учесника партије. Прецизније, ако партију играју x_i и x_j, $j > i$, онда се већи од бројева x_i, x_j копира на позицију j. У првом колу процесор P_i упоређује x_{2i-1} са x_{2i} ($1 \leq i \leq n/2$), и замењује их ако је потребно, тако да на већу позицију оде већи број. У другом колу процесор P_i упоређује x_{4i-2} са x_{4i} ($1 \leq i \leq n/4$), и тако даље. Ако је на пример $n = 2^k$, онда у последњем, k-том колу, P_1 упоређује $x_{n/2}$ са x_n, и евентуално их замењује. Највећи број налазиће се на позицији n. Пошто сваки број у једном тренутку учествује само у једној партији, довољан је модел EREW. Време извршавања овог једноставног алгоритма је очигледно $O(\log n)$. Покушаћемо сада да смањимо број коришћених процесора.

Алгоритам који смо управо размотрили захтева $\lfloor n/2 \rfloor$ процесора, а број корака је $T(n, \lfloor n/2 \rfloor) = \lceil \log_2 n \rceil$. Пошто је за секвенцијални алгоритам $T(n, 1) = n - 1$, ефикасност овог паралелног алгоритма је $E(n, n/2) \simeq 2/\log_2 n$. Ако нам је ионако на располагању $\lfloor n/2 \rfloor$ процесора (на пример, ако је алгоритам за налажење максимума део другог алгоритма, коме је неопходно толико процесора), онда је овај алгоритам једноставан и ефикасан. Међутим, уз мали напор може се доћи до алгоритма са временом извршавања $O(\log n)$ и ефикасношћу $O(1)$.

Укупан број упоређивања потребних за овај алгоритам је $n-1$, исти као и за секвенцијални алгоритам. Разлог мале ефикасности лежи у томе што се већина процесора не користи у каснијим колима. Ефикасност се може побољшати смањивањем броја процесора и уравнотежавањем њиховог оптерећења на следећи начин. Претпоставимо да користимо само око $n/\log_2 n$ процесора. Улаз се може поделити у $n/\log_2 n$ група (са приближно $\log_2 n$ елемената у свакој групи) и затим свакој групи доделити по један процесор. У првој фази сваки

процесор проналази максимум у својој групи користећи секвенцијални алгоритам, који се састоји од око $\log_2 n$ корака. После тога остаје да се одреди максимум отприлике $n/\log_2 n$ максимума, при чему сад има довољно процесора да се искористи турнирски алгоритам. Време извршавања турнира (под претпоставком да је n степен двојке) је $T(n, \lceil n/\log_2 n \rceil) \simeq 2\log_2 n$. Одговарајућа ефикасност је $E(n) \simeq 1/2$. Покушаћемо сада да формализујемо ову идеју, која омогућује уштеду на броју процесора.

За алгоритам кажемо да је **статички** ако се унапред зна придруживање процесора операцијама. Дакле, унапред знамо за сваки корак i алгоритма и за сваки процесор P_j операцију и операнде које P_j користи у кораку i. Алгоритам за налажење максимума је пример статичког алгоритма, јер се унапред знају индекси учесника у свакој партији.

Лема 12.1 (Брентова лема). *Ако постоји статички* EREW *алгоритам са* $T(n, p) = O(t(n))$, *такав да је укупан број корака (на свим процесорима)* $s(n)$, *онда постоји статички* EREW *алгоритам са* $T(n, s(n)/t(n)) = O(t(n))$.

Приметимо да ако је $s(n)$ једнако секвенцијалној сложености алгоритма, онда модификовани алгоритам има ефикасност $O(1)$.

Доказательство. Нека је $T(n, p) \le t(n)$ за све довољно велике n, и нека је a_i укупан број корака које извршавају сви процесори у i-том кораку алгоритма, $i = 1, 2, \ldots, t(n)$. Тада је $\sum_{i=1}^{t(n)} a_i = s(n)$. Ако је $a_i \le s(n)/t(n)$, онда има довољно процесора за паралелно извршавање корака i. У противном се корак i замењује са $\lceil a_i/(s(n)/t(n)) \rceil$ корака у којима расположивих $s(n)/t(n)$ процесора емулирају кораке, које у оригиналном алгоритму извршава p процесора (користећи принцип имитирања паралелизма). Укупан број корака је сада

$$\sum_{i=1}^{t(n)} \left\lceil \frac{a_i}{s(n)/t(n)} \right\rceil \le \sum_{i=1}^{t(n)} \left(\frac{a_i t(n)}{s(n)} + 1 \right) = t(n) + \frac{t(n)}{s(n)} \sum_{i=1}^{t(n)} a_i = 2t(n).$$

Према томе, време извршавања модификованог алгоритма је такође $O(t(n))$. □

Ово тврђење познато је као Брентова лема. Брентова лема показује да у неким случајевима ефикасност паралелног алгоритма одређена односом укупног броја операција (операција које извршавају сви процесори) и времена извршавања секвенцијалног алгоритма.

Ограничење да алгоритам буде статички је потребно, јер се мора знати које процесоре треба емулирати. Брентова лема је тачна и за алгоритме који нису статички, под условом да се емулација може лако извести. Пример где се ова лема не може применити је следећи. Претпоставимо да имамо n процесора и n елемената. После првог корака неки процесори одлучују (на основу резултата првог корака) да престану са радом. Исто се дешава и другом, трећем кораку, итд. Овај алгоритам је сличан турнирском алгоритму, изузев што се у овом случају не зна који процесори одустају од даљег рада. Ако покушамо да емулирамо преостале процесоре после на пример првог корака, потребно је да

знамо који су још активни. Да би се то установило, потребно је извршити нека израчунавања.

12.3.2.2. CRCW *модел.* Намеће се утисак да паралелни алгоритам не може да нађе максимум за мање од $\log_2 n$ корака, ако се користе само упоређивања. Међутим, то није тачно. Следећи алгоритам са временом извршавања $O(1)$ илуструје могућности истовремених уписа. Подразумева се варијанта истовремених уписа, у којој два или више процесора могу да пишу истовремено на исту локацију само ако записују исти податак.

Користи се $n(n-1)/2$ процесора, тако да се процесор P_{ij} додељује пару елемената $\{i,j\}$. Поред тога, сваком елементу x_i придружује се заједничка (дељена) променљива v_i, са почетном вредношћу 1. У првом кораку сваки процесор упоређује своја два елемента и записује 0 у променљиву придружену мањем елементу. Пошто је само један елеменат већи од свих осталих, само једна од променљивих v_i задржава вредност 1. У другом кораку процесори придружени победнику могу да установе да је он победник и да објаве ту чињеницу (на пример, уписивањем његовог индекса у посебну заједничку променљиву, резултат). Овај алгоритам захтева само два корака, независно од n. Међутим, његова ефикасност је врло мала, јер он захтева $O(n^2)$ процесора. Ово је такозвани *двокорачни алгоритам*.

претпоставка: сви елементи су различити

Ефикасност двокорачног алгоритма може се побољшати слично оном код модела EREW. Улазни подаци деле се у мале групе, тако да се свакој групи може доделити довољно процесора, да би се максимум групе могао одредити двокорачним алгоритмом. Са опадањем броја кандидата расте број расположивих процесора по кандидату, па се може повећати величина групе. Двокорачни алгоритам омогућује одређивање максимума у групи величине k са $k(k-1)/2$ процесора, за константно време. Претпоставимо да имамо укупно n процесора и да је n степен двојке. У првом циклусу величина сваке групе је 2 и максимум у свакој групи може се одредити у једном кораку. У други циклус улази се са $n/2$ елемената, и (наравно) n процесора. Ако формирамо групе од по 4 елемента, имаћемо $n/8$ група, што нам омогућује да свакој групи доделимо 8 процесора. Ово је довољно, јер је $4 \cdot (4-1)/2 = 6$. У трећи циклус улази се са $n/8$ елемената. Покушајмо да одредимо највећу могућу величину групе која се може обрадити на овај начин. Ако је величина групе g, онда је број група $n/8g$, и за сваку групу имамо на располагању $8g$ процесора. За примену двокорачног алгоритма на групу величине g потребно је $g(g-1)/2$ процесора, па мора да буде $g(g-1)/2 \le 8g$, односно $g \le 17$; једноставније је узети вредност $g = 16$. Уопште, у i-ти циклус се улази са $n/2^{2^{i-1}-1}$ елемената, који се деле на $n/2^{2^i-1}$ група по $g = 2^{2^{i-1}}$ елемената, $i \ge 1$. За налажење максимума у групи двокорачним алгоритмом довољно је $g(g-1)/2 \le g^2/2 = 2^{2^i-1}$ процесора, па је за налажење максимума у свим групама довољно

$$\frac{n}{2^{2^i-1}} \cdot 2^{2^i-1} = n$$

процесора. У наредни циклус улази се са по једним елементом из сваке групе, дакле са $n/2^{2^i-1}$ елемената, што индукцијом доказује исправност ове конструкције. Број циклуса i до завршетка алгоритма ограничен је условом да је број елемената на почетку i-тог циклуса мањи од један: $n/2^{2^{i-1}-1} \leq 1$, или $i \geq \log_2(\log_2 n + 1) + 1$. Дакле број циклуса, а тиме и број корака приликом извршења овог алгоритма је $O(\log\log n)$.

Иако је овај алгоритам нешто спорији од двокорачног ($O(\log\log n)$ у односу на $O(1)$), његова ефикасност је много боља. Она износи $O(1/\log\log n)$ у односу на $O(1/n)$ код двокорачног алгоритма. Описана техника може се назвати **подели и смрви**, јер се улаз дели у групе, које су довољно мале да се могу "смрвити" мноштвом процесора. Примена ове технике није ограничена на модел CRCW.

12.3.3. Паралелни проблем префикса. Паралелни проблем префикса је важан јер се користи као основни елеменат при конструкцији многих паралелних алгоритама. Нека је $\star$ произвољна *асоцијативна* бинарна операција (операција која задовољава услов $x \star (y \star z) = (x \star y) \star z$) за произвољне x, y и z), коју ћемо означавати именом *производ*. На пример, $\star$ може да означава сабирање, множење или максимум два броја.

Проблем. Дат је низ бројева $x_1, x_2, \ldots, x_n$. Израчунати производе $x_1 \star x_2 \star \cdots \star x_k$ за $k = 1, 2, \ldots, n$.

Означимо са $PR(i, j)$ производ $x_i \star x_{i+1} \star \cdots \star x_j$. Потребно је израчунати $PR(1, k)$ за $k = 1, 2, \ldots, n$. Секвенцијална верзија проблема префикса је тривијална — префикси се једноставно израчунавају редом. Паралелни проблем префикса није тако лако решити. Искористићемо метод декомпозиције, уз уобичајену претпоставку да је n степен двојке.

разлагања

Индуктивна хипотеза. Умемо да решимо паралелни проблем префикса за $n/2$ елемената.

2 пута n/2 процесора

Случај једног елемента је тривијалан. Алгоритам започиње поделом улаза на две половине, које се решавају индукцијом. На тај начин добијамо вредности $PR(1, k)$ и $PR(n/2 + 1, n/2 + k)$ за $k = 1, 2, \ldots, n/2$. Прва половина ових вредности може се искористити директно. Вредности $PR(1, m)$ за $m = n/2+1, n/2+2, \ldots, n$ добијају се израчунавањем производа $PR(1, n/2) \star PR(n/2+1, m)$. Оба ова члана позната су по индукцији (већ су израчуната; приметимо да је искоришћена асоцијативност операције $\star$). Алгоритам је приказан на слици 1.

Сложеност. Улаз је подељен у два дисјунктна скупа у сваком рекурзивном позиву алгоритма. Оба потпроблема се могу дакле решити паралелно у моделу EREW. Ако имамо n процесора за проблем величине n, онда се половина њих може доделити сваком потпроблему. Комбиновање решења потпроблема састоји се од $n/2$ множења, која се могу извршити паралелно, али је потребан модел CREW, јер се у сваком множењу користи $PR(1, n/2)$, односно $x[Srednji]$. Иако више процесора истовремено да читају $x[Srednji]$,

```
Алгоритам Paralelni_Prefiks_1(x, n);
Улаз: x (вектор са n елемената).
 {претпоставља се да је n степен двојке}
Излаз: x (чији i-ти елеменат садржи i-ти префикс).
begin
    PP_1(1, n)
end
procedure PP_1(Levi, Desni);
begin
    if Desni − Levi = 1 then
        x[Desni] := x[Levi] ⋆ x[Desni] {⋆ је асоцијативна бинарна операција}
    else
        Srednji := (Levi + Desni − 1)/2;
        do in parallel
            PP_1(Levi, Srednji); {додељено процесорима од 1 до n/2}
            PP_1(Srednji + 1, Desni); {додељено процесорима од n/2 + 1 до n}
        for i := Srednji + 1 to Desni do in parallel
            x[i] := x[Srednji] ⋆ x[i]
end
```

Рис. 1. Алгоритам $Paralelni_prefiks_1$.

они пишу на различите локације, па модел CRCW није неопходан. Укупан број корака је $T(n,n) = O(\log n)$, па је ефикасност алгоритма $E(n,n) = O(1/\log n)$ (време извршавања секвенцијалног алгоритма је $O(n)$).

На жалост, ефикасност овог алгоритма не може се побољшати коришћењем Брентове леме, јер је укупан број корака на свим процесорима $O(n \log n)$. Према томе, да би се побољшала ефикасност, мора се смањити укупан број корака.

12.3.3.1. *Побољшање ефикасности паралелног префикса.* Идеја која омогућује решавање проблема је коришћење исте индуктивне хипотезе, али уз поделу улаза на другачији начин. Претпоставимо поново да је n степен двојке и да имамо n процесора. Нека E означава скуп свих x_i са парним индексима i. Ако израчунамо префиксе свих елемената из E, онда је израчунавање осталих префикса (оних са непарним индексима) лако: ако је познато $PR(1, 2i)$, онда се непарни префикс $PR(1, 2i+1)$ добија израчунавањем само још једног производа $PR(1, 2i) \star x_{2i+1}$, $i = 1, 2, \ldots, n/2$, Префикси елемената из E могу се одредити у две фазе. Најпре се (паралелно) израчунавају производи $x_{2i-1} \star x_{2i}$, који се затим смештају у x_{2i}, $i = 1, 2, \ldots, n/2$. Другим речима, израчунавају се производи свих елемената из E са својим левим суседима. Затим се решава (индукцијом) проблем паралелног префикса за $n/2$ елемената из E. Резултат за свако x_{2i} је тачан префикс, јер је свако x_{2i} већ замењено производом са x_{2i-1}. Пошто се знају префикси за све елементе са парним индексима, преостали

префикси се могу израчунати у једном паралелном кораку на већ споменути начин. Лако се проверава да се овај алгоритам може извршавати у моделу EREW. Алгоритам је приказан на слици 2.

Алгоритам Paralelni_Prefiks_2(x, n);
Улаз: x (вектор са n елемената).
{претпоставља се да је n степен двојке}
Излаз: x (чији i-ти елеменат садржи i-ти префикс).
begin
    $PP_2(1)$
end
procedure PP_2$(Korak)$;
begin
    if $Korak = n/2$ **then**
        $x[n] := x[n/2] \star x[n]$ {$\star$ је асоцијативна бинарна операција}
    else
        for $i := 1$ **to** $n/(2 \cdot Korak)$ **do in parallel**
            $x[2 \cdot i \cdot Korak] := x[(2 \cdot i - 1) \cdot Korak] \star x[2 \cdot i \cdot Korak]$;
        $PP_2(2 \cdot Korak)$;
        for $i := 1$ **to** $n/(2 \cdot Korak) - 1$ **do in parallel**
            $x[(2 \cdot i + 1) \cdot Korak] := x[2 \cdot i \cdot Korak] \star x[(2 \cdot i + 1) \cdot Korak]$
end

x 0 0 0 0 0 0 0
x x x x x x x x

Рис. 2. Алгоритам *Paralelni_prefiks_2*.

Сложеност. Обе петље у алгоритму *Paralelni_prefiks_2* могу се извршити паралелно за време $O(1)$ са $n/2$ процесора. Рекурзивни позив примењује се на проблем двоструко мање величине, па је време извршавања алгоритма $O(\log n)$. Укупан број корака $s(n)$ задовољава диференцну једначину $s(n) = s(n/2) + n - 1$, $s(2) = 1$, из чега следи да је $s(n) = O(n)$ (прецизније, $s(2^k) = 2^{k+1} - k - 2$). Због тога се сада може искористити Брентова лема за побољшање ефикасности: алгоритам се може променити тако да се за време $O(\log n)$ извршава на само $O(n/\log n)$ процесора, односно да му ефикасност буде $O(1)$. Кључна идеја побољшања је коришћење само једног рекурзивног позива (уместо два), при чему се корак обједињавања и даље извршава паралелно.

12.3.4. Одређивање рангова у повезаној листи. У паралелним алгоритмима много је теже радити са повезаним листама него са векторима, јер су листе суштински секвенцијалне. Повезаној листи може се приступити само преко главе (првог елемента), и листа се мора пролазити елеменат по елеменат, без могућности паралелизације. У многим случајевима су, међутим, елементи

i	1	2	3	4	5	6	7	8
N[i]	8	5	7	0	3	4	1	6
R[i]				1		2		3

i 1 2 3 4 5 6 7 8
N[i] 2 3 4 5 6 7 8 0
R[i] 1
N[i] 3 4 5 6 7 8 2
4 3
N[i] 5 6 7 8
8 7 6 5

листе (односно показивачи на њих) смештени у вектор; редослед елемената листе независан је од редоследа у вектору. У таквим случајевима, кад се листи приступа паралелно, постоји могућност примене брзих паралелних алгоритама.

Ранг елемента у повезаној листи дефинише се као растојање елемента од краја листе. Тако, на пример, први елеменат има ранг n, други $n-1$, итд.

Проблем. Дата је повезана листа од n елемената који су смештени у низ A дужине n. Израчунати рангове свих елемената листе.

Секвенцијални проблем се може решити простим проласком кроз листу. Метод који ћемо искористити за конструкцију паралелног алгоритма зове се **удвостручавање**. Сваком елементу додељује се по један процесор. На почетку сваки процесор зна само адресу десног (наредног) суседа свог елемента у листи. После првог корака сваки процесор зна елеменат на растојању два (дуж листе) од свог елемента. Ако у кораку i, сваки процесор зна адресу елемента на растојању k од свог елемента, онда у наредном кораку сваки процесор може да пронађе адресу елемента на растојању $2k$. Процес се наставља све док сви процесори не достигну крај листе. Нека је $N[i]$ адреса најдаљег елемента десно од елемента i у листи, коју зна процесор P_i. На почетку је $N[i]$ десни сусед елемента i (изузев за последњи елеменат у листи, чији је показивач на десног суседа **nil**). У суштини, процесор P_i у сваком кораку замењује $N[i]$ вредношћу $N[N[i]]$, све док не достигне крај листе. Нека је $R[i]$ ранг елемента i. На почетку се променљивој $R[i]$ додељује вредност 0, изузев за последњи елеменат у листи, за кога се она поставља на вредност 1 (овај елеменат се од осталих разликује по показивачу, који има вредност **nil**). Кад процесор добије адресу суседа са рангом R различитим од нуле, он може да израчуна свој ранг (односно ранг свог елемента). На почетку само елеменат ранга 1 зна свој ранг. После првог корака елеменат ранга 2 открива да његов сусед има ранг 1, па закључује да је његов сопствени ранг 2. После другог корака елементи са рангом 3 и 4 установљавају своје рангове, итд. Ако P_i установи да $N[i]$ показује на "рангирани" елеменат ранга R после d корака удвостручавања, онда је ранг елемента i једнак $2^{d-1}+R$. Овај алгоритам (слика 3) се може лако прилагодити моделу EREW, видети задатак 12.2. (довољно је да сваки процесор независно израчунава своју копију $D[i]$ променљиве D).

Сложеност. Процес удвостручавања омогућује да сваки процесор достигне крај листе после највише $\lceil \log_2 n \rceil$ корака, па је $T(n,n) = O(\log n)$. Ефикасност алгоритма је $E(n,n) = O(1/\log n)$. Поправка ефикасности захтевала би темељну прераду алгоритма, јер је укупан број корака $O(n \log n)$.

Познавање рангова омогућује трансформацију листе у вектор за $O(\log n)$ корака. После израчунавања свих рангова, елементи се могу прекопирати на своје локације у вектору, па се остатак израчунавања може извршити директно на вектору, што је много једноставније.

```
Алгоритам Rangovi(N);
Улаз: N (низ од n елемената).
Излаз: R (рангови свих елемената у низу).
begin
    D := 1;
    {Сваки процесор може имати своју локалну променљиву D}
    {овде је D заједничка променљива}
    do in parallel {процесор P_i је активан док R[i] не постане различито од нуле}
        R[i] := 0;
        if N[i] = nil then R[i] := 1;
        while R[i] = 0 do
            if R[N[i]] ≠ 0 then
                R[i] := D + R[N[i]]
            else
                N[i] := N[N[i]];
                D := 2 · D
end
```

Рис. 3. Паралелни алгоритам за одређивање рангова елемената повезане листе.

12.3.5. Техника Ојлеровог обиласка. Многи алгоритми за рад са стаблима могу се паралелизовати тако да се паралелно обрађује комплетна генерација чворова (на пример, код турнирског алгоритма за налажење максимума). Време извршавања таквог алгоритма је пропорционално висини стабла. Ако је стабло са n чворова довољно уравнотежено и висина му је $O(\log n)$, онда је овај приступ сасвим добар. Међутим, ако стабло није уравнотежено, висина стабла може да буде $n-1$ у најгорем случају, па се мора тражити неки други приступ. *Техника Ојлеровог обиласка* је алатка за конструкцију паралелних алгоритама на стаблима, погодна и за неуравнотежена стабла.

Нека је T стабло. Претпоставимо да је T представљено уобичајеном листом повезаности, уз једну допуну. Као и обично, постоји показивач $E[i]$ на почетак листе грана суседних чвору i (ако је ова листа празна, онда $E[i]$ има вредност **nil**). Та листа састоји се од слогова који садрже одговарајућу грану (i,j) (при томе је довољно сместити само j, јер је i познато) и показивач $Naredna(i,j)$ на наредну грану у листи. Свака неусмерена грана (i,j) представљена је са две усмерене копије, (i,j) и (j,i). Слогови листе садрже и додатни показивач: слог који одговара грани (i,j) садржи показивач на грану (j,i). Ово је потребно да би се грана (j,i) могла брзо пронаћи кад се зна адреса гране (i,j). Пример овако представљеног стабла дат је на слици 4; показивачи на копије грана због прегледности нису приказани.

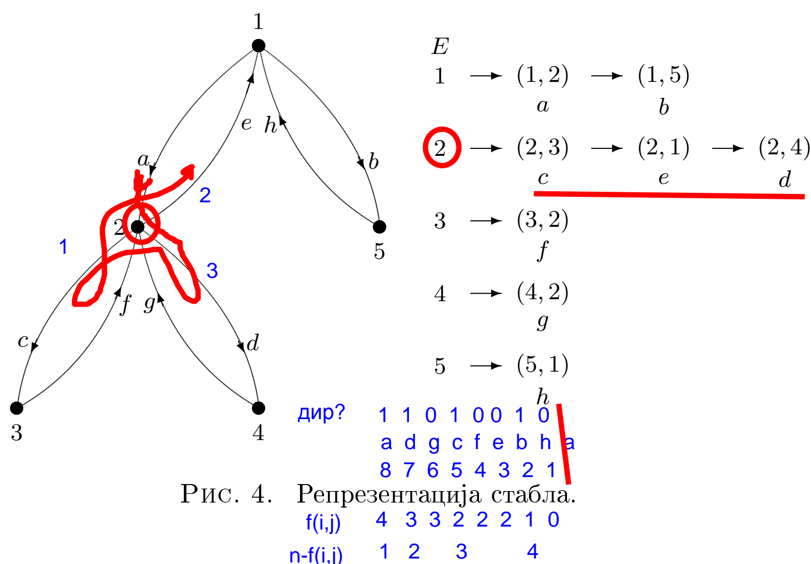


Рис. 4. Репрезентација стабла.

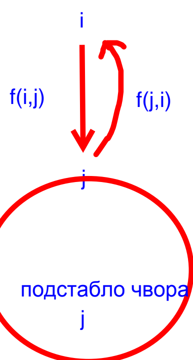
Техника Ојлеровог обиласка заснива се на идеји да се формира листа грана стабла, и то оним редом којим се гране појављују у Ојлеровом циклусу за усмерену верзију стабла (у циклусу се свака грана појављује два пута). Кад се зна ова листа, многе операције са стаблом могу се извести директно на листи, као да је листа линеарна. Секвенцијалним алгоритмом лако се може прегледати стабло и успут извршити потребне операције. Оваква "линеаризација" омогућује да се операције са стаблом ефикасно изводе паралелно. Видећемо два примера таквих операција, пошто претходно размотримо формирање Ојлеровог циклуса.

Секвенцијално налажење Ојлеровог обиласка стабла T (у коме се свака грана појављује два пута) је једноставно. Може се извести обилазак стабла користећи претрагу у дубину, враћајући се супротно усмереном граном приликом сваког повратка назад у току претраге. Слична ствар се може извести и паралелно. Нека $w(i,j)$ означава грану која следи иза гране (i,j) у циклусу. Тврдимо да се $w(i,j)$ може дефинисати следећом једнакошћу

$$w(i,j) = \begin{cases} Naredna(j,i) & \text{ако } Naredna(j,i) \text{ није } \mathbf{nil} \\ E[j] & \text{у осталим случајевима} \end{cases},$$

на основу које се лако паралелно израчунава. Другим речима, листа грана суседних чвору j пролази се цикличким редоследом (ако је (j,i) последња грана у листи чвора j, онда се узима прва грана из те листе, она на коју показује $E[j]$). На пример, ако кренемо од гране a на слици 4, онда се циклус састоји од грана a, d, g, c, f, e, b, h, и поново a. Чињеница да $Naredna(j,i)$ следи иза (i,j) у циклусу обезбеђује да ће грана (j,i) доћи на ред тек кад буду изабране све остале гране суседне чвору j. Према томе, подстабло са кореном у j биће комплетно прегледано пре повратка у чвор i. Доказ исправности ове процедуре остављамо читаоцу као вежбање.

Кад је листа грана у Ојлеровом обиласку конструисана, произвољна грана (r, t) бира се за полазну, а грана која јој претходи означава се као крај листе. Чвор r се бира за *корен* стабла. После тога се гране алгоритмом *Rangovi* са слике 3 могу нумерисати, у складу са својим положајем у листи. Нека $R(i, j)$ означава ранг гране (i, j) у листи. Тако је, на пример, $R(r, t) = 2(n-1)$, где је n број чворова. Приказаћемо сада два примера операција са стаблом — долазну нумерацију чворова, и израчунавање броја потомака за све чворове.



За грану (i, j) у циклусу кажемо да је *директна грана* ако је усмерена од корена, односно да је *повратна грана* у противном. Нумерација чворова омогућује разликовање директних од повратних грана: грана (i, j) је директна грана ако и само ако је $R(i, j) > R(j, i)$. Пошто су две копије гране (i, j) повезане показивачима, лако је установити која је од њих директна грана. Шта више, ова провера се може обавити паралелно за све гране. Директне гране су интересантне, јер одређују редослед чворова при долазној нумерацији. Нека је (i, j) директна грана која води до чвора j (односно, чвор i је отац чвора j у стаблу). Ако је $f(i, j)$ број директних грана које следе иза (i, j) у листи, онда је редни број чвора j једнак $n - f(i, j)$. Редни број корена r, јединог чвора до кога не води ни једна директна грана, је 1. Применом варијанте алгоритма са удвостручавањем може се сада израчунати вредност $f(i, j)$ за сваку директну грану (i, j). Прецизнију разраду алгоритма остављамо читаоцу као вежбање.

Други пример је израчунавање броја потомака сваког чвора у стаблу. Нека је (i, j) (јединствена) директна грана која води до задатог чвора j. Посматрајмо гране које следе иза гране (i, j) у листи. Број чворова испод j у стаблу једнак је броју директних грана испод j у стаблу. Ми већ знамо како се паралелно израчунавају вредности $f(i, j)$, једнаке броју директних грана које следе иза гране (i, j) у листи. На сличан начин $f(j, i)$ је број директних грана које следе иза гране (j, i) у листи. Лако је видети да је број потомака чвора j једнак $f(i, j) - f(j, i)$. Време извршавања оба описана алгоритма на моделу EREW је $T(n, n) = O(\log n)$

12.4. Алгоритми за мреже рачунара

Мреже рачунара могу се моделирати графовима, обично неусмереним. Процесори одговарају чворовима, а два чвора су повезана граном ако постоји директна веза између одговарајућих процесора. Сваки процесор има своју локалну меморију, а посредством мреже може да приступи локалним меморијама других процесора. Према томе, сва меморија се може делити, али цена приступа некој променљивој зависи од локација процесора и променљиве. Приступ жељеној променљивој може бити брз колико и локални приступ (ако је променљива у истом процесору), или толико спор колико и пролазак кроз целу мрежу (у случају кад граф има облик низа повезаних чворова). Трајање приступа је негде између ове две крајности. Процесори комуницирају разменом порука. Кад процесор жели да приступи променљивој смештеној у локалној меморији другог процесора, он шаље поруку са захтевом за променљивом. Порука се усмерава кроз мрежу.

Више различитих графова користе се за мреже рачунара. Најједноставнији међу њима су линеарни низ, прстен, бинарно стабло, звезда и дводимензионална мрежа. Ефикасност комуникације расте са бројем грана у мрежи. Међутим, гране су скупе — то се може објаснити повећањем површине коју заузимају везе, а тиме повећањем димензија мреже и времена комуникације. Због тога се обично тражи компромис. Не постоји тип графа који је универзално добар. Погодност одређеног графа битно зависи од начина комуницирања у конкретном алгоритму. Међутим, постоје неке особине графова, које могу бити врло корисне. У наставку ћемо их навести, заједно са примерима мрежа рачунара.

Битан параметар мреже је **дијаметар** одговарајућег графа, тј. највеће растојање нека два чвора. Дијаметар одређује максимални број грана на путу порука до одредишта. Дводимензионална мрежа $n \times n$ има дијаметар $2n$, а комплетно бинарно стабло са n чворова има дијаметар $2\log_2(n+1) - 2$. према томе може да испоручи поруку много брже него дводимензионална мрежа. С друге стране, стабло има уско грло, јер сав саобраћај из једне у другу половину стабла пролази кроз корен. Дводимензионална мрежа нема уско грло и врло је симетрична, што је важно за алгоритме у којима је комуникација симетрична.

Хиперкоцка је популарна структура која комбинује предности високе симетрије, малог дијаметра, мноштва алтернативних путева између два чвора и одсуства уских грла. d-димензионална хиперкоцка састоји се од $n = 2^d$ процесора. Адресе процесора су бројеви од 0 до $2^d - 1$. Према томе, свака адреса се састоји од d бита. Процесор P_i је повезан са процесором P_j ако и само ако се i разликује од j на тачно једном биту. Растојање између произвољна два процесора је увек $\leq d$, јер се од P_i до P_j може доћи променом највише d бита, једног по једног. На слици 5 приказана је четвородимензионална хиперкоцка. Хиперкоцка обезбеђује богатство веза, јер постоји много различитих путева између свака два процесора (одговарајући бити могу се мењати произвољним редоследом). Хиперкоцка се може такође комбиновати са мрежом, на пример умећући мреже на стране хиперкоцке. У примени се појављују и друге структуре мрежа.

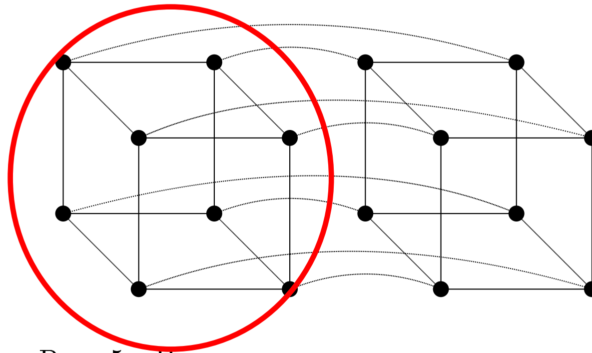


Рис. 5. Четвородимензионална хиперкоцка.

12.4.1. Сортирање на низу. Размотрићемо најпре једноставан проблем сортирања на низу процесора. На располагању је n процесора $P_1, P_2, \ldots, P_n$ и задато је n бројева $x_1, x_2, \ldots, x_n$. Сваки процесор чува један улазни податак. Циљ је прерасподелити бројеве међу процесорима тако да најмањи од њих буде у P_1, следећи у P_2, итд. У општем случају могуће је додељивање више улазних података једном процесору. Видећемо да се алгоритам може прилагодити и таквим условима. Процесори су повезани у линеарни низ: процесор P_i је повезан са процесором P_{i+1}, $i = 1, 2, \ldots, n-1$. Пошто сваки процесор може да комуницира само са суседима, упоређивање и размену података могуће је вршити само између елемената који су суседни у низу. У најгорем случају алгоритам захтева извршавање $n-1$ корака, колико је потребно да се податак премести са једног на други крај низа. Алгоритам се у основи извршава на следећи начин. Сваки процесор упоређује свој број са бројем једног од својих суседа, размењује бројеве ако је њихов редослед погрешан, а затим исти посао обавља са другим суседом (суседи се морају смењивати, јер би се у противном упоређивали увек исти бројеви). Исти процес наставља се све док бројеви не буду поређани на жељени начин. Кораци се деле на *непарне* и *парне*. У непарним корацима процесори са непарним индексом упоређују своје са бројевима својих десних суседа; у парним корацима процесори са парним индексом упоређују своје са бројевима својих десних суседа (слика 6). На тај начин су сви процесори синхронизовани и упоређивање увек врше процесори који то и треба да раде. Ако процесор нема одговарајућег суседа (на пример први процесор у другом кораку), он у току тог корака мирује. Овај алгоритам се зове **сортирање парно-непарним транспозицијама**, видети слику 7. Пример рада алгоритма приказан је на слици 8. Приметимо да се у овом примеру сортирање завршава после само шест корака. Ипак, ранији завршетак тешко је открити у мрежи. Према томе, боље је оставити алгоритам да се извршава до свог завршетка у најгорем случају.

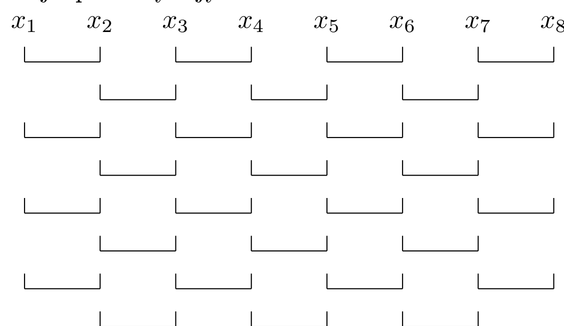


Рис. 6. Сортирање парно-непарним транспозицијама.

Алгоритам *Sortiranje_na_nizu* изгледа природно и јасно, али доказ исправности његовог рада није тривијалан. На пример, елеменат се у неким корацима може *удаљавати* од свог коначног положаја. У примеру на слици 8

Алгоритам Sortiranje_na_nizu(x, n);
Улаз: x (вектор са n елемената, при чему је x_i у процесору P_i).
Излаз: x (сортирани вектор, тако да је i-ти најмањи елеменат у P_i).
begin
  do in parallel $\lceil n/2 \rceil$ пута
    P_{2i-1} и P_{2i} упоређују своје елементе и по потреби их размењују;
      {за све i, такве да је $1 < 2i \leq n$}
    P_{2i} и P_{2i+1} упоређују своје елементе и по потреби их размењују;
      {за све i, такве да је $1 \leq 2i < n$}
      {ако је n непарно, овај корак се извршава само $\lfloor n/2 \rfloor$ пута}
end

Рис. 7. Алгоритам за сортирање на низу процесора.

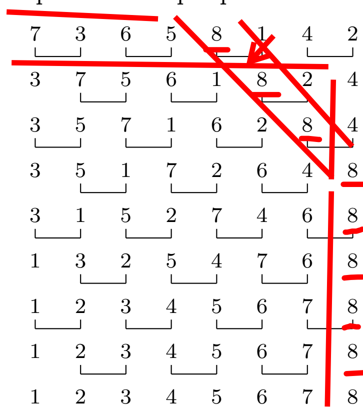


Рис. 8. Пример сортирања парно-непарним транспозицијама.

број 5 се креће улево два корака пре него што почне са кретањем удесно, а 3 иде до левог краја и остаје тамо три корака пре него што крене назад удесно. Доказ исправности рада паралелних алгоритама није једноставан, због међузависности деловања различитих процесора. Понашање једног процесора утиче на све остале процесоре, па је обично тешко усредсредити се на један процесор и доказати да је то што он ради исправно; морају се посматрати сви процесори заједно.

Теорема 12.1. *На крају извршавања алгоритма Sortiranje_na_nizu, дати бројеви су сортирани.*

Доказательство. Доказ се изводи индукцијом по броју елемената. При томе се тврђење мало појачава: сортирање се завршава после n корака, без обзира да ли је први корак паран или непаран. За $n = 2$ сортирање се завршава после највише два корака; сортирање траје два корака ако је први корак

непаран. Претпоставимо да је теорема тачна за n процесора, и посматрајмо случај $n+1$ процесора. Сконцентришимо пажњу на максимални елеменат, и претпоставимо да је то x_m (у примеру на слици 8 то је x_5). У првом кораку x_m се упоређује са x_{m-1} или x_{m+1}, зависно од тога да ли је m парно или непарно. Ако је m парно, нема замене јер је x_m веће од x_{m-1}. Исход је исти као у случају да је број x_m на почетку био у процесору P_{m-1}, и да је замена била извршена. Према томе, без губитка општости може се претпоставити да је m непарно. У том случају број x_m се упоређује са x_{m+1}, замењује, и као највећи се премешта корак по корак удесно (дијагонално на слици 8), све док не дође на место x_{n+1}, а онда остаје тамо. То је позиција коју x_m и треба да заузме, па сортирање исправно третира максимални елемент.

Показаћемо сада индукцијом да је и сортирање осталих n елемената коректно. Да бисмо то доказали, треба да пресликамо акције n процесора на вектору дужине $n+1$ на могући рад $n-1$ процесора на вектору дужинее n. То се може урадити на следећи начин. Посматрајмо дијагоналу насталу кретањем максималног елемената (видети слику 9). Упоређивања у којима учествује максимални елеменат се игноришу. Упоређивања делимо на две групе, она испод, и она изнад дијагонале. Затим "транслирамо" троугао изнад дијагонале за једно поље наниже и једно поље улево. Другим речима, за упоређивања у горњем троуглу корак i се сада зове $i+1$. На пример, посматрајмо упоређивања 1 са 8 и 4 са 2 у првом кораку на слици 8. Прво упоређивање је на дијагонали, па се игнорише; друго је изнад дијагонале, па се сматра делом корака 2, уместо корака 1. Према томе, корак 2 састоји се од упоређивања 7 са 5, 6 са 1, и 4 са 2. Међутим, ово је регуларни парни корак са само n елемената. Корак 1 и сва упоређивања у којима учествује максимални елеменат се сада могу просто игнорисати, после чега су преостала упоређивања идентична са низом упоређивања која се изводе при извршавању алгоритма са n елемената (при чему је први корак непаран). Према индуктивној хипотези сортирање n елемената се изводи коректно; према томе, и сортирање свих $n+1$ елемената је коректно, а завршава се после $n+1$ корака. □

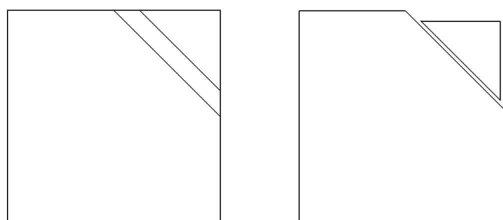


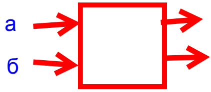
Рис. 9. Корак индукције у доказу исправности сортирања парно-непарним транспозицијама, Теорема 12.1.

До сада смо разматрали случај једног елемента по процесору. Претпоставимо сада да сваки процесор памти k елемената, и размотримо за почетак

случај само два процесора. Претпоставимо да је циљ да процесори међусобно размене елементе, тако да k најмањих елемената дођу у P_1, а k највећих у P_2. Јасно је да у најгорем случају сви елементи морају бити размењени, па је тада број размена елемената $2k$. Сортирање се може извести понављањем следећег корака све док је потребно: P_1 шаље свој највећи елемент у P_2, а P_2 свој најмањи елемент у P_1. Процес се завршава у тренутку кад је највећи елемент у P_1 мањи или једнак од најмањег елемента у P_2. Овај корак зове се *објединавање-раздвајање*. Користећи овај корак као основни у сортирању парно-непарним транспозицијама, добија се уопштење сортирања на случај више елемената по процесору.

Иако је овакав алгоритам сортирања оптималан за низ процесора, његова ефикасност је мала. Имамо n процесора који извршавају n корака; према томе, укупан број корака је n^2. Мала ефикасност $O(\log n/n)$ није изненађујућа, јер ефикасан алгоритам за сортирање мора имати могућност да замењује места међусобно удаљеним елементима. Низ процесора не омогућује такве замене. У следећем одељку приказаћемо специјализоване мреже за ефикасно сортирање.

12.4.2. Мреже за сортирање. Кад конструишемо ефикасан секвенцијални алгоритам, интересује нас само укупан број корака. У случају конструкције паралелних алгоритама, циљ је учинити кораке што независнијим. Посматрајмо сортирање обједињавањем (одељак 5.3.3). Два рекурзивна позива су потпуно независна и могу се извршити паралелно. Међутим, обједињавање као део алгоритма извршава се на секвенцијални начин. У излазни вектор се i-ти елемент ставља тек кад је тамо већ стављено првих $i-1$ елемената. Ако нам пође за руком паралелизација обједињавања, онда ћемо моћи да паралелизујемо и сортирање обједињавањем.



Описаћемо сада другачији алгоритам обједињавања, заснован на техници декомпозиције. Претпоставимо због једноставности да је n степен двојке. Нека су $a_1, a_2, \ldots, a_n$; $b_1, b_2, \ldots, b_n$ два сортирана низа које треба обединити, и нека је $x_1, x_2, \ldots, x_{2n}$ резултат њиховог обједињавања. Специјално је, на пример, $x_1 = \min\{a_1, b_1\}$. Потребно је обединити различите делове ових низова паралелно, тако да њихово комплетно обједињавање буде једноставно. Циљ се може постићи поделом два низа на по два дела — са непарним, односно парним индексима. Сваки део се обједињава са одговарајућим делом другог низа, а онда се комплетира обједињавање. Нека је $o_1, o_2, \ldots, o_n$ обједињени редослед непарних низова $a_1, a_3, \ldots, a_{n-1}$; $b_1, b_3, \ldots, b_{n-1}$, а нека је $e_1, e_2, \ldots, e_n$ обједињени редослед парних низова $a_2, a_4, \ldots, a_n$; $b_2, b_4, \ldots, b_n$. Очигледно је $x_1 = o_1$ и $x_{2n} = e_n$. Остатак обједињавања се такође лако изводи, што следи из следеће теореме.

Теорема 12.2. *У складу са уведеним ознакама, за* $i = 1, 2, \ldots, n-1$ *важи* $x_{2i} = \min\{o_{i+1}, e_i\}$ *и* $x_{2i+1} = \max\{o_{i+1}, e_i\}$.

Доказательство. Због једноставности се под парним, односно непарним елементима подразумевају елементи низова a_i, b_i са парним, односно непарним

индексом. Размотримо елеменат e_i. Пошто је e_i i-ти елеменат у сортираном редоследу парних низова, e_i је веће или једнако од *бар* i парних елемената у оба низа. Међутим, сваком парном елементу који је $\leq e_i$ одговара још један непарни елеменат који је такође $\leq e_i$ (јер смо пошли од два сортирана низа). Према томе, e_i је веће или једнако од бар $2i$ елемената из оба низа. Другим речима, установили смо да је $e_i \geq x_{2i}$. На сличан начин, o_{i+1} је веће или једнако од бар $i+1$ непарних елемената, из чега следи да је веће или једнако од бар $2i$ елемената укупно (за сваки непаран елеменат *сем првог*, који је $\leq o_{i+1}$, постоји још један парни елеменат који је $\leq o_{i+1}$). Према томе, важи и неједнакост $o_{i+1} \geq x_{2i}$. Специјално, за $i = n-1$ добија се $e_{n-1} \geq x_{2n-2}$ и $o_n \geq x_{2n-2}$, па пошто је $e_n = x_{2n}$, тврђење теореме је тачно за $i = n-1$. Стављајући даље у горње неједнакости редом $i = n-2, n-3, \ldots, 1$, добијамо да је тврђење теореме тачно и за остале вредности i, чиме је завршен доказ теореме. □

Важна последица Теореме 12.2 је да се комплетно обједињавање низова $o_1, o_2, \ldots, o_n$ и $e_1, e_2, \ldots, e_n$ извршава у једном паралелном кораку. Остатак посла извршавају рекурзивни позиви описаног алгоритма. Конструкција паралелног алгоритма следи директно из теореме. Слика 10 илуструје рекурзивну конструкцију обједињавања, а слика 11 показује комплетно сортирање, које се зове **сортирање парно-непарним обједињавањем**. Правоугаоници на левој страни слике 11, означени са "$n/2$ сортирање", представљају рекурзивне копије комплетне мреже за сортирање, које сортирају по $n/2$ бројева.

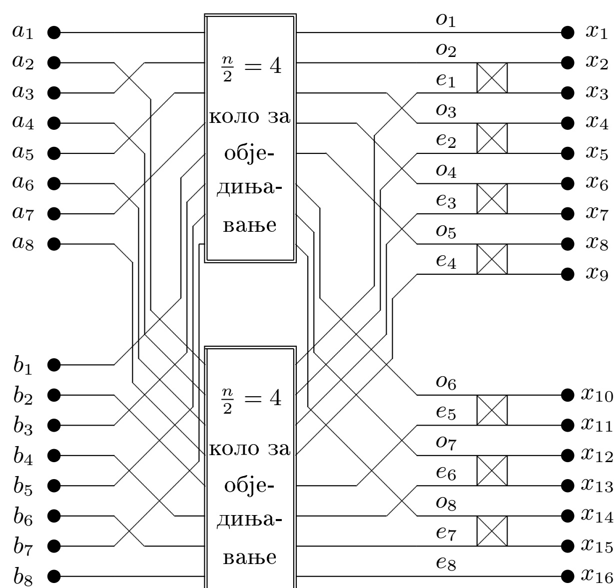


Рис. 10. Коло за парно–непарно обједињавање два низа од по $n = 8$ елемената.

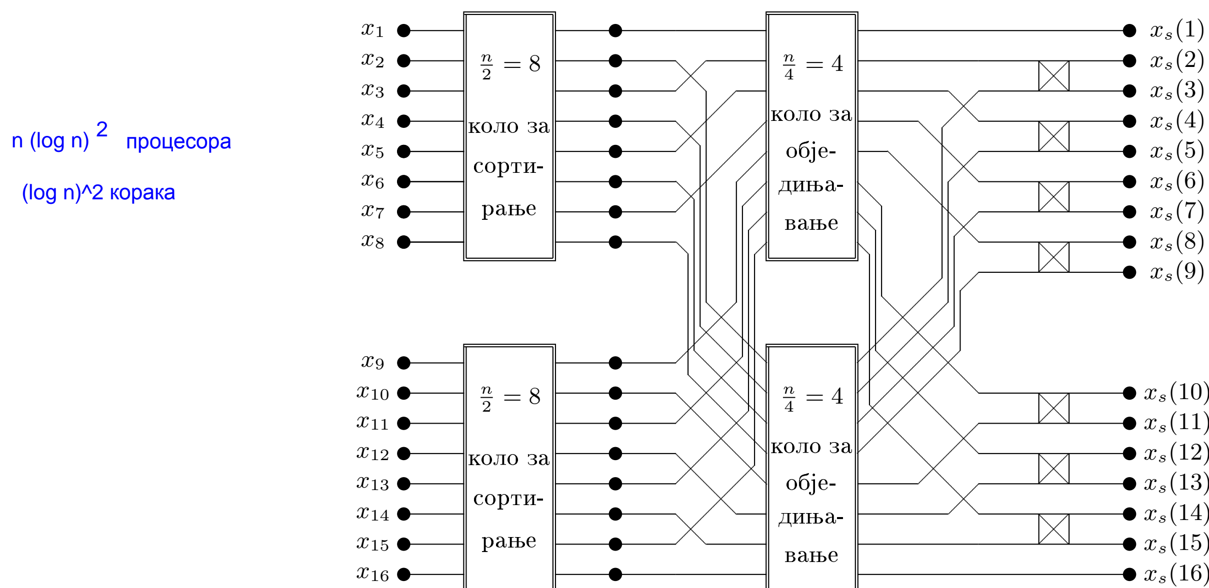
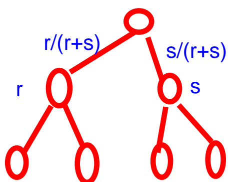
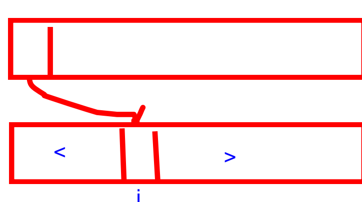


Рис. 11. Пример сортирања 16 бројева помоћу кола за сортирање парно-непарним обједињавањем.

Сложеност. Диференцна једначина за укупан број корака $T_M(n)$ за обједињавање је $T_M(2n) = 2T_M(n)+n-1$, $T_M(1) = 1$. Из тога следи да је укупан број упоређивања $O(n \log n)$, у поређењу са секвенцијалним алгоритмом који захтева само $O(n)$ корака. Дубина рекурзије, која одговара паралелном времену, је $O(\log n)$. Диференцна једначина за укупан број корака $T_S(n)$ за сортирање парно–непарним обједињавањем је $T_S(2n) = 2T_S(n) + O(n \log n)$, $T_S(2) = 1$. Њено решење је $T_S(n) = O(n \log^2 n)$ (видети нпр. анализу сложености алгоритма из одељка 7.5.2). Мрежа садржи по n процесора у свакој "колони", а њена дубина (тј. број колона) је $O(\log^2 n)$, па је укупан број процесора у мрежи $O(n \log^2 n)$. Приметимо да би се истих n процесора могли користити у свим колонама, али они би тада морали бити скоро потпуно исповезивани. Једини тип израчунавања у мрежи је упоређивање и евентуална замена, па су потребни једноставни процесори који се састоје од компаратора са два улаза и два излаза.



12.4.3. Налажење k-тог најмањег елемента на стаблу. Претпоставимо сада да је мрежа рачунара комплетно бинарно стабло висине $h-1$ са $n = 2^{h-1}$ листова, односно $2^h - 1$ процесора, придружених чворовима стабла. Улаз је низ $x_1, x_2, \ldots, x_n$, при чему се у почетном тренутку x_i чува у листу i. Рачунари у облику стабла користе се нпр. у вези са обрадом слика, при чему листови одговарају елементима улазног низа а алгоритми који их обрађују



i<k
i=k
i>k

1. избор пивота
2. дистрибуција пивота
3. одређивање ранга пивота
4. дистрибуција ранга пивота

су хијерархијски. Овде ћемо размотрити проблем налажења k-тог најмањег елемента.

Подсетимо се најпре секвенцијалног алгоритма за налажење k-тог најмањег елемента, описаног у одељку 5.4.2. Због једноставности претпоставимо да су елементи низа различити. Алгоритам је пробабилистички. У сваком кораку бира се случајни елеменат x као **пивот**. Ранг елемента x израчунава се упоредјивањем x са свим осталим елементима, а онда се елиминишу елементи који су мањи или већи од x, зависно од тога да ли је ранг мањи или већи од k. Извршавање алгоритма обуставља се у тренутку кад је ранг пивота k. Очекивани број итерација је $O(\log n)$, а очекивани број упоређивања је $O(n)$. Постоје три различите фазе у свакој итерацији алгоритма: (1) избор случајног елемента, (2) израчунавање његовог ранга, и (3) елиминација. Описаћамо ефикасну паралелну реализацију сваке од фаза.

Избор случајног елемента може се постићи организовањем турнира на стаблу. Сваки лист шаље свој број оцу, где се он "такмичи" са бројем брата. Победник у партији одређује се бацањем новчића. Број који је победио прелази у друго коло такмичења — иде увис по стаблу. Процес се наставља све дотле док корен стабла не изабере тачно један број као укупног победника (овај поступак је регуларан само у првој итерацији; размотрићемо касније како га треба поправити, да би радио исправно и кад су неки елементи елиминисани). Број–победник се затим шаље низ стабло до свих листова, ниво по ниво; у процесу слања учествује сваки чвор стабла, шаљући добијени број и левом и десном сину. Пошто сви листови сазнају који је број пивот, они своје бројеве могу да упореде са пивотом у једном паралелном кораку. Листови затим шаљу навише, свом оцу, јединицу, ако је њихов број мањи или једнак од пивота, односно нулу у противном. Ранг пивота је број јединица послатих навише. Сабирање бројева који се шаљу навише лако се изводи тако што сваки чвор сабира бројеве добијене од синова. После тога корен шаље наниже ранг пивота, и сваки лист може независно да установи да ли треба елиминише свој број. Укупно постоје четири "таласа" комуникације у свакој итерацији: (1) уз стабло да би се изабрао пивот, (2) низ стабло, да би се пивот послао до листова, (3) уз стабло, да би се израчунао ранг пивота, и (4) низ стабло се шаље ранг пивота.

После прве итерације појављује се један проблем; пошто су неки елементи елиминисани, такмичари више нису равноправни. Посматрајмо, на пример, екстремни случај кад су у једној половини стабла елиминисани сви листови сем једног. Преостали елеменат у тој половини стабла "провући" ће се до "финала" без икаквог такмичења, а онда ће бити изабран са вероватноћом 1/2, док је вероватноћа избора осталих елемената много мања. Волели бисмо да очувамо униформност расподеле вероватноћа приликом избора пивота у свакој итерацији. Униформност се може очувати на следећи начин. Процесори, чији су бројеви елиминисани у претходним колима, шаљу навише специјалну вредност **nil**, коју сваки елеменат побеђује. Сваком елементу који учествује у такмичењу придружује се бројач, чија је почетна вредност 1. Бројач показује колико је

стварних "противника" учествовало у делу турнира, из кога је елеменат изашао као победник (односно број нелиминисаних елемената у подстаблу чвора, у коме се налази елеменат). Кад елеменат победи у партији у неком чвору стабла, он се упућује навише, а вредност његовог бројача повећава се за вредност бројача његовог пораженог противника (као у филму "Горштак"). Партије се одигравају са "несиметричним" новчићем, код кога је однос вероватноћа нуле и јединице једнак односу вредности бројача два такмичара. Ако су бројачи играча p, односно q, онда први играч побеђује са вероватноћом $p/(p+q)$, а други са вероватноћом $q/(p+q)$. На пример, ако је у првој партији x победио играча y, а z прошао даље без борбе, онда бројачи елемената x, односно z имају вредности 2, односно 1. У игри x против z користи се новчић који је наклоњен елементу x, односно додељује му предност у односу $2:1$. Резултат је да z има вероватноћу $1/3$ победе у овој партији, а x и y са вероватноћом $(1/2)\cdot(2/3) = 1/3$ побеђују у обе своје партије. Индукцијом се показује да овакав процес избора обезбеђује да у сваком колу сви елементи се једнаком вероватноћом могу бити изабрани за пивот.

Сложеност. Број паралелних корака у свакој итерацији једнак је четворострукој висини стабла. Пошто овај алгоритам елиминише елементе на потпуно исти начин као и секвенцијални алгоритам, очекивани број итерација је $O(\log n)$. Према томе, очекивано време извршавања је $O(\log^2 n)$.

12.4.4. Множење матрица на дводимензионалној мрежи. Мрежа рачунара коју ћемо сада размотрити је дводимензионална мрежа $n\times n$. Процесор $P[i,j]$ налази се у пресеку i-те врсте и j-те колоне, и повезан је са процесорима $P[i,j+1]$, $P[i+1,j]$, $P[i,j-1]$ и $P[i-1,j]$. Претпоставља се да су супротни крајеви мреже међусобно повезани, односно да мрежа личи на торус. Тако, на пример, $P[0,0]$ је повезан са $P[0,n-1]$ и $P[n-1,0]$, поред $P[0,1]$ $P[1,0]$, видети слику 12. Другим речима, сва сабирања и одузимања индекса врше се по модулу n. Алгоритам који приказујемо је симетричнији и елегантнији уз ову претпоставку; његово време извршавања једнако је (до на константни фактор) времену извршавања на обичној мрежи (оној која није пресавијањем повезана као торус).

Проблем. Дате су две $n\times n$ матрице A и B, тако да су њихови елементи $A[i,j]$ и $B[i,j]$ у процесору $P[i,j]$. Израчунати $C = AB$, тако да елеменат $C[i,j]$ буде у процесору $P[i,j]$.

Користићемо обичан алгоритам за множење матрица. Проблем је преместити податке тако да се прави бројеви нађу на правом месту у правом тренутку. Посматрајмо елеменат $C[0,0] = \sum_{k=0}^{n-1} A[0,k]\cdot B[k,0]$, који је једнак производу прве врсте матрице A и прве колоне матрице B; Редни бројеви врста и колона су при оваквом означавању за један већи од њихових индекса. Волели бисмо да број $C[0,0]$ буде израчунат у процесору $P[0,0]$. Ово се може постићи цикличким померањем прве врсте A улево, и истовременим цикличким померањем прве колоне B увис, корак по корак. У првом кораку $P[0,0]$ садржи $A[0,0]$ и $B[0,0]$

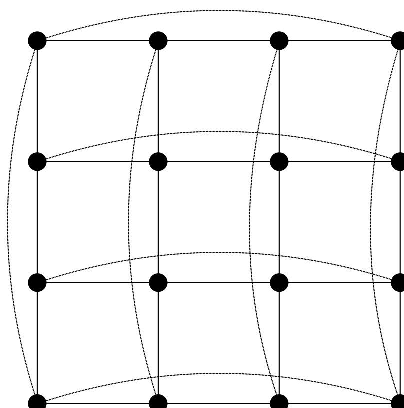


Рис. 12. Дводимензионална пресавијена мрежа.

и израчунава њихов производ; у другом кораку $P[0,0]$ добија $A[0,1]$ (од десног суседа) и $B[1,0]$ (од суседа испод себе) и њихов производ додаје парцијалној суми, итд. Вредност $C[0,0]$ се тако израчунава после n корака.

Проблем је обезбедити да сви процесори обављају исти овакав посао, а сви морају да деле податке међусобно. Потребно је да податке преуредимо тако да не само $P[0,0]$, него и сви остали процесори добију све потребне податке. Идеја је да се подаци у матрицама испремештају на такав начин, да сваки процесор у сваком кораку добије два броја чији производ треба да израчуна. Битан је дакле почетни распоред података. Преуредићемо их тако да на почетку процесор $P[i,j]$ има елементе $A[i,i+j]$ и $B[i+j,j]$, односно да други индекс елемента A буде једнак првом индексу елемента B (при чему се, као што је речено, индекси рачунају по модулу n). Пошто се формира овакав распоред, сваки корак се састоји од истовременог цикличког померања врста A и колона B, чиме $P[i,j]$ добија елементе $A[i,i+j+k]$ и $B[i+j+k,j]$, $0 \leq k \leq n-1$, управо оне елементе који су му потребни. До почетног распореда може се доћи цикличким померањем врсте A са индексом i за i места улево, а колоне B са индексом i за i места навише, $i = 0,1,\ldots,n-1$. Алгоритам је приказан на слици 13. У табели 1 приказано је формирање почетног распореда елемената матрица за $n = 4$. Лева страна показује почетно стање података, а десна њихов размештај после извршавања почетних цикличких померања.

Сложеност. Почетна циклична померања врста A трају $n/2$ паралелних корака (кад број померања постане већи од $n/2$, померања се изводе у супротном смеру — удесно уместо улево); исто важи и за почетна померања колона B. У наредних n корака изводе се у сваком процесору израчунавања и померања. Ти кораци могу се извршити паралелно. Укупно време извршавања је $O(n)$. Ефикасност алгоритма је $O(1)$, ако упоређујемо паралелни алгоритам са

Алгоритам Množenje_matrica(A, B);

Улаз: A и B ($n \times n$ матрице).

Излаз: C (производ AB).

begin

  for $i := 0$ **to** $n-1$ **do in parallel**

    циклички помери улево врсту i матрице A за i места;

    {односно, изврши $A[i,j] := A[i,(j+1) \bmod n]$ i пута}

  for $j := 0$ **to** $n-1$ **do in parallel**

    циклички помери навише колону j матрице B за j места;

    {односно, изврши $B[i,j] := B[(i+1) \bmod n, j]$ j пута}

  {подаци су сада на жељеним почетним позицијама}

  for све парове i, j **do in parallel**

    $C[i,j] := A[i,j] \cdot B[i,j]$;

  for $k := 1$ **to** $n-1$ **do**

    for све парове i, j **do in parallel**

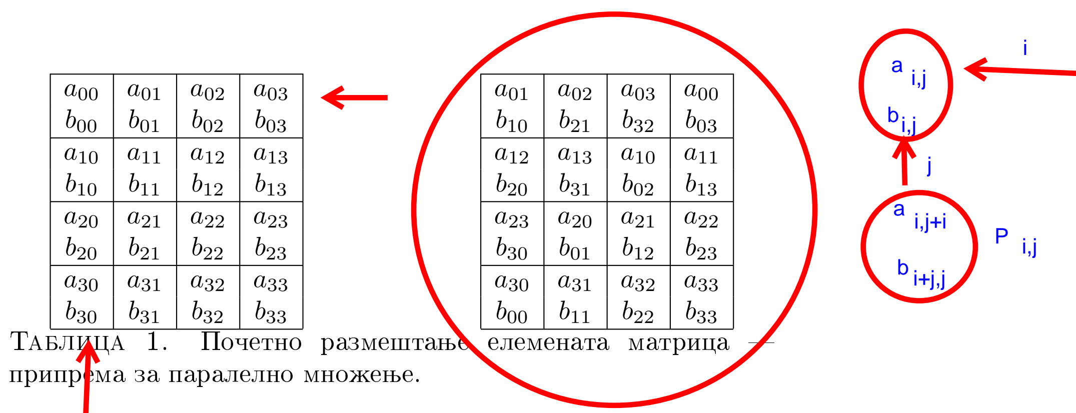
      $A[i,j] := A[i,(j+1) \bmod n]$;

      $B[i,j] := B[(i+1) \bmod n, j]$;

      $C[i,j] := C[i,j] + A[i,j] \cdot B[i,j]$

end

Рис. 13. Алгоритам за паралелно множење матрица на мрежи.



a_{00} b_{00}	a_{01} b_{01}	a_{02} b_{02}	a_{03} b_{03}
a_{10} b_{10}	a_{11} b_{11}	a_{12} b_{12}	a_{13} b_{13}
a_{20} b_{20}	a_{21} b_{21}	a_{22} b_{22}	a_{23} b_{23}
a_{30} b_{30}	a_{31} b_{31}	a_{32} b_{32}	a_{33} b_{33}

a_{01} b_{10}	a_{02} b_{21}	a_{03} b_{32}	a_{00} b_{03}
a_{12} b_{20}	a_{13} b_{31}	a_{10} b_{02}	a_{11} b_{13}
a_{23} b_{30}	a_{20} b_{01}	a_{21} b_{12}	a_{22} b_{23}
a_{30} b_{00}	a_{31} b_{11}	a_{32} b_{22}	a_{33} b_{33}

Таблица 1. Почетно размештање елемената матрица — припрема за паралелно множење.

обичним секвенцијалним множењем матрица сложености $O(n^3)$. Ефикасност је асимптотски мања, ако паралелни алгоритам упоређујемо са нпр. Штрасеновим алгоритмом.

12.5. Систолички алгоритми

Систоличка архитектура личи на покретну траку у фабрици. Процесори су обично распоређени на врло правилан начин (најчешће у облику једнодимензионалног или дводимензионалног поља), а подаци се кроз њих ритмички померају. Сваки процесор извршава једноставне операције са подацима које је добио у претходном кораку, а своје резултате дотура следећој "станици". Сваки процесор може да садржи малу локалну меморију. Улази се најчешће "утискују" у систем један по један, уместо да се сви одједном упишу у неке меморијске локације. Предност систоличке архитектуре је ефикасност, и у хардверском погледу (процесори су специјализовани и једноставни) и у погледу брзине (минимизиран је број приступа меморији). Као и на покретној траци, кључно је да се избегне потреба за довлачењем алата и материјала за време рада. Све што је потребно за извршавање операције, долази покретном траком. Озбиљан недостатак оваквих рачунара је недовољна флексибилност. Систоличке архитектуре су ефикасне само за одређене алгоритме. Размотрићемо два примера систоличких алгоритама.

12.5.1. Множење матрице и вектора. Започињемо са једноставним алгоритмом, који ћемо затим користити за развој компликованијег алгоритма.

Проблем. Израчунати производ $x = Ab$ матрице A димензије $m \times n$ и вектора b дужине n.

Систем се састоји од n процесора, тако да процесор P_i додаје парцијалној суми члан у коме је чинилац b_i, i-та компонента вектора b. Кретање података и операције које извршава сваки од процесора приказани су за $n = m = 4$ на слици 14. Претпостављамо да се вектор b налази у одговарајућим процесорима (или је у њих убачен корак по корак). Резултати се акумулирају током кретања слева удесно кроз процесоре. Излазне промењиве x_i на почетку имају вредност 0. У првом кораку се x_1 $(= 0)$ заједно са a_{11} убацује у P_1, а сви остали улази напредују један корак на путу ка процесорима. Процесор P_1 израчунава $x_1 + a_{11} \cdot b_1$, и резултат прослеђује удесно. У другом кораку P_2 прима слева $x_1 = a_{11} \cdot b_1$ и одозго a_{12}, па израчунава $x_1 + a_{12} \cdot b_2$, резултат прослеђује удесно, итд. У i-том кораку израчунавања x_1 процесор P_i прима слева парцијални резултат $x_1 = \sum_{k=1}^{i-1} a_{1k} b_k$, одозго одговарајући елеменат матрице a_{1i}, а одговарајућу координату b_i било из локалне меморије (као на слици), било одоздо. Процесор P_i израчунава вредност израза $x_1 + a_{1i} b_i$ и предаје је даље, удесно. У тренутку напуштања низа процесора, после n корака, x_1 очигледно садржи жељену вредност. Израчунавање x_2 прати " у стопу" израчунавање x_1 и завршава се у $(n + 1)$-ом кораку. Другим речима, израчунавања x_1 и x_2 су временски скоро потпуно преклопљена. Уопште, израчунати елеменат x_j појављује се на излазу после $n+j-1$ корака, $j = 1, 2, \ldots, m$, а комплетан производ израчунава се после $m + n - 1$ корака.

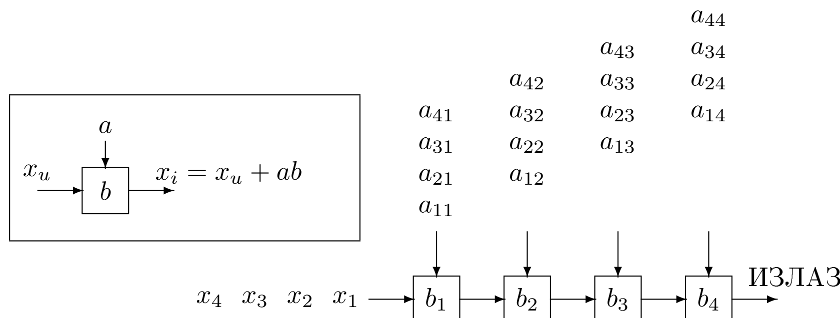


Рис. 14. Множење вектора матрицом.

Основни проблем при конструкцији систоличких алгоритама је организација кретања података. Сваки податак мора се наћи на правом месту у правом тренутку. У овом примеру то је постигнуто увођењем кашњења, тако да почетак i-те колоне матрице A стиже у процесор P_i у кораку i. Овај пример је једноставан, јер се сваки елеменат матрице A користи само једном. Кад се иста вредност користи више пута, много је компликованије организовати њено кретање, што ћемо видети у следећем примеру.

12.5.2. Израчунавање конволуције. Нека су $x = (x_1, x_2, \dots, x_n)$ и $w = (w_1, w_2, \dots, w_k)$ два реална вектора, при чему је $k < n$. **Конволуција** вектора x и w је вектор $y = (y_1, y_2, \dots, y_{n+1-k})$, са координатама

$$y_i = w_1 x_i + w_2 x_{i+1} + \cdots w_k x_{i+k-1}, \quad i = 1, 2, \dots, n+1-k.$$

Проблем. Израчунати конволуцију y вектора x и w.

Проблем израчунавања конволуције може се свести на проблем израчунавања производа матрице и вектора на следећи начин:

$$\begin{pmatrix} x_1 & x_2 & x_3 & \cdots & x_k \\ x_2 & x_3 & x_4 & \cdots & x_{k+1} \\ x_3 & x_4 & x_5 & \cdots & x_{k+2} \\ \dots & \dots & \dots & \dots & \dots \\ x_{n+1-k} & x_{n+2-k} & x_{n+3-k} & \cdots & x_n \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ \cdots \\ w_k \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \cdots \\ y_{n+1-k} \end{pmatrix}$$

Систолички алгоритам који решава овај проблем може се добити као специјални случај општијег алгоритма за израчунавање производа матрице и вектора (видети претходни проблем, слика 14). Ово је приказано на слици 15. Приметимо да се x_i истовремено користи дуж целог низа процесора (сем првих $k-1$ вредности x_i, које се користе само у почетном делу низа). Због тога је потребна линија за простирање. Приказаћемо сада решење проблема конволуције без простирања.

Процесори на слици 15 примају два улаза, а дају само један излаз. Употребићемо сада процесоре који примају улазе из два смера, и шаљу излаз у

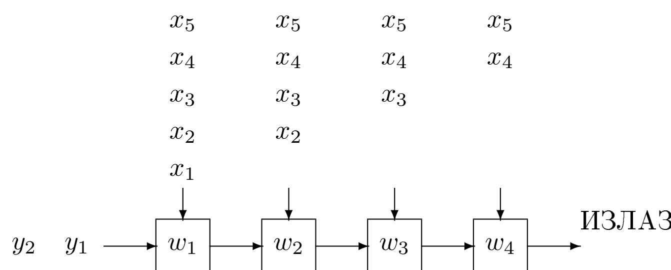


Рис. 15. Конволуција са простирањем.

два смера. Идеја је померати вектор x слева удесно, а вектор w здесна улево. Резултат y акумулира се у процесорима. Кретање података треба тако подесити да се одговарајуће координате w и x сретну тамо где треба да буду помножене. Проблем је у томе што, кад се два вектора крећу један према другом, онда је брзина једног вектора у односу на други двоструко већа. Последица ове чињенице је да би један елеменат вектора x пропустио половину елемената вектора w, и обрнуто. Решење је померати векторе *двоструко мањом брзином.* Улаз слева је дакле "x_1, ништа, x_2, ништа, ...", а здесна исто то за вектор w. Решење је приказано на слици 16 (црне тачке одговарају одсутним подацима).

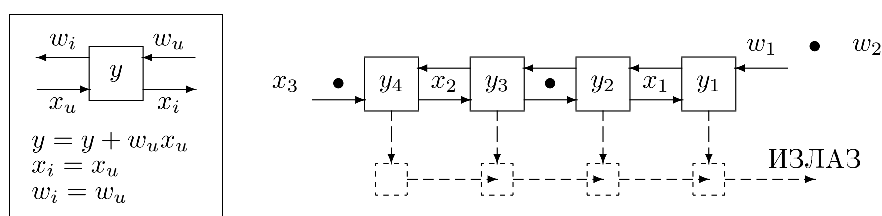


Рис. 16. Конволуција помоћу двосмерног низа.

Препуштамо читаоцу да се увери да сваки процесор P_i на крају израчунава вредност y_i. Кад w_k напусти P_i, израчуната је коначна вредност y_i, па се подаци могу "изручити" из низа процесора путем назначеним испод низа процесора на слици 16. Недостатак кретања података двоструко мањом брзином је у томе што израчунавање траје двоструко дуже.

12.6. Резиме

Пошто је паралелне алгоритме компликованије конструисати од секвенцијалних, корисно је што више употребљавати готове блокове. Један од таквих моћних блокова је алгоритам за паралелно израчунавање префикса, који је чак предлаган за примитивну машинску инструкцију. Тешко је у овом тренутку проценити које ће од разматраних паралелних архитектура (односно да ли ће

нека од њих) доминирати у будућности. Због тога је важно идентификовати технике пројектовања алгоритама заједничке за више модела. Анализирали смо четири такве технике: *удвостручавање* (одређивање рангова елемената листе и друге операције са повезаним листама), *паралелну варијанту декомпозиције* (сабирање, паралелно израчунавање префикса, сортирање), *преклапање* (код систоличких алгоритама), и *технику Ојлеровог обиласка* (која је корисна код мноштва алгоритама за рад са стаблима, односно графовима).

Задаци

12.1. Конструисати алгоритам за сабирање k n-битних бинарних бројева. Време извршавања алгоритма на nk процесора треба да буде $O(\log n \log k)$.

12.2. Доказати да је за извршавање алгоритма *Rangovi* (слика 12.3) довољан модел EREW.

12.3. Модификовати CRCW алгоритам за налажење максимума (одељак 12.3.2.2) тако да ради и ако дати бројеви нису сви различити, а да време извршавања и даље буде $O(1)$.

12.4. У вектору A дужине n треба вредност $A[1]$ прекопирати на све остале локације. (а) Конструисати EREW алгоритам за решавање овог проблема, сложености $T(n,n) = O(\log n)$. (б) Конструисати EREW алгоритам ефикасности $O(1)$ за решавање овог проблема, са временом извршавања $O(\log n)$.

12.5. Конструисати алгоритам за рачунар са заједничком меморијом и n процесора, модел EREW, који на локацију $A[i]$ вектора A, смешта вредност $A[1]+i$, $1 < i \leq n$.

12.6. Паралелизовати Хорнерову шему за израчунавање вредности полинома степена n на моделу EREW, тако да сложеност, односно ефикасност алгоритма, буду редом $O(\log n)$, односно $O(1)$.

12.7. Конструисати алгоритам за израчунавање дисјункције n Булових променљивих на моделу CRCW за време $O(1)$.

12.8. Конструисати коло за решавање проблема паралелног префикса, дубине $O(\log n)$ са $O(n \log n)$ чворова, тако да сваки процесор (елеменат кола) има највише два улаза и два излаза.

12.9. Конструисати алгоритам за сортирање низа различитих бројева $x_1, x_2, \ldots, x_n$, временске сложености $O(\log n)$, на паралелном CREW рачунару са заједничком меморијом и довољним бројем процесора.

12.10. Конструисати CRCW алгоритам за обједињавање два сортирана низа A и B у један сортирани низ, за време $O(1)$. На располагању је неограничен број процесора и неограничен меморијски простор.

12.11. **Кронекеров производ** вектора $A[0]$, $A[1], \ldots, A[n-1]$ и $B[0]$, $B[1], \ldots, B[m-1]$ се дефинише као вектор $C[0]$, $C[1], \ldots, C[nm-1]$ са елементима $C[km+r] = A[k]B[r]$, $0 \leq k \leq n-1$, $0 \leq r \leq m-1$ (другим речима, C садржи $n \times m$ матрицу чији (i,j) елеменат је $A[i]B[j]$). Конструисати паралелни EREW алгоритам за израчунавање C помоћу p процесора за време $O(mn \log p/p)$. На располагању је меморијски простор $O(mn)$ (може се претпоставити да су n, m и p степени двојке).

12.12. Задат је вектор, чији су елементи слогови. Сваки слог садржи неке податке и Булову променљиву *Oznaka*. Циљ је препаковати на почетак вектора све слогове, којима поље *Oznaka* има вредност 1. Редослед премештених слогова треба да остане непромењен. Конструисати EREW алгоритам сложености $T(n,n) = O(\log n)$ за решавање овог проблема.

12.13. Дата је повезана листа, чији су елементи произвољним редоследом смештени у вектору. Нека је $F[i]$ Булова променљива придружена i-том елементу листе. Конструисати EREW алгоритам сложености $T(n,n) = O(\log n)$ за формирање друге повезане листе, која се састоји само од оних елемената за које је $F[i] = 1$, и то истим редом којим се елементи појављују у полазној листи (овај алгоритам потребан је, на пример, за извршење долазне нумерације у одељку 12.3.5).

12.14. Дата је повезана листа, чији су елементи (произвољним редоследом) смештени у вектору. Конструисати ефикасан паралелни алгоритам за формирање повезане листе од истих елемената у обрнутом редоследу (не премештајући ни један елеменат). Може се претпоставити да је на располагању довољан меморијски простор за допунске показиваче.

12.15. На мрежи рачунара у облику бинарног стабла, при чему се у i-том листу налази број x_i, $1 \leq i \leq n$, треба решити проблем паралелног префикса (по извршењу алгоритма лист i треба да садржи суму $x_1 \star x_2 \star \cdots \star x_i$, где је $\star$ асоцијативна операција која се извршава у једном кораку). Време извршавања треба да буде $O(\log n)$.

12.16. Како треба модификовати алгоритам за налажење k-тог највећег броја међу бројевима $x_1, x_2, \ldots, x_n$ на стаблу (одељак 12.4.3), тако да ради исправно и ако бројеви $x_1, x_2, \ldots, x_n$ нису сви различити?

12.17. Процесори, њих n, повезани су у прстен. Процесор са индексом i чува $(i+1)$-у врсту дате $n \times n$ матрице A, и i-ти елеменат датог вектора $b = (b_0, b_1, \ldots, b_{n-1})^T$, $0 \leq i \leq n-1$. Потребно је израчунати производ $x = Ab$, тако да на крају елеменат $x_i = \sum_{k=0}^{n-1} a_{ik} b_k$ буде смештен у процесору P_i, $0 \leq i \leq n-1$. Конструисати алгоритам временске сложености $O(n)$ који решава овај проблем.

12.18. Решити задатак 12.17 ако на почетку процесор P_i садржи i-ту колону матрице и компоненту b_i.

12.19. Нека је $n \times n$ матрица $A = (a_{ij})$ смештена у квадратној мрежи истих димензија, тако да процесор $P[i,j]$ чува елеменат a_{ij}. Конструисати EREW алгоритам за транспоновање матрице A, временске сложености $O(n)$.

12.20. У квадратној мрежи од $n \times n$ процесора сваки процесор садржи по једну тачку $n \times n$ црно-беле слике (бит 1 одговара црној, а 0 белој боји). Потребно је пронаћи *повезане компоненте* слике (две црне тачке припадају истој компоненти ако постоји пут од једне до друге преко црних тачака; сваке две узастопне тачке на путу су хоризонтално или вертикално суседне — не дијагонално). Све тачке у истој компоненти треба да буду означене јединственом ознаком компоненте. На почетку је тачка (i,j) означена бројем $in + j$, $0 \leq i,j \leq n-1$. Размотримо следећи поступак за означавање сваке компоненте најмањом од ознака тачака које јој припадају. У сваком кораку процесор, који садржи црну тачку, проверава ознаке суседне четири тачке. Ако било која од њих има мању ознаку, онда процесор ту вредност уписује као ознаку своје тачке. Доказати да описани алгоритам после највише n^2 корака коректно означава све компоненте, али да се за неке улазе означавање не може извршити за мање од cn^2 корака, где је $c > 0$ константа. Како се може изабрати тренутак за прекид извршавања описаног поступка?

12.21. Нека је A квадратна матрица реда n смештена у квадратној мрежи на уобичајени начин. Нека су σ и π две пермутације скупа $\{0, 1, \ldots, n-1\}$, такве да је σ смештена у првој, а π у другој врсти мреже. Конструисати алгоритам сложености $O(n)$ за пермутовање врста A према σ, а колона A према π (најпре врсте). Другим речима, врста i треба да буде премештена на позицију $\sigma(i)$, а колона j на позицију $\pi(j)$.

12.22. На располагању је 2^n процесора повезаних у n-димензионалну хиперкоцку. Конструисати алгоритам временске сложености $O(n)$ за налажење највећег од 2^n задатих елемената.

12.23. Конструисати алгоритам за решавање на n-димензионалној хиперкоцки. Сваки процесор чува један број. На крају сваки процесор треба да има израчунату вредност префикса свог елемента.

12.24. Показати како се може дводимензионална мрежа $2^k \times 2^m$ уметнути у $(k+m)$-димензионалу хиперкоцку.

12.25. Паралелни проблем трачева. Претпоставимо да $n = 2^k$ особа знају неке трачеве. У сваком кораку свака особа може да комуницира са неком другом особом (највише једном), и да са њом размени све трачеве које у том тренутку зна. Конструисати редослед комуницирања, такав да после $\log_2 n$ корака свака особа зна све трачеве.

Глава 13

Упутства и решења задатака

13.1. Математичка индукција

1.1. Тврђење је тачно за $n = 1, 2$; за $k \geq 3$ је

$$x^k - y^k = (x+y)(x^{k-1} - y^{k-1}) + xy(x^{k-2} - y^{k-2}),$$

па из претпоставке да тврђење важи за бројеве мање од n следи да оно важи и за n. Тиме је математичком индукцијом доказано да тврђење важи за сваки природан број n.

1.2. Израчунавши неколико првих збирова, наслућујемо да је тражени израз $\frac{1}{3}n(n+1)(n+2)$ (ако никако другачије, до овог израза се може доћи одређивањем коефицијената полинома трећег степена $a + bn + cn^2 + dn^3$, једнаког суми, тако да израз буде тачан за $n = 1, 2, 3, 4$). Доказ индукцијом је лакши део посла. За $n = 1$ је израз тачан ($1 \cdot 2 = \frac{1}{3} \cdot 1 \cdot 2 \cdot 3$), а ако је тачан за неко n, онда је

$$\sum_{i=1}^{n+1} i(i+1) = \tfrac{1}{3}n(n+1)(n+2) + (n+1)(n+2) = \tfrac{1}{3}(n+1)(n+2)(n+3),$$

односно израз је тачан и за $n + 1$ сабирака.

1.3. Индукцијом; $(-1)^{k-1}k(k+1)/2 + (-1)^k(k+1)^2 = (-1)^k(k+1)(k+2)/2$.

1.4. Сваки елеменат $a \in A$ може се написати у облику $2^{s(a)}o(a)$, где је $o(a)$ непаран број, "непарна основа" броја a. Пошто непарних природних бројева мањих од $2n$ има n, а A има $n+1$ бројева, у A постоје нека два броја a и b са једнаким непарним основама, $o(a) = o(b)$; ако је нпр. $s(a) < s(b)$, онда $a|b$.

1.5. Доказ се може извести индукцијом по n. Ако неједнакост из задатка узмемо као индуктивну хипотезу, онда је $(a+b)^{n+1} = (a+b)(a+b)^n \leq (a+b)2^{n-1}(a^n + b^n)$. Доказ би био завршен ако бисмо доказали да је $(a+b)2^{n-1}(a^n + b^n) \leq 2^n(a^{n+1} + b^{n+1})$. Ову је пак неједнакост лако доказати: после множења и сређивања добија се да је она еквивалентна очигледној неједнакости $(a^n - b^n)(a - b) \geq 0$.

1.6. Ако врсте Паскаловог троугла нумеришемо почевши од 0, онда је сума бројева у i-тој врсти једнака $S_i = 2^i$. За $i = 0$ тврђење је тачно. У збиру бројева $(i+1)$-е врсте се, после замене тих бројева збировима по два броја из претходне врсте, сваки број из i-те врсте појављује тачно два пута, па је $S_{i+1} = 2^{i+1}$.

1.7. За $n = 2$ је $\frac{1}{3} + \frac{1}{4} = \frac{7}{12} > \frac{13}{24}$. Ако је неједнакост тачна за неко n, онда је

$$\begin{aligned}\sum_{i=n+2}^{2(n+1)} \tfrac{1}{i} \quad &= \quad \sum_{i=n+1}^{2n} \tfrac{1}{i} - \tfrac{1}{n+1} + \tfrac{1}{2n+1} + \tfrac{1}{2n+2} > \\ &> \quad \tfrac{13}{24} + \tfrac{1}{2n+1} - \tfrac{1}{2n+2} = \tfrac{13}{24} + \tfrac{1}{(2n+1)(2n+2)} > \tfrac{13}{24},\end{aligned}$$

па је неједнакост тачна и за $n + 1$. Према томе, неједнакост је тачна за свако $n > 1$.

1.8. Нека је $\sum_{i=1}^{n} \frac{1}{i} = k_n/m_n$, и нека је s_n највећи експонент степена двојке којим је дељив именилац m_n. Другим речима, $m_n = 2^{s_n}o_n$, где је o_n непаран број. Израчунавши неколико првих збирова, запажамо да су бројеви k_n непарни, и да је s_n неопадајућа функција од n, једнака највећем експоненту степена двојке којим је дељив неки број $\leq n$, односно

$s_n = \lfloor \log_2 n \rfloor$. Доказаћемо ово тврђење индукцијом по n. Тврђење је тачно за $n = 1, 2$. Ако је тачно за неко n, онда је

$$\sum_{i=1}^{n+1} \frac{1}{i} = \sum_{i=1}^{n} \frac{1}{i} + \frac{1}{n+1} = \frac{k_n}{m_n} + \frac{1}{n+1} = \frac{m_n + k_n(n+1)}{m_n(n+1)} = \frac{2^{s_n} o_n + k_n 2^x y}{2^{s_n + x} y o_n},$$

где је $n + 1 = 2^x y$, а y је непаран број.

Ако је $n + 1$ степен двојке, односно $y = 1$, онда је $s_n = \lfloor \log_2 n \rfloor = x - 1$, па горњи разломак постаје $(o_n + 2k_n)/(2^x o_n)$, односно $k_{n+1} = o_n + 2k_n$ је непаран број, а највећи експонент степена двојке којим је дељив именилац $m_{n+1} = 2^x o_n$ је $x = \lfloor \log_2(n+1) \rfloor$.

У противном, ако $n + 1$ није степен двојке, односно за неко r важи $2^r < n + 1 < 2^{r+1}$, онда је $s_n = r$ и $x < r$ (јер између 2^r и 2^{r+1} нема бројева дељивих са 2^r), па је

$$\frac{k_{n+1}}{m_{n+1}} = \frac{2^{r-x} o_n + k_n y}{2^r y o_n},$$

односно бројилац $k_{n+1} = 2^{r-x} o_n + k_n y$ је непаран, $m_{n+1} = 2^r y o_n$ и $s_{n+1} = r = \lfloor \log_2(n+1) \rfloor$.

1.9. Згодно је тврђење формулисати на следећи начин: бројеви мањи од $5 \cdot 2^n$, $n \geq 0$, могу се представити у облику збира различитих бројева из овог низа. За $n = 0$ тврђење је тачно. Нека је тврђење тачно за бројеве мање од n, и нека је дат произвољан број a из интервала $[5 \cdot 2^{n-1}, 5 \cdot 2^n)$. Пошто је $a - 5 \cdot 2^{n-1} < 5 \cdot 2^n - 5 \cdot 2^{n-1} = 5 \cdot 2^{n-1}$, број $a - 5 \cdot 2^{n-1}$ може се представити у облику збира различитих бројева из овог низа, мањих од $5 \cdot 2^{n-1}$, тј. a се добија додавањем том збиру броја $5 \cdot 2^{n-1}$, различитог од свих претходних сабирака.

1.10. За $n = 3$ тврђење је тачно, јер су праве у општем положају. Претпоставимо да тврђење важи за произвољних n правих у општем положају, и нека је дато $n + 1$ правих у општем положају. Међу областима на које првих n правих деле раван постоји по индуктивној хипотези бар један троугао. Ако $(n+1)$-а права p не сече тај троугао, онда је доказ завршен — може се узети баш тај троугао. У противном, права p сече неке две странице тог троугла. Заједничко теме те две странице са њиховим пресечним тачкама са p чине тражени троугао.

1.11. Задатак се може преформулисати на следећи начин: дато је n кругова полупречника 1 у равни, тако да свака три имају бар једну заједничку тачку; доказати да је пресек свих n кругова непразан (тачка из пресека свих кругова је центар круга полупречника 1 који садржи центре свих n кругова — задате тачке у оригиналној формулацији). Ово тврђење ћемо појачати: показаћемо да оно важи за произвољних n конвексних скупова (тзв. Хелијева теорема). Доказ се изводи индукцијом по n.

База индукције је случај $n = 4$. Нека су дати (конвексни) скупови k_1, k_2, k_3, k_4, и нека је C_i нека тачка из пресека скупова из фамилије $\{k_1, k_2, k_3, k_4\} \setminus \{k_i\}$, $i = 1, 2, 3, 4$. Могући су следећи случајеви.

- Тачке C_i су темена конвексног четвороугла; тада пресек дијагонала тог четвороугла припада пресеку свих скупова k_i.
- Једна од тачака, нпр. C_1, припада троуглу који чине остале три тачке C_2, C_3, C_4; тада скуп k_1 садржи тачке C_2, C_3, C_4, а са њима и тачку, C_1, тачку троугла $C_2C_3C_4$, па је C_1 тражена тачка.

Нека тврђење важи за произвољних $n - 1$ конвексних скупова у равни, и нека је дато $n \geq 5$ произвољних конвексних скупова $k_1, k_2, \ldots, k_n$, таквих да је пресек било која три од њих непразан. Скупови $k'_i = k_i \cap k_n$, $i = 1, 2, \ldots, n-1$, задовољавају индуктивну хипотезу: конвексни су и пресек свака три је непразан:

$$k'_i \cap k'_j \cap k'_l = k_i \cap k_j \cap k_l \cap k_n \neq \emptyset,$$

јер је тачност тврђења за $n = 4$ доказана. Због тога им је пресек непразан, тј. постоји тачка $C \in k'_1 \cap k'_2 \cap \ldots \cap k'_{n-1} = k_1 \cap k_2 \cap \ldots \cap k_n$.

1.12. Решење се заснива на сличној идеји као и доказ теореме 1.7. Боје су 0, 1 и 2. Базни случај $n = 1$ је тривијалан. Новододата, n-та кружница са тетивом дели раван на три области. Областима ван кружнице не мењају се боје. Областима у кружници са једне (друге)

стране тетиве боја i замењује се бојом $i+1 \bmod 3$ ($i-1 \bmod 3$), $i=0,1,2$. На тај начин добија се исправно бојење са три боје и после додавања n-те фигуре.

1.13. Доказ се може извести индукцијом по броју области n на које мапа дели раван. За $n=1$ бојење је могуће — довољна је једна боја. Претпоставимо да је тврђење тачно за мапе са мање од n области и нека је дата мапа са n области. Пошто су сви чворови парног степена (имају паран број суседних чворова), свака грана је део границе неке области, јер не постоје чворови из којих води само једна грана. Посматрајмо спољашњу, бесконачну област. Уклањањем грана на граници спољашње области, добија се мапа са мањим бројем области, чији су сви чворови парног степена (граница бесконачне области за сваки свој чвор има две суседне гране), па се она по индуктивној хипотези може исправно обојити. Враћањем уклоњених грана се од бесконачне области одсецају области које су у полазној мапи биле суседне бесконачној области; њима се не мења боја. Те области су једини суседи полазне бесконачне области, која се због тога може обојити супротном бојом. На тај начин је индукцијом конструисано исправно бојење планарне мапе са n области, односно завршен је доказ индукцијом.

1.14. Доказ се изводи индукцијом по n. За $n=2$ је $d_1+d_2=2$, па мора бити $d_1=d_2=1$; стабло са два чвора степена један постоји: то су два чвора повезана граном. Претпоставимо да је тврђење тачно за $n-1$ бројева, и нека је дато $n \geq 3$ природних бројева $d_1, d_2, \ldots, d_n$ са сумом $2n-2$. Тада бар један од њих (нпр. d_i) мора бити једнак 1 (у противном би им сума била већа или једнака од $2n$), и бар један од њих (нпр. d_j) мора бити већи од 1 (у противном би им сума била мања или једнака од $n<2n-2$). Избацивши d_i из скупа и умањивши d_j за један, добијамо скуп за кога по индуктивној хипотези постоји стабло са тим степенима. Додавши у то стабло један нови чвор степена $d_i=1$ (лист), "прикачен" за чвор степена $d_j-1 \geq 1$, добијамо стабло са n чворова и степенима $d_1, d_2, \ldots, d_n$. Јасно је да решење овог проблема у општем случају није јединствено.

1.15. Ако са $f(n)$ означимо тражени број области, непосредно се израчунава неколико првих вредности $f(1)=1$, $f(2)=2$, $f(3)=4$, $f(4)=8$, $f(5)=16$. Диференцна једначина за низ $f(n)$ може се извести тако да се израчуна повећање броја области после додавања n-те тачке P_n, пошто су на кружници већ распоређене тачке $P_1, P_2, \ldots, P_{n-1}$. Дуж P_nP_k ($k=1,2,\ldots,n-1$) сече дужи P_iP_j, где је $1 \leq i \leq k-1$, $k+1 \leq j \leq n-1$, па њено додавање повећава број области за $1+(k-1)(n-k-1)$ (толико области дуж P_nP_k дели на тачно две подобласти; дужи P_nP_k, $k=1,2,\ldots,n-1$, не секу се унутар круга). Према томе,

$$f(n)=f(n-1)+\sum_{i=2}^{n}(1+(k-1)(n-k-1))=f(n-1)+\tfrac{1}{6}(n^3-6n^2+17n-12).$$

Сумирањем (видети (2.10), (2.11)) се добија решење ове диференцне једначине

$$f(n)=1+\tfrac{1}{6}\sum_{i=2}^{n}(i^3-6i^2+17i-12)=\tfrac{1}{24}(n^4-6n^3+23n^2-18n)+1.$$

Ова и претходна сума могу се израчунати коришћењем сума $\sum_{i=1}^{n} i^k$, видети (2.5.3). Друга могућност је просто доказати тачност ових израза индукцијом (пошто се претходно "погоде").

1.16. Доказаћемо најпре три помоћна тврђења; дефиниције појмова видети у уводу поглавља 6.

Најпре се запажа да ако је стабло G повезано, онда је повезано и стабло $f(G)$ (које се састоји од чворова — слика чворова из G, и свих грана из G које их повезују). Тврђење је последица чињенице да f сваки пут из G пресликава или у пут или у један чвор у $f(G)$.

Даље, ако је G стабло и важи $f(G)=G$, онда f сваку грану G пресликава у грану (по дефиницији), и обратно, свака грана у $f(G)=G$ је слика неке гране из G (у противном би грана без инверзне слике затварала циклус у $f(G)=G$, супротно претпоставци да је G стабло). Другим речима, ако је $f(G)=G$, онда је f изоморфизам, пресликавање које чува суседност чворова. Специјално, ако за стабло G важи $f(G)=G$, онда је чвор $v \in V$ лист, акко је чвор $f(v)$ лист.

Нека је $G = (V, E)$ стабло и нека је $V' \subset V$ скуп његових унутрашњих чворова. Тада је индуковани подграф $G' = (V', E')$ такође стабло (повезан је — пут између свака два унутрашња чвора G води само преко унутрашнијх чворова G, и нема циклуса).

Наведене чињенице довољне су за извођење доказа. Скуп чворова графа $f(G)$ је подскуп V, а ако је $V \neq \emptyset$, онда и $f(G)$ има бар један чвор. Посматрајмо низ стабала $G_0 = G$, $G_1 = f(G)$, ..., $G_i = f(G_{i-1})$, ..., и нека је $G_i = (V_i, E_i)$. За неко i мора да буде $V_i = V_{i-1}$ (јер G има коначно много чворова). На крају се индукцијом по $m = |F|$ доказује да ако за стабло $H = (U, F)$ важи $f(H) = H$, онда H садржи непокретни чвор или грану (база је очигледно тврђење за $m = 1, 2$; за $m > 2$ се H замењује са H', стаблом индукованим унутрашњим чворовима H, за које такође важи $f(H') = H'$, па пошто H' има мање грана од H, за њега важи индуктивна хипотеза).

1.17. За $n = 1$ тврђење је тачно (у једној урни се налазе две куглице). Нека је принцип тачан за n куглица и $n+1$ урни. Претпоставимо да су $n+2$ куглице убачене у $n+1$ урни. Да је принцип у том случају тачан, доказујемо претпостављајући супротно, да се у свим уранама налази највише по једна куглица. Издвојивши на страну произвољну урну са само једном куглицом, добијамо систем од n урни са $n + 1$ куглицом, у коме по индуктивној хипотези мора да постоји урна са бар две куглице — супротно претпоставци. Тиме је доказано да Дирихлеов принцип важи за свако n.

1.18. Сума висина чворова КБС висине 1 је $1 = 2^2 - 1 - 2$. Ако је тврђење тачно за КБС висине h, онда је сума висина чворова КБС висине $h + 1$ једнака двострукој суми висина чворова КБС висине h увећаној за $h + 1$ (висину корена): $2(2^{h+1} - h - 2) + h + 1 = 2^{h+2} - (h + 1) - 2$, па је тврђење тачно и за КБС висине $h + 1$.

1.19. Доказ се може извести појачавањем индуктивне хипотезе, тако што се докаже да за $n \geq 2$ важи $F_n^2 + F_{n+1}^2 = F_{2n+1}$ и $F_{n-1}F_n + F_nF_{n+1} = F_{2n}$.

1.20. Тврђење је тачно за $n = 2$ (K_2 је стабло, $n/2 = 1$). Претпоставимо да имамо разлагање K_n на $n/2$ повезујућих стабала. Граф K_{n+2} се од K_n добија додавањем два нова чвора u, v, са граном (u, v) и још $2n$ грана које u, v повезују са чворовима K_n. Чворови u и v могу се "прикачити" као листови свим повезујућим стаблима K_n тако што се искористе $n/2$ грана које u повезују са $n/2$ чворова K_n, и $n/2$ грана које v повезују са преосталих $n/2$ чворова K_n. Грана (u, v) са преосталих $n/2 + n/2$ грана које повезују u и v са свим чворовима K_n чине ново повезујуће стабло графа K_{n+2}, са редним бројем $n/2 + 1$, па је добијено разлагање скупа грана K_{n+2} на $(n + 2)/2$ повезујућих стабала (слика 1).

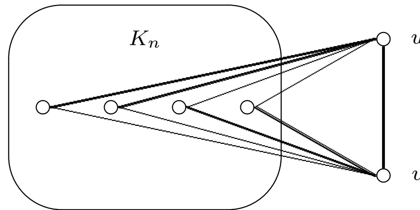


Рис. 1. Уз задатак 1.20.

1.21. За $n = 1$ то је граф са два чвора и једном граном. Како конструисати граф G_n полазећи од графа G_{n-1}? Потребно је додати два нова чвора v, w и $n^2 - (n-1)^2 = 2n-1$ нових грана; да би конструкција била "циста", згодно је да нове гране повезују или нови чвор са старим, или два нова чвора. Ако се v и w повежу граном, и ако то буде једина грана суседна нпр. чвору v, онда тај чвор у савршено упаривање може ући само граном (v, w) — то је изнуђени избор; то такође из упаривања искључује све гране суседне чвору w; интересантно је да се управо w може повезати са $2n - 2$ грана са свим претходним чворовима, па ни једна од тих грана не би могла да уђе у савршено упаривање. Пошто по индуктивној хипотези у

графу G_{n-1} постоји тачно једно савршено упаривање, видимо да и у овако конструисаном графу G_n постоји тачно једно савршено упаривање (видети слику 2).

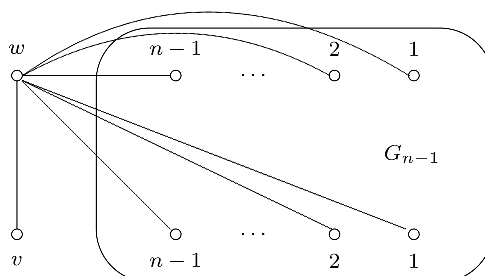


Рис. 2. Уз задатак 1.21.

1.22. Неједнакост се може преписати у облику $\prod_{i=1}^{n}(1+a_i)/2 \geq 1$. За $n=1$ неједнакост (у овом случају једнакост) је тривијална. Претпоставимо да је неједнакост тачна за произвољних $n-1$ реалних бројева са производом 1. Случај n бројева $a_1, a_2, \dots, a_n$ природно је свести на случај $n-1$ бројева тако да се уместо нека два од њих, a и b, узме њихов производ ab — та замена требало би да увек смањује (то треба доказати) вредност израза са леве стране неједнакости. Другим речима, да би свођење успело, морало би да буде

$$\tfrac{1+a}{2} + \tfrac{1+b}{2} \geq \tfrac{1+ab}{2},$$

што је еквивалентно са $(1-a)(1-b) \leq 0$. Да ова неједнакост буде испуњена може се постићи тако да се за a изабере неки број већи или једнак од 1, а за b неки број мањи или једнак од 1. Сада је јасно како се изводи доказ. Међу бројевима $a_1, a_2, \dots, a_n$ сигурно постоји неки (нпр. a_i) који је већи или једнак од 1, и неки (нпр. a_j) који је мањи или једнак од 1. Користећи доказану неједнакост

$$\tfrac{1+a_i}{2} + \tfrac{1+a_j}{2} \geq \tfrac{1+a_i a_j}{2},$$

добијамо

$$\prod_{k=1}^{n} \tfrac{1+a_k}{2} = \tfrac{1+a_i}{2} \cdot \tfrac{1+a_j}{2} \cdot \prod_{\substack{1\leq k\leq n\\ k\neq i,j}} \tfrac{1+a_k}{2} \geq \tfrac{1+a_i a_j}{2} \prod_{\substack{1\leq k\leq n\\ k\neq i,j}} \tfrac{1+a_k}{2} \geq 1.$$

Последња неједнакост у низу је тачна, јер се добија применом индуктивне хипотезе на $n-1$ бројева $a_1, a_2, \dots, a_{i-1}, a_{i+1}, \dots a_{j-1}, a_{j+1}, \dots a_n, a_i a_j$.

1.23. Претпоставимо да је највиши бит бинарног броја $b[1]$. Размотримо следећи алгоритам.

```
begin
    n := 0;
    for i := 1 to k do
        n := n · 2 + b[i]
end
```

Показаћемо да после i-тог проласка кроз петљу важи $n = \sum_{j=1}^{i} b[j]2^{i-j}$, $i = 1, 2, \dots, k$. Заиста, тврђење је тачно за $i = 1$, а ако је тачно за неко i, онда после наредног проласка

кроз петљу n има вредност

$$2\sum_{j=1}^{i} b[j]2^{i-j} + b[i+1] = 2\sum_{j=1}^{i+1} b[j]2^{i+1-j}.$$

Тиме је доказано да је после k-тог проласка кроз петљу $n = \sum_{j=1}^{k} b[j]2^{k-j}$, односно наведени алгоритам је коректан.

13.2. Анализа алгоритама

2.1. Ако је $f(n) = o(g(n))$, онда за произвољно $\epsilon > 0$ постоји природан број n_0 такав да за свако $n > n_0$ важи $f(n) < \epsilon g(n)$, одакле следи да је $f(n) = O(g(n))$. Обрнуто не важи; на пример, $n = O(2n)$, али није $n = o(2n)$.

2.2. Може се поћи од неједнакости $(f(n))^c = O\left(a^{f(n)}\right)$ $(a > 1,\ c > 0)$, и у њој заменити $f(n)$ са $\log_2 n$, a са 2^b и c са a.

2.3. Сабирајући једнакости $T(k) = T(k-1) + \frac{k}{2}$ за $k = 2, 3, \ldots, n$ добија се

$$T(n) = T(1) + \sum_{k=2}^{n} \tfrac{k}{2} = \tfrac{1}{2} + \tfrac{n(n+1)}{4}.$$

2.4. Карактеристична једначина ове диференцне једначине је $z^2 - 8z + 15 = 0$, а њени корени су $z_1 = 5$ и $z_2 = 3$. Због тога решење тражимо у облику $T(n) = c_1 5^n + c_2 3^n$. Константе c_1 и c_2 одређују се из услова $T(1) = 5c_1 + 3c_2$ и $T(2) = 25c_1 + 9c_2$. Решавањем овог система од две линеарне једначине добија се $c_1 = 1/10$, $c_2 = 1/6$, па је $T(n) = (5^{n-1} + 3^{n-1})/2$.

2.5. Карактеристична једначина ове диференцне једначине је $z^3 - 4z^2 + 5z - 2 = 0$, или $(z-1)^2(z-2) = 0$, па је њено решење облика $T(n) = (c_1 + c_2)n + c_3 2^n$. Константе c_1, c_2, c_3 одређују се из почетних услова; добија се $c_1 = c_3 = 0$, $c_2 = 1$, па је $T(n) = n$.

2.6. Због једноставности уведимо ознаке $T(2n) = a_n$ и $T(2n-1) = b_n$, $n \geq 1$. Горње диференцне једначине могу се преписати у облику $a_n = 2\sum_{i=1}^{n} b_i$, $b_{n+1} = 2\sum_{i=1}^{n} a_i$ $(n \geq 1)$, $b_1 = 1$. Првих неколико чланова ових низова су $a_1 = 2b_1 = 2$, $b_2 = a_1 = 2$, $a_2 = 2(b_1 + b_2) = 6$, итд. Идеја је да се из диференцних једначина најпре уклоне суме, формирањем разлика два узастопна члана $(n \geq 1)$: $a_{n+1} - a_n = 2\sum_{i=1}^{n+1} b_i - 2\sum_{i=1}^{n} b_i = 2b_{n+1}$, $b_{n+2} - b_{n+1} = \sum_{i=1}^{n+1} a_i - \sum_{i=1}^{n} a_i = a_{n+1}$, Сада се заменом b_{n+1}, b_{n+2} у другу, односно a_n, a_{n+1} у прву једначину добијају хомогене линеарне диференцне једначине за низове a_n и b_n:

$$(b_{n+2} - b_{n+1}) - (b_{n+1} - b_n) = 2b_{n+1}, \quad \tfrac{1}{2}(a_{n+1} - a_n) - \tfrac{1}{2}(a_n - a_{n-1}) = a_n, \quad n \geq 2,$$

односно

$$b_{n+2} = 4b_{n+1} - b_n, \quad a_{n+1} = 4a_n - a_{n-1}, \quad n \geq 2.$$

Корени заједничке карактеристичне једначине $z^2 - 4z + 1 = 0$ су $z_1 = 2 + \sqrt{3}$ и $z_2 = 2 - \sqrt{3}$, па су решења диференцних једначина облика $a_n = c_1 z_1^n + c_2 z_2^n$ и $b_n = c_3 z_1^n + c_4 z_2^n$; константе c_1, c_2, c_3, c_4 одређују се из почетних услова $b_2 = 2$, $b_3 = 8$ (јер $b_{n+2} = 4b_{n+1} - b_n$ важи тек за $n \geq 2$!!!), односно $a_1 = 2$, $a_2 = 6$. Добија се да је

$$a_n = T(2n) = \tfrac{1}{\sqrt{3}}\left((\sqrt{3}-1)(2+\sqrt{3})^n + (\sqrt{3}+1)(2-\sqrt{3})^n\right), \quad n \geq 1,$$

$$b_n = T(2n-1) = \tfrac{1}{\sqrt{3}}\left(((2+\sqrt{3})^{n-1} + ((2-\sqrt{3})^{n-1}\right), \quad n \geq 2.$$

2.7. Ако са a_n, односно b_n означимо трајање решавања проблема P_n применом алгоритма A, односно B, добијамо систем диференцних једначина

$$a_n = b_{n-1} + n, \quad b_n = a_{n-1} + n, \quad (n \geq 2), \quad a_1 = 1,\ b_1 = 2.$$

Заменом $b_{n-1} = a_{n-2} + n - 1$ ($n \geq 3$) у прву једнакост, добија се диференцна једначина $a_n = a_{n-2} + 2n - 1$ за низ a_n ($a_2 = b_1 + 2 = 4$). Ова једначина решава се сумирањем:

$$\begin{aligned} a_{2n+1} &= a_{2n-1} + 4n + 1 = a_{2n-3} + (4n+1) + (4n-3) = \dots \\ &= a_1 + (4n+1) + (4n-3) + \dots + 1 = 1 + (n+1)(2n+1), \\ a_{2n} &= a_{2n-2} + 4n - 1 = a_{2n-4} + (4n-1) + (4n-5) = \dots \\ &= a_2 + (4n-1) + (4n-5) + \dots + 7 = n(2n+1) + 1. \end{aligned}$$

Дакле,

$$a_n = \begin{cases} 1 + \frac{n(n+1)}{2}, & \text{за } n \text{ парно} \\ \frac{n(n+1)}{2}, & \text{за } n \text{ непарно} \end{cases}.$$

2.8. Довољно је доказати индукцијом да постоји таква константа $c > 0$ да за све довољно велике n важи $T(n) \leq cn \log_2^2 n$. Ова неједнакост је за $n = 1$ тачна за произвољно $c > 0$. Треба још показати да ако је неједнакост тачна за бројеве $n < N$, онда је тачна и за $n = N$. Заиста, користећи индуктивну хипотезу и неједнакост $\lfloor x \rfloor \leq x$, добијамо

$$\begin{aligned} T(N) &= 2T\left(\lfloor N/2 \rfloor\right) + 2N\log_2 N \leq 2c\lfloor N/2\rfloor \log_2^2 \lfloor N/2 \rfloor + 2N \log_2 N \\ &\leq cN\left(\log_2 N - 1\right)^2 + 2N\log_2 N = cN\log_2^2 N + N\left(2(1-c)\log_2 N + c\right). \end{aligned}$$

Ако је $c > 1$, онда за $N \geq 2$ важи $2(1-c)\log_2 N + c \leq 2(1-c)\log_2 2 + c = 2 - c$; за $c \geq 2$ је овај израз мањи или једнак од нуле, па важи $T(N) \leq cN \log_2^2 N$, чиме је доказано да ова неједнакост важи за свако $N \geq 1$ ако је $c \geq 2$.

2.9. Нека је $n = 2^k$. Множењем са 2^{-n} добија се $2^{-k}T(2^k) = 2^{-(k-1)}T(2^{k-1}) + ck2^{-k}$. Према томе,

$$\begin{aligned} 2^{-k}T(2^k) &= 2^{-(k-2)}T(2^{k-2}) + c\left(k2^{-k} + (k-1)2^{-(k-1)}\right) = \cdots \\ &= 2^{-0}T(2^0) + c\sum_{i=1}^{k} i2^i = 2^{-k}c(2^{k+1} - k - 2) \end{aligned}$$

(видети пример 2.5, страна 20), односно $T(n) = O(n)$.

Друго решење. Индукцијом се може доказати да је $T(n) \leq 2cn$.

2.10. (а) Неједнакост се може доказати индукцијом по n. За $n = 1$ неједнакост је тачна. Ако претпоставимо да је тачна за бројеве мање од n, онда је

$$T(n) \leq T\left(\lfloor n/2 \rfloor\right) + 1 \leq \lfloor \log_2 \lfloor n/2 \rfloor \rfloor + 2 \leq \lfloor \log_2\left(2\lfloor n/2\rfloor\right)\rfloor + 1 \leq \lfloor \log_2 n \rfloor + 1$$

(јер за свако n важи $2\lfloor n/2 \rfloor \leq n$), па је тврђење тачно и за n.

(б) Слично као у претходном случају

$$T(n) \leq T\left(\lfloor n/2 \rfloor\right) + 1 \leq \lceil \log_2 \lfloor n/2 \rfloor \rceil + 1 = \lceil \log_2\left(2\lfloor n/2 \rfloor\right)\rceil \leq \lceil \log_2 n \rceil.$$

2.11. Одузимањем дате једнакости од $T(n+1) = n + 1 + \sum_{i=1}^{n} T(i)$, добија се $T(n+1) - T(n) = 1 + T(n)$, односно $T(n+1) = 2T(n) + 1$, или

$$T(n+1) + 1 = 2(T(n) + 1) = 2^2(T(n-1) + 1) = \cdots 2^{n-1}(T(2) + 1) = 2^{n+1}$$

(јер је $T(2) = 2 + T(1) = 3$). Дакле, $T(n) = 2^n - 1$.

2.12. За $i \leq x \leq i + 1$ је $i^k \log_2 i \leq x^k \log_2 x$, па је

$$\begin{aligned} \sum_{i=1}^{n} i^k \log_2 i &= \sum_{i=1}^{n} \int_i^{i+1} i^k \log_2 i \, dx \leq \sum_{i=1}^{n} \int_i^{i+1} x^k \log_2 x \, dx = \int_1^{n+1} x^k \log_2 x \, dx \\ &= \frac{\log_2 e}{k+1}\left(x^{k+1}\ln x - \frac{x^{k+1}}{k+1}\right)\Big|_1^{n+1} = O(n^{k+1}\log n). \end{aligned}$$

2.13. Контрапример: $f(n) = 3n$, $g(n) = 2n$, $r(n) = s(n) = n$,

2.14. За $f(n) = s(n) = g(n) = n$ и $r(n) = n^2$ је ако је $f(n) = O(s(n))$ и $g(n) = n = O(n^2) = O(r(n))$, али $f(n)/g(n) = 1 \neq O(\frac{1}{n}) = O(s(n)/r(n))$.

2.15. Функције се могу изабрати тако да једна од њих нагло расте за парне, а друга за непарне n. Посматрајмо следеће две растуће функције:

$$f(n)=\begin{cases} n!, & n \text{ парно} \\ (n-1)!+1, & n>1 \text{ непарно} \end{cases} \quad ; \quad g(n)=\begin{cases} (n-1)!+1, & n \text{ парно} \\ n!, & n \text{ непарно} \end{cases} .$$

Пошто је

$$\frac{f(n)}{g(n)}=\begin{cases} n\left(1+\frac{1}{(n-1)!}\right)^{-1}, & n \text{ парно} \\ \frac{1}{n}\left(1+\frac{1}{(n-1)!}\right), & n>1 \text{ непарно} \end{cases} ,$$

ни $f(n)/g(n)$ ни $g(n)/f(n)$ нису ограничене функције: за парно n је $f(n)/g(n) \geq \frac{n}{2}$, а за непарно n је $g(n)/f(n) \geq \frac{n}{2}$.

2.16. Нека је $a_k = T(2^k)$. Тада је $a_1 = 1$ и $a_k = 2a_{k-1}+1$, или

$$a_k+1 = 2(a_{k-1}+1) = 2^2(a_{k-2}+1) = \cdots = 2^{k-1}(a_1+1) = 2^k.$$

Према томе, $T(n) = n-1$ ако је n степен двојке. У описаном покушају се могу избећи тешкоће ако се као циљ постави доказивање неједнакости $T(n) \leq cn+d$. Из претпоставке да је она тачна за $k<n$ следи да је

$$T(n) = 2T\left(\frac{n}{2}\right)+1 \leq 2\left(c\frac{n}{2}+d\right)+1 = cn+d+(d+1),$$

што је мање или једнако од $cn+d$ акко је $d \leq -1$. Према томе, неједнакост $T(n) \leq cn+d$ је тачна (за $n=2^k$) акко је $d \leq -1$ и $T(2) = 1 \leq 2c+d$, односно $c \geq (1-d)/2$. Најмање c добија се за највеће $d=-1$, па је најбоља горња граница ове врсте $T(n) \leq n-1$ — у складу са претходним решењем.

2.17. Може се искористити теорема 2.5.4, јер су m и c константе. Према томе,

$$S(n)=\begin{cases} O\left(n^{\log_m(cm\log_2 m)}\right), & m>2^{1/c} \\ O(n\log n), & m=2^{1/c} \\ O(n), & m<2^{1/c} \end{cases} .$$

2.18. За n облика $cm+n_0$, $0 \leq n_0 < c$ означимо израз $T(cm+n_0)$ са $R(m)$. Тада је $R(m) = 2R(m-1)+k$. Множећи ову једнакост са 2^{-m}, добијамо $S(m) = S(m-1)+k2^{-m}$, где је $S(m) = 2^{-m}R(m)$. Решење последње диференцне једначине је

$$S(m) = k(2^{-m}+2^{-(m-1)}+\ldots+2^{-0})+S(0) = S(0)+k(1-2^{-m}).$$

Даље је

$$R(m) = T(cm+n_0) = 2^m\left(S(0)+k(1-2^{-m}\right) = O(2^m) = O(d^{cm+n_0})$$

(где је $d=2^{1/c}$), односно $T(n) = O(d^n)$.

2.19. Решење је $T(n)=O(n)$. Да бисмо то доказали, претпоставимо да постоји константа d таква да је $T(k) < dk$ за $k<n$. Тада је

$$T(n) = \sum_{i=1}^{k} a_i T(n/b_i) + cn < \sum_{i=1}^{k} a_i dn/b_i + cn = nd\left(\sum_{i=1}^{k} a_i/b_i + c/d\right).$$

Константа d може се изабрати тако да буде $d \geq c/\left(1-\sum_{i=1}^k a_i/b_i\right)$ односно $\sum_{i=1}^k a_i/b_i + c/d < 1$. Тада из горње неједнакости следи $T(n) < dn$, односно тиме је доказано да неједнакост $T(n) < dn$ важи за свако (довољно велико) n.

2.20. У оба случаја се први утисак о низовима добија израчунавањем неколико првих чланова; оба низа су монотоно неопадајућа, што се лако доказује индукцијом.

(а) Првих неколико чланова низа, почевши од $T(2)$ су 1, 8, 10, 26, 28, 30, 32, 34, 40, 42, ... Индукцијом се доказује да је

$$T(n) = 1+4+4^2+\ldots+4^k = (4^{k+1}-1)/3 \quad \text{за } 2^{2^{k-1}}+1 \leq n \leq 2^{2^k}.$$

Одатле се види да за $2^{k-1} < \log_2 n \leq 2^k$ важи

$$(4\log_2^2 n-1)/3 \leq (4(2^k)^2-1)/3 = T(n) = (16(2^{k-1})^2-1)/3 < (16\log_2^2 n-1)/3,$$

односно $T(n) = \Theta(\log^2 n)$.

(б) Очигледно је $T(n) \geq 2n$. С друге стране, индукцијом се показује да је $T(n) < 6n$. Непосредно се проверава да је ово тачно за $n \leq 14$. За $n \geq 15 > 3(5+\sqrt{21})/2$ из неједнакости $T(\lceil\sqrt{n}\rceil) < 6\lceil\sqrt{n}\rceil$ следи

$$T(n) < 2 \cdot 6\lceil\sqrt{n}\rceil + 2n \leq 12(\sqrt{n}+1) + 2n = 6n - \left((2\sqrt{n}-3)^2 - 21\right) < 6n,$$

јер је $\sqrt{n} > (3+\sqrt{21})/2$ еквивалентно са $n > 3(5+\sqrt{21})/2$.

2.21. Нека је $n = 2^i$, $a_i = T(2^i)$, $a_0 = c$. За $i \geq 1$ је тада $a_i = ka_{i-1} + f(2^i)$, односно

$$\begin{aligned} a_i k^{-i} &= a_{i-1}k^{-(i-1)} + k^{-i}f(2^i) = a_{i-2}k^{-(i-2)} + k^{-(i-1)}f(2^{i-1}) + k^{-i}f(2^i) = \\ &= = \ldots c + k^{-1}f(2^1) + k^{-2}f(2^2) + \ldots + k^{-i}f(2^i). \end{aligned}$$

Ако уведемо ознаку $2^j = m$, онда је

$$k^{-j}f(2^j) = k^{-\log_2 m}f(m) = 2^{-\log_2 k \log_2 m}f(m) = m^{-\log_2 k}f(m) = g(m).$$

Пошто је $k^i = k^{\log_2 n} = 2^{\log_2 k \log_2 n} = n^{\log_2 k}$, добија се да је

$$T(n) = n^{\log_2 k}\left(c + \sum_{j=1}^{i} k^{-j}f(2^j)\right),$$

што је и требало доказати.

2.22. Претпостављајући да је $n = 2^k$, добијамо

$$\begin{aligned} T(n) &= T(2^k) = T(2^{k-1}) + 2^{k/2} = T(2^{k-2}) + 2^{(k-1)/2} + 2^{k/2} = \ldots \\ &= T(1) + 2^{1/2} + 2^{2/2} + \cdots + 2^{k/2} = \\ &= \tfrac{2^{(k+1)/2}-1}{\sqrt{2}-1} = (\sqrt{2n}-1)(\sqrt{2}+1) = O(\sqrt{n}). \end{aligned}$$

2.23. Очигледно је $T(n) > 3n$. Индукцијом се лако доказује да је за довољно велико n $T(n)$ неопадајућа функција. За велике n је $T(n)$ са горње стране ограничено низом $S(n)$ који задовољава диференцну једначину $S(n) = 2T(n/b) + 3n$ (за произвољно велико b), односно $S(n) = O(n^{\log_b 2}) = O(n^{1+\epsilon})$, где је ϵ произвољно мало. Наслућује се да би $T(n)$ и са горње стране могло бити ограничено линеарном функцијом.

Нека је $a > 3$ константа. Из претпоставке да за $k < n$ важи $T(k) < ak$, за $\log_2 n > 2a/(a-3)$ следи

$$T(n) = 2T\left(\left\lfloor \tfrac{n}{\log_2 n} \right\rfloor\right) + 3n < 2a\left\lfloor \tfrac{n}{\log_2 n}\right\rfloor + 3n < \tfrac{2an}{\log_2 n} + 3n < an.$$

Специјално, за $a = 9$, $T(n) < 9n$ следи из $T(k) < 9k$ за $k < n$ ако је $\log_2 n > 3$, односно $n > 8$. За $n \leq 8$ непосредно се проверава да је $T(n) < 9n$, па за свако $n \geq 1$ важи $3n \leq T(n) \leq 9n$, односно $T(n) = \Theta(n)$.

2.24. Неједнакост

$$\lceil \log_2 \tfrac{n}{i} \rceil \leq 1 + \log_2 \tfrac{n}{i} \leq 1 + \log_2 \tfrac{n}{x}$$

важи за свако x из интервала $[i-1, i]$, $2 \leq i \leq n$. Интеграцијом у границама од $i-1$ до i, па сумирањем по i од 2 до n, добија се неједнакост

$$S(n) \leq log_2 n + n - 1 + \int_1^n \log_2 \frac{n}{x}\,dx = (n-1)(1 + \log_2 e),$$

па је $S(n) = O(n)$.

2.25. Нека је $n = 2^k$. Скуп $\{1, 2, \ldots, n-1\}$ вредности индекса i, (сабирак за $i = n$ једнак је нули), може се разложити на k подскупова $\{2^{j-1}, 2^{j-1}+1, \ldots, 2^j - 1\}$, $j = 1, 2, \ldots, k$, на којима израз $\lfloor \log_2 \frac{n}{i} \rfloor$ има константну вредност $k - j + 1$. Према томе,

$$\sum_{i=1}^{n} \lfloor \log_2 \tfrac{n}{i} \rfloor = \sum_{j=1}^{k} 2^{j-1}(k-j+1) = \sum_{j=0}^{k-1} 2^j(k-j) = \sum_{j=1}^{k} j2^{k-j} = 2^{k+1} - k - 2$$

(видети пример 2.5). Тражена сума је дакле $2n - \log_2 n - 2$.

2.26. Бројеви $F(n)$ и $G(n)$ су дефинисани за све целе индексе n. Чланови низа $G(n)$ са индексима 0 и 1 су $G(0) = F(0) = 0$, $G(1) = F(-1) = 1$, а одговарајућа диференцна једначина је $G(-n-2) = G(-n-1)+G(-n)$, односно (стављајући $-n-2 = k$), $G(k+2) = G(k)-G(k+1)$. Непосредно се проверава да бројеви $G'(k) = (-1)^{k+1}F(k)$ задовољавају ову диференцну једначину, а пошто је и $G'(0) = G(0) = 0$ и $G'(1) = F(1) = 1 = G(1)$, онда за $k \geq 0$ важи $G(k) = G'(k) = (-1)^{k+1}F(k)$.

13.3. Структуре података

3.1. Тело рекурзивне процедуре се слике 3.12 може се заменити блоком са слике 3.

```
begin
    v := Koren;
    if Koren = nil or Koren^.Ključ = x then
        Čvor := Koren
    else
        while v ≠ nil and v^.Ključ ≠ x do
            if v^.Ključ < x then
                v := v^.Levi
            else
                v := v^.Desni
end
```

Рис. 3. Уз задатак 3.1.

3.2. Проблем се може решити индукцијом, односно рекурзивним алгоритмом. Тражени низ се добија надовезивањем низа који одговара левом подстаблу, кључа корена стабла, и на крају низа који одговара десном подстаблу. За стабло које има само корен, тражени низ је једночлан — садржи само кључ корена. Број рекурзивних позива једнак је броју чворова стабла n, а трајање сваког позива је $O(1)$, па је сложеност алгоритма $O(n)$.

3.3. Ако чвор v са кључем a у стаблу има десног сина, онда је тражени број кључ следбеника чвора v у стаблу, односно кључ чвора до кога се од десног сина v долази идући наниже улево све док је то могуће. У противном, ако v нема десног сина, а леви је син свог оца, онда је кључ оца тражени број. На крају, ако v нема десног сина, и десни је син свог оца, онда је a највећи елеменат скупа A. Сложеност алгоритма је $O(\log n)$.

3.4. Пошто је позиција 16 заузета, мора бити заузета и позиција 8, на којој треба да буде отац чвора 16. Слично, морају бити заузете и позиције 4, 2 и 1. Према томе, минимални број елемената хипа у низу дужине 16 је пет.

3.5. Обе операције, уметање новог елемента и уклањање елемента са највећим кључем, имају сложеност пропорционалну висини стабла, дакле $O(\log n)$, где је n број елемената листе.

3.6. За реализацију ове структуре података може се искористити стек, комбинација вектора и променљиве која показује на позицију последњег податка у вектору. Нови податак се записује на крају вектора, а са краја вектора узима се и податак кога треба уклонити.

3.7. Погодна структура је бинарно стабло претраге. Сложеност операција сразмерна је висини стабла, па у најгорем случају може да буде $O(n)$, где је n број елемената стабла; ако се користе нпр. АВЛ стабла, онда је сложеност операција у најгорем случају $O(\log n)$.

3.8. Користи се вектор дужине n, а у њему се елеменат i смешта на позицију i. Према томе, уметање је једноставно. Уклањање је компликованије, јер i није прецизирано, а ми не

знамо које позиције у низу нису заузете. Проблем се решава повезивањем свих заузетих позиција у повезану листу. Уметање i према томе обухвата означавање i-те позиције и уметање позиције i у листу. Уклањање брише први елеменат из листе и уклања ознаку са позиције тог елемента.

3.9. Најпре се уклања произвољан елеменат из једног од хипова (нпр. његов корен) и изведу се одговарајуће поправке хипа. Тај елеменат се затим умеће као корен новог хипа, тако да корени два стара хипа буду његови синови. Поправка добијеног стабла — хипа (евентуално премештање наниже елемента из корена) захтева $O(m+n)$ корака.

3.10. Погодна структура података за реализацију овог типа је АВЛ стабло, у коме се уметања и брисања извршавају за $O(\log n)$ корака. Да би се извела операција **Naredni**(x) користи се најпре обичан алгоритам за тражење x, после чега треба пронаћи следбеника x у стаблу (видети задатак 3.3).

3.11. Погодна структура података за реализацију овог типа је АВЛ стабло, у коме се уметања и брисања извршавају за $O(\log n)$ корака. Сваком чвору v додајемо ново поље $v.D$, које садржи број потомака v, заједно са v. Нека за дати чвор v $Rang(v)$ означава број елемената стабла, са кључем мањим или једнаким од $v.Klju\check{c}$. Коришћењем поља D може се одредити *ранг* чвора v у току његовог тражења, на следећи начин. Индуктивна хипотеза је да знамо рангове свих чворова на путу од корена до неког чвора w. Ако корен има левог сина, онда је његов ранг за један већи од вредности D његовог левог сина; у противном је ранг корена 1. Нека је w_L леви син чвора w, w_D десни син w, w_{LD} десни син w_L и w_{DL} леви син w_D. Рангови синова чвора w могу се изразити на следећи начин

$$\begin{aligned} Rang(w_L) &= Rang(w) - w_{LD}.D - 1 \\ Rang(w_D) &= Rang(w) + w_{DL}.D + 1 \end{aligned}$$

(ако неки од синова не постоји, узима се да је одговарајућа вредност поља D једнака 0). Да би се извршила операција **Naredni**(x, k), треба најпре пронаћи чвор v такав да је $v.Klju\check{c} = x$ (ако такав чвор не постоји, онда је потрага неуспешна). Затим се тражи чвор са рангом $Rang(v) + k$; тражење према рангу исте је сложености као и тражење према кључевима: рангови се израчунавају успут, а редослед у стаблу исти им је као редослед кључева. Тиме посао није завршен; пошто је уведено ново поље, треба га ажурирати после операција које га мењају. После уметања, односно брисања елемената инкрементирају се, односно декрементирају поља D у свим чворовима на путу од корена; у случају неуспешног уметања или брисања ове промене се морају поништити. Зашто је уведено поље D уместо поља $Rang$, које би тражење кључа са датим рангом чинило једноставнијим? Због тога што уметање и брисање у најгорем случају мењају рангове $O(n)$ елемената!

3.12. Нека су T_1 и T_2 два дата стабла, при чему су сви кључеви у стаблу T_2 већи свих кључева у стаблу T_1, и нека је висина h стабла T_2 већа од висине стабла T_1. У стаблу T_2 треба пронаћи чвор v са најмањим кључем (спуштајући се полазећи од корена улево, све док је могуће); за то је потребно $O(h)$ операција. Чвор v се затим брише из T_2, па се формира ново стабло са кореном v и подстаблима T_1 и T_2 (без v); за све ово потребно је извршити $O(h)$ операција.

3.13. Решење је слично решењу задатка 3.12, при чему треба обезбедити да добијено стабло буде уравнотежено, без обзира што дата стабла T_1 и T_2 могу бити различитих висина h_1 и h_2. Пошто се ради о АВЛ стаблима, претпоставља се да се уз сваки чвор чува податак о његовој висини. Нека је нпр. $h_1 \geq h_2$. Према дефиницији конкатенације, претпоставља се да је највећи кључ из T_1 мањи од најмањег кључа из T_2. Идеја је да се највећи елеменат r из T_1 обрише, па да се искористи као корен преко кога се T_2 привезује на одговарајуће место у левом подстаблу T_1. Прецизније, спуштајући се у T_1 само десним гранама, долазимо до чвора v чија је висина или h_2 или $h_2 - 1$ (такав чвор увек постоји у T_1; лако се проверава да се висине оца и сина у АВЛ стаблу разликују или за један или за два); нека је p отац v. Тада на p као десног сина уместо v качимо r, чији је леви син v (са својим подстаблом из T_1), а десни син корен T_2. Тако се добија исправно бинарно стабло претраге. После ове операције

висина чвора r може да постане за један већа од висине чвора v који је био на том месту, па је евентуално потребно извршити уравнотежавање помоћу ротације.

3.14. (а) Случајно АВЛ стабло висине 0 састоји се од само једног чвора — корена. Случајно АВЛ стабло висине 1 добија се избором са вероватноћама 1/3 једне од три могућности: корен има или само десног сина, или оба сина, или само левог сина. Случајно АВЛ стабло висине $h > 2$ конструише се рекурзивно, тако што се са вероватноћама 1/3 бира једна од три варијанте у којима су висине левог и десног подстабла редом $(h-2, h-1)$, $(h-1, h-1)$ и $(h-1, h-2)$.

(б) Посматрајмо факторе равнотеже у чворовима пута којим се прелази од корена случајног АВЛ стабла, преко критичног чвора v, до новоуметнутог чвора u, и то редом од u до v (могуће је такође да на путу до u *нема* критичног чвора; вероватноћа таквог исхода је 3^{-m}, где је m дужина пута од корена до u). На том путу, почевши од оца w чвора u, фактори равнотеже имају униформну расподелу на скупу $\{-1, 0, 1\}$, а пут се завршава (јер се дошло до критичног чвора) оног тренутка кад се наиђе на фактор различит од 0. Вероватноћа да овај пут има дужину i је $\left(\frac{1}{3}\right)^{i-1}\frac{2}{3}$. Математичко очекивање дужине овог пута је $\sum_{i=1}^{\infty} i \cdot 2 \cdot 3^{-i} = \frac{3}{2}$; ова вредност не зависи од висине h АВЛ стабла.

3.15. Први број се ставља у корен стабла. Индукцијом по k показује се да бројеви са индексима $2^k, 2^k+1, \ldots, 2^{k+1}-1$ могу бити редом уметнути у стабло тако да у сваком чвору у сваком кораку буде испуњен услов равнотеже (да разлика броја чворова у левом и десном подстаблу не премашује 1). Ако је ово тачно за неко k, онда се 2^{k+1} бројева са индексима $2^{k+1}, 2^{k+1}+1, \ldots, 2^{k+2}-1$ такође могу редом убацити у стабло тако да буде задовољен услов равнотеже. Заиста, ове бројеве треба наизменично убацивати у лево, односно десно подстабло; по индуктивној хипотези, пошто лево и десно подстабло на почетку имају по $2^k - 1$ чворова, у њих се може редом убацити 2^k бројева тако да услов равнотеже буде испуњен у сваком кораку. Поред тога, услов равнотеже је увек испуњен и за корен стабла, јер се бројеви наизменично убацују у лево и десно подстабло.

Друго решење. Исти алгоритам може се формулисати и директно (уместо рекурзивно): приликом убацивања броја са индексом $n = (b_k b_{k-1} \ldots b_0)_2$, где су $b_k = 1, b_{k-1}, \ldots, b_0$ бинарне цифре броја n, бинарне цифре $b_k, b_{k-1}, \ldots, b_0$ одређују пут до места убацивања чвора: цифра 0, односно 1, одређује скретање улево, односно удесно. Ово тврђење доказује се индукцијом.

3.16. Утисак о начину нарастања АВЛ стабла може се стећи исцртавањем неколико првих стабала (на слици 4 — до $n = 7$). запажа се да се за $n = 2^{k+1} - 1$ добија потпуно бинарно стабло K_k висине k; да се додавањем броја 2^{k+1} (без ротације) добија стабло висине $k+1$, и та висина се не мења после додавања бројева $2^{k+1}+1, 2^{k+1}+2, \ldots, 2^{k+1}+2^{k+1}-1$ (укупно 2^{k+1} бројева). Ово тврђење се доказује индукцијом (у корену K_k је број 2^k; приликом додавања првих 2^k бројева $2^{k+1}, 2^{k+1}+1, \ldots, 2^{k+1}+2^k-1$ мења се само десно подстабло, које од K_{k-1} прелази у K_k; затим се при додавању $2^{k+1}+2^k$ извршава ротација, и корен уместо 2^k постаје 2^{k+1}, па десно подстабло поново постаје K_{k-1}, спремно да прими без промене висине нових $2^k - 1$ бројева $2^{k+1}+2^k+1, 2^{k+1}+2^k+2, \ldots, 2^{k+1}+2^{k+1}-1$, прелазећи поново у K_k). Уравнотежавање стабла се увек изводи једноструком ротацијом, јер нови број, као највећи, увек завршава у десном подстаблу десног подстабла.

3.17. Свако бинарно стабло може се нацртати тако (у "растављеном облику") да за сваки његов чвор v важи да су сви чворови његовог левог (десног) подстабла нацртани лево (десно) испод њега. Ако се чворовима стабла као кључеви придруже њихове x-координате после исцртавања у растављеном облику, стабло постаје БСП. Погодним избором координатног почетка лако се постиже да сви кључеви буду позитивни. Нека је v леви син произвољног чвора u и нека је T_1, односно T_2, лево, односно десно подстабло чвора v. Ротација око гране (u, v) стабла је трансформација при којој

- T_2 уместо десног подстабла v постаје лево подстабло u;
- u уместо сина неког чвора x постаје десни син чвора v;
- v уместо левог сина чвора u постаје син чвора x.

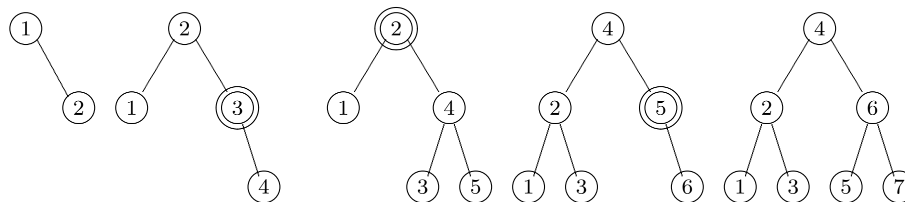


Рис. 4. Уз задатак 3.16.

Симетрично се може описати ротација око гране (u, v) ако је v десни син чвора u. Ако је полазно стабло приказано у растављеном облику, после ротације се такође може представити у растављеном облику, при чему x-координате чворова *нису промењене.* Јасно је да је ротација реверзибилна трансформација (после ротације новог стабла око гране (v, u) добија се полазно стабло).

Лема. Бинарно стабло са n чворова може се помоћу n ротација трансформисати у "линеарно" стабло, у коме ни један чвор нема левог сина.

Лема се може доказати индукцијом. За $n = 1, 2$ тврђење је тачно. Нека је тврђење тачно за свако бинарно стабло са $< n$ чворова, и нека је дато произвољно бинарно стабло са n чворова. Из њега привремено уклонимо произвољан лист v који је леви син неког чвора (ако такав лист не постоји, стабло је већ линеарно). Остатак стабла се помоћу највише $n-1$ ротације трансформише у линеарно стабло. Грана којом је v повезан са својим оцем u није учествовала ни у једној ротацији, па је v и даље лист. Применом још једне ротације око гране (u, v) добија се линеарно стабло. Тиме је лема доказана.

Приметимо да је резултујуће линеарно стабло једнозначно одређено полазним стаблом, јер су x-координате полазних чворова једнаке као у полазном стаблу. Исти низ ротација, примењених обрнутим редоследом, преводи линеаризовано стабло у полазно. Помоћу ове леме лако је доказати тврђење задатка. Заиста, T_1 и T_2 се са по највише n ротација трансформишу у исто, линеарно стабло T од n чворова. После тога се стабло T помоћу највише n ротација (истих оних које T_2 преводе у T, али обрнутим редоследом) преводи у стабло T_2.

3.18. Може се искористити слична идеја као за формирање унија скупова. Полази се од "елементарних" графова — графова задатих листом повезаности. У општем случају се графови представљају коренским стаблом, при чему су елементарни графови представљени само једним чвором — кореном. Унија два графа–стабла формира се тако што се показивач на оца корена једног стабла (претходно **nil**) усмери ка корену другог стабла, дакле помоћу $O(1)$ операција. Преостаје питање како за граф представљен оваквим стаблом одредити скуп чворова и како за два чвора установити да ли су повезани граном. Скуп чворова графа је унија свих скупова чворова — елементарних графова. Да би се за два чвора установило да ли су повезани граном, најпре се проверава да ли припадају елементарном графу; ако нису, онда су сигурно повезани граном; у противном, користи се листа повезаности њиховог заједничког елементарног графа.

3.19. За скуп T може се искористити структура од два нивоа АВЛ стабала: чворови стабла на вишем нивоу (главног стабла) су блокови чији елементи се појављују у T, са показивачима на корене стабала нижег нивоа, намењених за смештање елемената појединих блокова. Операције **Umetni**(s_i) и **Obriši**(s_i) почињу извршењем процедуре **Koji_blok**(s_i); добијени број блока се користи да се из главног стабла добије показивач на стабло са елементима тог блока у T, после чега преостаје да се на обичан начин изврше операције у том стаблу. Трећа операција **Obriši_blok**(j) извршава се уклањањем блока j из главног стабла (после тога помоћно стабло j постаје недоступно).

3.20. За израчунавање сума може се искористити помоћни низ S, у коме се суме бројева смештају на посебан, "разгранат" начин. Претпоставимо због једноставности да је $n = 2^k - 1$

за неко k. Збир бројева са индексима од 1 до 2^{k-1} смештамо у $S[2^{k-1}]$. На тај начин је проблем подељен на два потпроблема двоструко мање величине — налажење суме за леву (од 1 до $2^{k-1}-1$) и десну (од $2^{k-1}+1$ до 2^k-1) половину низа A. Сваки од потпроблема може се решити индукцијом. Први потпроблем решава се независно, изузев што се после сваке операције **Dodaj** мора извршити одговарајућа измена суме у $S[2^{k-1}]$. Други потпроблем се такође решава независно, изузев што добијеној вредности **Parcijalna_suma** треба додати $S[2^{k-1}]$. "Размотавајући" рекурзију видимо да $S[i]$ садржи суму бројева од $A[j+1]$ до $A[i]$, где је j са доње стране најближи индекс индексу i, који је дељив већим степеном двојке него i (другим речима, j се од i добија претварањем у нулу његове најдесније цифре 1 у бинарном запису). На пример, $S[12]$ садржи суму $A[9]+A[10]+A[11]+A[12]$ ($i=12=(1100)_2$, па је $j=(1000)_2=8$.

3.21. Све операције осим последње лако се реализују помоћу АВЛ стабла. Да би се реализовала операција **Dodaj_svima**(y) користи се нова глобална променљива $Nivo$, у којој се чува збир свих вредности — аргумената операције **Dodaj_svima**. Према томе, **Dodaj_svima**(y) просто додаје вредност y променљивој $Nivo$. **Odredi_Vrednost**(x) додаје $Nivo$ вредности придруженој x. **Umetni**(x, y) додељује кључу x вредност $y - Nivo$.

13.4. Конструкција алгоритама индукцијом

4.1. Претпоставимо због једноставности да је степен n полинома број облика 2^k-1. Могућа је декомпозиција полинома $P(x)=a_0+a_1x+\cdots+a_nx^n$ на парни и непарни део, $P(x)=P_p(x^2)+xP_n(x^2)$, где су

$$P_p(x)=a_0+a_2x+a_4x^2+\cdots+a_{n-1}x^{(n-1)/2},$$
$$P_n(x)=a_1+a_3x+a_5x^2+\cdots+a_nx^{(n-1)/2}$$

полиноми степена $(n-1)/2$ са коефицијентима парних, односно непарних индекса полинома P. Тиме се израчунавање вредности полинома степена n своди на израчунавање вредности два полинома степена $(n-1)/2$, два множења и једно сабирање. Ако $T(n)$ означава број операција потребних за израчунавање вредности полинома степена $n-1$ (полинома са n коефицијената), онда је $T(2^k)=2T(2^{k-1})+c$, где је $c=2$ за број множења, а $c=1$ за број сабирања. Почетне вредности су $T(0)=0$ и $T(1)=1$. Решење ове диференцне једначине је $T(2^k)=2^k-1$ (сабирања, за $c=1$), односно $T(2^k)=2^k+2^{k-1}-2$ (множења, за $c=2$). Дакле, за израчунавање вредности полинома степена $n=2^k-1$ потребно је извршити n сабирања и $(3n-4)/2$ множења — нешто више (за фактор 3/2) у односу на Хорнерову шему. Предност овог алгоритма над Хорнеровом шемом је једноставна паралелизација, јер се вредности $P_p(x^2)$ и $P_n(x^2)$ могу израчунавати независно.

4.2. Тврђење да за $A=\{1,2,\ldots,n\}$ и произвољну функцију $f:A\to A$ постоји непразан подскуп $S\subset A$ такав да је на њему f бијекција, може се доказати индукцијом по n, малом изменом индуктивне конструкције алгоритма. Тврђење је тачно за $n=1$. Ако је тачно за бројеве $<n$, онда ли су у $A=\{1,2,\ldots,n\}$ сви елементи слике неких елемената из A (тада је $S=A\neq\emptyset$), или постоји такав елеменат $a\in A$ у који се не пресликава ни један елеменат A. Тада је $f:A\backslash\{a\}\to A\backslash\{a\}$ функција дефинисана на подскупу $A\backslash\{a\}\subset A$, па по индуктивној хипотези постоји подскуп $S\subset A\setminus\{a\}\subset A$, $S\neq\emptyset$, на коме је f бијекција.

4.3. Инваријанта петље је следећи исказ: само једна међу особама $1,2,\ldots,Naredni-1$ може да буде кандидат за звезду, а то је особа i ако је $j=Naredni$, или особа j ако је $i=Naredni$.

4.4. Рекурзивни алгоритам за израчунавање фактора равнотеже и висина свих чворова из одељка 4.7 решава овај проблем као специјални случај, а ефикасан је: за приближно уравнотежена стабла сложеност $T(n)$ (где је n број чворова) задовољава диференцну једначину $T(n)=2T(n/2)+c$, чије је решење $T(n)=O(n)$. Прецизније, да би се дошло до алгоритма, треба мало променити индуктивну хипотезу: за стабло са мање од n чворова умемо да израчунамо висине свих чворова и да проверимо да ли је АВЛ стабло. За дато стабло са n чворова најпре се израчунава висина корена преко висине подстабала. Стабло је АВЛ стабло ако је

фактор равнотеже корена (разлика висина левог и десног подстабла, ако оба постоје, или висина левог подстабла, ако само оно постоји, или висина десног подстабла са знаком минус, ако само оно постоји, или 0 ако стабло има само корен) у скупу $\{-1, 0, 1\}$, и оба подстабла су му (ако постоје) АВЛ стабла. Ако се на појаву чвора са фактором равнотеже модула већег од 1 реагује изласком из рекурзије, онда је алгоритам ефикасан и за неуравнотежена стабла.

4.5. За С-АВЛ чвор v рећи ћемо да је ПС-АВЛ чвор ако ни један његов потомак није С-АВЛ чвор. У стаблу постоји бар један С-АВЛ чвор акко у њему постоји бар један ПС-АВЛ чвор. Проблем се решава индукцијом, односно рекурзивном процедуром која за подстабло са датим кореном одређује висину и проверава да ли у њему постоји неки С-АВЛ чвор (и означава све ПС-АВЛ чворове у случају потврдног одговора). Процедура, примењена на корен стабла, генерише рекурзивне позиве за лево, односно десно подстабло. Тиме су у подстаблима означени сви ПС-АВЛ чворови — ако их има; ако С-АВЛ чворова нема ни у левом ни у десном подстаблу, а висине подстабала су једнаке, корен се означава као ПС-АВЛ чвор; висина стабла (ако има више од једног чвора) добија се повећавањем за један веће од висина подстабала.

4.6. Може се применити алгоритам за рекурзивно израчунавање фактора равнотеже и висина свих чворова стабла (видети одељак 4.7), у коме за сваки израчунати фактор равнотеже чвора проверава да ли је он једнак 1; ако се за било који чвор добије вредност различита од 1, стабло није Л-АВЛ стабло.

4.7. Проблем се може решити рекурзивним алгоритмом. Проверава се да ли су оба стабла непразна; ако јесу, проверава се да ли оба корена имају левог сина, а ако га имају, рекурзивно се проверава да ли су једнака лева подстабла. Ако јесу, проверава се да ли оба корена имају десног сина, а ако га имају, рекурзивно се проверава да ли су једнака десна подстабла.

4.8. Који од сабирака $S[1], S[2], \ldots, S[n]$ улазе у суму K може се описати низом R дужине n, таквим да је $R[i] = true$ акко $S[i]$ учествује у суми. Компоненте $R[i]$ израчунавају се редом за $i = n, n-1, \ldots, 1$, коришћењем делимичне суме k, чија је почетна вредност $k = K$. У петљи се извршавају две операције $R[i] := P[i,k].Pripada;\ k := k - S[i];$.

4.9. Видети слику 5.

begin
  $i := 0; j := 0; k := 0;$ {индекси текуће прекидне тачке $f'(x)$, $f''(x)$ и $f(x)$}
  while $i \le k$ **and** $j \le l$ **do**
    $x := \min\{x'_{i+1}, x''_{j+1}\};$ {наредна прекидна тачка x}
    if $x \le x'_{i+1}$ **then** $i := i + 1;$ {поправка вредности $f'(x) = f'(x'_i) = h'_i$}
    if $x \le x''_{j+1}$ **then** $j := j + 1;$ {поправка вредности $f''(x) = f''(x''_j) = h''_j$}
    $H := \max\{h'_i, h''_j\};$ {вредност $f(x)$ после новог прекида x}
    if $H \ne h_k$ **then**
      $k := k + 1; x_k := x; h_k := H$ {нова прекидна тачка $f(x)$}
end

Рис. 5. Уз задатак 4.9.

4.10. Дато је n функција

$$g_i(x) = G_{L_i,D_i,V_i}(x) = \begin{cases} 0, & x < L_i \\ V_i + (x - L_i), & L_i \le x < L_i + \delta_i \\ V_i + \delta_i + x - (L_i + \delta_i), & L_i + \delta_i \le x < D_i \\ 0, & x \ge D_i \end{cases} .$$

где је $\delta_i = (D_i - L_i)/2$. Максимум неколико оваквих функција је фукција облика

$$g_{(x_1,h_1,k_1),(x_2,h_2,k_2),\ldots,(x_m,h_m,k_m)}(x)$$

са прекидним тачкама (x_i, h_i) и нагибом $k_i \in \{0, \pm 1\}$ у интервалу $[x_i, x_{i+1})$, $0 \le i \le m$. Другим речима, вредност функције у тачки $x \in [x_i, x_{i+1})$ је $h_i + k_i(x - x_i)$. Због једноставности се може сматрати да је $x_0 = -\infty$, $h_0 = 0$, $k_0 = 0$, $x_{m+1} = +\infty$. Проблем се такође решава декомпозицијом: n задатих функција деле се у две једнаке групе, рекурзивно се за њих одређују две функције — максимуми, и на крају се у $O(n)$ корака налази максимум те две функције. Обједињавање две функције почиње обједињавањем њихових низова прекидних тачака, $O(n)$ корака; у интервалима између две узастопне прекидне тачке обе функције су линеарне, па се максимум лако одређује. Ако се две линеарне функције у интервалу секу, додаје се нова прекидна тачка, која интервал дели на два подинтервала. Обрада свих интервала извршава се за $O(n)$ корака. Укупна сложеност алгоритма је решење диференцне једначине $T(n) = 2T(n/2) + cn$, тј. $O(n \log n)$.

4.11. Решење је слично решењу проблема налажења подниза са максималним збиром, при чему овде треба даље појачати индуктивну хипотезу због промене знака производа елемената после множења новододатим негативним бројем. Претпостављамо да за низ x_1, x_2, $\ldots$, x_m, $m \le n$, знамо

- подниз $x_i, x_{i+1}, \ldots, x_j$ са максималним производом p_1;
- суфикс $x_r, x_{r+1}, \ldots, x_m$ са максималним производом p_2, и
- суфикс $x_s, x_{s+1}, \ldots, x_m$ са минималним негативним производом p_3.

После додавања броја x_{m+1}, нове вредности p_1, p_2, p_3 су

$$(p_2, p_3) := \begin{cases} (x_{m+1}p_2, x_{m+1}p_3), & x_{m+1} > 0 \\ (1, -1), & x_{m+1} = 0 \\ (x_{m+1}p_3, x_{m+1}p_2), & x_{m+1} < 0 \end{cases} ,$$

$$p_1 := \max\{p_1, p_2\}.$$

Без смањења општости може се претпоставити да су бројеви x_i различити од нуле: у противном нуле раздвајају низ на поднизове, за које се проблем решава независно, јер су сви производи који прелазе границу неког од ових поднизова, једнаки нули.

4.12. Проблем се решава слично као налажење фактора равнотеже чворова графа (одељак 4.7). Индуктивна хипотеза је да за стабла са мање од n чворова умемо да израчунамо висину корена, да означимо све критичне чворове, а тиме и да установимо да ли унутар стабла постоји неки не-АВЛ чвор (а тиме и неки критичан чвор). За дато стабло са n чворова примењује се индуктивна хипотеза на његово лево и десно подстабло, после чега се може израчунати висина корена, проверити да ли је корен критичан чвор, означити га ако јесте, а тиме и установити да ли унутар целог стабла постоји неки не-АВЛ чвор. Трајање обраде корена је $O(1)$, па је сложеност рекурзивног алгоритма $O(n)$.

4.13. Решење је једноставно за стабло са једним или два чвора. Претпоставимо да умемо да решимо проблем за стабла са n чворова, и нека је дато стабло T са $n+1$ чвором. Нека је v произвољни лист у T (чвор који је повезан са само још једним чвором), и нека је wједини чвор суседан са v. Уклањањем v из T добијамо стабло T' са n чворова, за које према индуктивној хипотези умемо да израчунамо матрицу свих растојања у $O(n^2)$ корака. Растојање од v до произвољног чвора u из T' је за један веће од растојања од w до u; другим речима, сваки елеменат наредне, $(n+1)$-е врсте матрице израчунава се помоћу $O(1)$ операција.

Није тешко трансформисати овај алгоритам у нерекурзивни, формирајући најпре у $O(n)$ корака два низа чворова u_i, v_i таквих да се полазећи од чвора $u_1 = v_2$ у i-том кораку чвор u_i повезује граном са чвором v_i, $i = 2, 3, \ldots, n$, и тако добија T. При томе је увек $v_i \in \{u_1, u_2, \ldots, u_{i-1}\}$. Врста матрице која одговара чвору u_i (i-та врста) добија се тако што се одговарајући елементи врсте чвора v_i (дакле једне од претходних врста) повећају за један; изузетно, дијагоналним елементима додељује се вредност 0.

4.14. Стабло са дијаметром 0 је један (изоловани) чвор; стабло са дијаметром 1 чине два чвора повезана граном. Стабло са више од два чвора има дијаметар најмање два. Ако је $\delta = \delta(T)$ дијаметар стабла T и чворови u, v су такви да им је растојање $d(u, v) = \delta$, онда су чворови u, v листови, тј. чворови са по само једним суседом (у противном, ако нпр. u није лист, пут између u и v могао би се продужити другом граном из u до чвора нпр. w, и било би $d(w, v) = \delta + 1$, супротно претпоставци). Ако се од стабла T са бар три чвора формира стабло T' уклањањем свих листова, онда су са сваког пута између два листа у T уклоњени први и последњи чвор, па је $\delta(T') = \delta(T) - 2$. Исти поступак може се поновити са стаблом T', ако је број његових чворова бар 3, итд. На крају се долази или до изолованог чвора ($\delta_0 = 0$), или до два чвора повезана граном ($\delta_0 = 1$), а дијаметар полазног стабла је $\delta_0 + 2k$, где је k број итерација. Сложеност алгоритма је $O(n)$, где је $n = |V|$, јер је за уклањање сваког чвора потребно извршити $O(1)$ операција.

4.15. За решавање проблема ранца (одређивање подскупа бројева $k_1, k_2, \ldots, k_n$ са сумом K) довољан је један радни вектор A дужине K, у коме са за свако k, $0 \le k \le K$, памти само последњи сабирак у суми једнакој k, или се вредношћу -1 означава да не постоји подскуп са збиром k. Најпре се решава проблем са 0 сабирака, стављањем $A[0] := 0$ и $A[k] := -1$ за $k = 1, 2, \ldots, K$ (што значи да је без иједног сабирка могућ само збир 0). Затим се за $i = 1, 2, \ldots, n$ праве сви могући збирови у које улази нови сабирак k_i: за све вредности $k = 0, 1, 2, \ldots, K$, такве да је $A[k] \ge 0$, ако је $k + k_i \le K$, ставља се $A[k + k_i] := k_i$, чиме се региструје да је могуће добити збир $k + k_i$ са последњим сабирком k_i (претходно је збир k добијен као збир неког подскупа скупа $k_1, k_2, \ldots, k_{i-1}$). Ако се на крају добије $A[K] \ge 0$, онда постоји подскуп сабирака са сумом K, и тај подскуп је лако реконструисати: први сабирак је $A[K]$; наредни је $A[K - A[K]]$, итд.

4.16. За решавање ове варијанте проблема ранца може се искористити матрица $P[i, j]$ са пољима $Postoji$ и $Pripada$ из алгоритма $Ranac$ (слика 4.11). Треба ставити $P[i, k].Postoji :=$ $true$ акко вредност $true$ има било који од елемената $P[i-1, k]$, $P[i-1, k-k_i]$, $P[i-1, k-2k_i]$, ... Елегантније решење је ставити $P[i, k].Postoji := true$ акко ($P[i-1, k].Postoji := true$ или ($k - k_i \ge 0$ и $P[i, k - k_i].Postoji := true$). Ако је од услова у загради испуњен други, и само онда ставља се $P[i, k].Pripada := true$.

4.17. И овај проблем може се решити применом динамичког програмирања. За сваки пар (i, k), $0 \le i \le n$, $0 \le k \le K$, нека је $C[i, k]$ највећа могућа вредност ранца са садржајем тежине k, ако се у њему могу наћи само првих i предмета, и нека је $\lambda(i, k)$ број примерака предмета типа i у неком од садржаја ранца са вредношћу $C[i, k]$. Гранични услови за матрице C и λ су $C[i, 0] = \lambda[i, 0] = 0$, $C[0, k] = \lambda[0, k] = 0$, $0 \le i \le n$, $0 \le k \le K$. Подскуп првих i елемената са сумом k, такав да има највећу вредност, може се добити на један од следећа два начина: подскуп садржи, или не садржи i-ти елеменат. У првом, односно другом случају највећа вредност подскупа је $C[i-1, k]$ (при чему је $\lambda[i, k] = 0$), односно $C[i, k - k_i] + v_i$ (при чему је $\lambda[i, k] = \lambda[i, k - k_i] + 1$). Према томе, $C[i, k] = \max\{C[i-1, k], C[i, k - k_i] + v_i\}$, а $\lambda[i, k]$ има вредност 0 или $\lambda[i, k - k_i] + 1$, у зависности од тога да ли је већа прва или друга вредост у изразу за $C[i, k]$. На тај начин се помоћу $O(nK)$ операција израчунавају комплетне матрице C и λ. На крају се рекурзивно израчунавају бројеви a_i коришћењем помоћног низа w. Најпре је $a_n = \lambda[n, K]$, $w_n = K$, а затим $w_{i-1} = w_i - \lambda[i, w_i]k_i$, $a_{i-1} = \lambda[i-1, w_{i-1}]$, за $i = n, n-1, \ldots, 2$.

4.18. Збир елемената у траженим подскуповима треба да буде $S/2$, па S мора да буде паран број. За налажење подскупа са сумом $S/2$ може се искористити обичан алгоритам за решавање проблема ранца, (слика 4.11) сложености $O(n \cdot S/2) = O(nS)$.

4.19. Претпоставимо да постоји подскуп датог скупа $k_1, k_2, \dots, k_n$ са сумом k. Црну кутију примењујемо на низ $k_1, k_2, \dots, k_{n-1}$; ако је одговор "да", онда постоји сума–решење у коју не улази k_n; у противном, k_n обавезно улази у суму, и потребно је одредити подскуп скупа $k_1, k_2, \dots, k_{n-1}$ са сумом $k - k_n$. У оба случаја смањен је улаз једном употребом црне кутије, и установљено је да ли k_n улази у збир.

4.20. Нека је h_n најмањи број потеза потребних да се n дискова пребаци са штапића A на штапић B. Очигледно је $h_1 = 1$. Претпоставимо да знамо решење за проблем са $n-1$ диском. Проблем са n дискова на штапићу A *мора* се решавати тако да се највећи диск не дира све док се $n-1$ мањих дискова не пребаци у h_{n-1} потеза на штапић C — тек тада се највећи диск може преместити на штапић B. После тога се на њега могу наслагати остали дискови, што се може извести најмање за наредних h_{n-1} потеза. Дакле, проблем са n дискова може се решити за $h_n = 2h_{n-1} + 1$ потеза. Ова диференцна једначина може се преписати у облику $h_n + 1 = 2(h_{n-1} + 1)$, одакле је $h_n + 1 = 2(h_{n-1} + 1) = 2^2(h_{n-2} + 1) = \cdots = 2^{n-1}(h_1 + 1) = 2^n$, односно $h_n = 2^n - 1$.

4.21. Проблем се решава индукцијом, као и основна варијанта, користећи запажање да се највећи диск не може померити док задати штапић не буде празан, а остали дискови сложени на трећем (преосталом) штапићу. Прецизније, постоје две могућности: или је највећи диск на дну задатог штапића, или се налази на дну погрешног штапића. У првом случају само треба на највећи диск рекурзивно преместити осталих $n-1$ дискова — највећи диск не омета њихова премештања због своје величине. У другом случају се $n-1$ мањих дискова рекурзивно (на најједноставнији начин) премешта на трећи диск, највећи диск се премешта на задати штапић, па се други пут рекурзивно (као у основној варијанти проблема!) $n-1$ мањих дискова премешта на задати штапић.

13.5. Алгоритми за рад са низовима и скуповима

5.1. B може да постави серију питања "да ли је 2^i веће од n", за $i = 1, 2, \dots, m$, где је $m = \lfloor \log_2 n \rfloor + 1$. После тога зна се да је $2^{m-1} \le n < 2^m$, па следи серија питања која одговара бинарној претрази интервала $(2^{m-1}, 2^m]$ величине 2^{m-1} — укупно $m-1$ питање. Укупан број питања је $2m - 1 = 2\lfloor \log_2 n \rfloor + 1$.

5.2. Ако са $T(n)$ означимо очекивани број питања за погађање броја из интервала од n узастопних природних бројева (очекивани број питања не зависи од почетка интервала!), онда $T(n)$ задовољава следеће диференцне једначине

$$T(2k) = 1 + \tfrac{k}{2k}T(k) + \tfrac{k}{2k}T(k) = 1 + T(k), \quad k \ge 1$$

(интервал од $2k$ бројева се помоћу једног питања бројем m разбија на два подинтервала од по k бројева са једнаким вероватноћама $k/(2k) = 1/2$), односно

$$T(2k+1) = 1 + \tfrac{k+1}{2k+1}T(k+1) + \tfrac{k}{2k+1}T(k), \quad k \ge 1$$

(интервал од $2k+1$ бројева се аналогно разбија на један интервал од $k+1$ и један интервал од k бројева, при чему су вероватноће да изабрани број буде у првом, односно другом интервалу редом $(k+1)/(2k+1)$, односно $k/(2k+1)$). Пошто је $T(1) = 0$, за $T(97)$ добија се

$$\begin{aligned}
& 1 + \tfrac{49}{97}T(49) + \tfrac{48}{97}T(48) = 1 + \tfrac{49}{97}(1 + \tfrac{25}{49}T(25) + \tfrac{24}{49}T(24)) + \tfrac{48}{97}(1 + T(24)) \\
= \; & 2 + \tfrac{25}{97}T(25) + \tfrac{72}{97}T(24) = 2 + \tfrac{25}{97}(1 + \tfrac{13}{25}T(13) + \tfrac{12}{25}T(12)) + \tfrac{72}{97}(1 + T(12)) \\
= \; & 3 + \tfrac{13}{97}T(13) + \tfrac{84}{97}T(12) = 3 + \tfrac{13}{97}(1 + \tfrac{7}{13}T(7) + \tfrac{6}{13}T(6)) + \tfrac{84}{97}(1 + T(6)) \\
= \; & 4 + \tfrac{7}{97}T(7) + \tfrac{90}{97}T(6) = 4 + \tfrac{7}{97}(1 + \tfrac{4}{7}T(4) + \tfrac{3}{7}T(3)) + \tfrac{90}{97}(1 + T(3)) \\
= \; & 5 + \tfrac{4}{97}T(4) + \tfrac{93}{97}T(3) = 5 + \tfrac{4}{97}(1 + T(2)) + \tfrac{93}{97}(1 + \tfrac{2}{3}T(2) + \tfrac{1}{3}T(1)) \\
= \; & 6 + \tfrac{1}{97}T(2)(4 + 93 \cdot \tfrac{2}{3}) = 6 + \tfrac{66}{97}.
\end{aligned}$$

5.3. Ако је у неком тренутку тражена страна x у опсегу између страна m и n, и ако је књига отворена на непарној страни y, онда је по услову

$$-0.1(n-m+1) \leq y - \lfloor \tfrac{m+n}{2} \rfloor \leq 0.1(n-m+1).$$

Ако је $x = y$ или $x = y+1$, тражена страна је пронађена; у противном треба претражити или опсег од m до $y-1$ (дужине $y-m$) или опсег од $y+2$ до n (дужине $n-y-1$). Користећи очигледне неједнакости

$$\tfrac{m+n}{2} - \tfrac{1}{2} \leq \lfloor \tfrac{m+n}{2} \rfloor \leq \tfrac{m+n}{2},$$

добијамо да је величина новог опсега мања или једнака од

$$\begin{aligned} & \min\{y-m, n-y-1\} \leq \\ = \; & \min\left\{\lfloor \tfrac{m+n}{2} \rfloor - m + 0.1(n-m+1), n-1-\lfloor \tfrac{m+n}{2} \rfloor + 0.1(n-m+1)\right\} \\ = \; & \min\left\{\tfrac{m+n}{2} - m + 0.1(n-m+1), n-1-\tfrac{m+n}{2} + \tfrac{1}{2} + 0.1(n-m+1)\right\} \\ = \; & \min\left\{0.6(n-m)-0.1, 0.6(n-m)-0.4\right\} = 0.6(n-m)-0.4. \end{aligned}$$

Ако са a_k означимо величину опсега страна после k отварања књиге, онда је $a_0 = 1998$, $a_k \leq 0.6a_{k-1} - 0.4$, односно $a_k + 1 \leq 0.6(a_{k-1}+1) \leq 0.6^2(a_{k-2}+1) \leq \cdots \leq 0.6^k(a_0+1)$. Тражење се завршава најкасније у тренутку кад је $a_k \leq 1$, односно $0.6^k \cdot 1999 \leq 2$, или $k \geq 13.52$. Књигу треба отворити највише 14 пута.

5.4. Може се искористити бинарна претрага; улазни текст подели се на две половине и стартује се програм са првом половином текста на улазу. Ако дође до грешке, знамо да је грешка у тој половини, и настављамо даље на исти начин; у противном, грешка је у другој половини (претпостављамо да део текста који изазива грешку овом поделом није пресечен).

5.5. Интерполациона претрага добро ради ако вредност x_k не одступа много од неке линеарне функције од индекса k, $1 \leq k \leq n$. Зато пример низа на коме интерполациона претрага лоше ради треба тражити у категорији јако "нелинеарних" монотоних низова. На пример, посматрајмо низ $x_k = k!$, и нека у њему треба наћи број $z = (n-1)!$. После сваког корака се лева граница интервала индекса помера само за 1. Заиста, ако је $k < n-1$ и полази се од интервала $x_k = k!$, $x_n = n!$, следећа вредност леве границе интервала за индексе је $k+1$, јер је

$$k+1 \leq \left\lceil k + \tfrac{(n-1)!-k!}{n!-k!}(n-k) \right\rceil = \left\lceil k+1 - \tfrac{k(n-1)!+k!(n-k-1)}{n!-k!} \right\rceil \leq k+1$$

5.6. Пример се може конструисати индукцијом. Треба обезбедити да на сваком нивоу рекурзије највећа два броја у низовима који се обједињавају буду у различитим низовима. Индуктивну хипотезу да се највећа два броја налазе у првој и другој половини низа проширујемо захтевом да се највећи број налази на последњем месту у низу. За $k = 1$ низ $1, 2$ задовољава ове услове. Ако је $x_1, x_2, \ldots, x_n$ низ са траженим особинама, онда и низ од $2n = 2^{k+1}$ елемената

$$2x_1 - 1, 2x_2 - 1, \ldots 2x_n - 1, 2x_1, 2x_2, \ldots 2x_n,$$

има те особине: обе половине овог низа имају исте међусобне односе елемената као и низ $x_1, x_2, \ldots, x_n$, па сва обједињавања при њиховом сортирању по индуктивној хипотези садрже максимални број упоређивања. Поред тога, највећи бројеви $2x_n - 1$ и $2x_n$ (x_n је по индуктивној хипотези највећи међу бројевима $x_1, x_2, \ldots, x_n$) се налазе у различитим поднизовима, а највећи од $2n$ бројева, $2x_n$ налази се на последњем месту у низу.

5.7. Сортирање се може свести на максимум правоугаоника. За дате бројеве a_1, a_2, $\ldots$, a_n најпре се израчуна $a = \min\{a_i \mid 1 \leq i \leq n\}$ и $b = \max\{a_i \mid 1 \leq i \leq n\}$, па се за правоугаонике $(a_i - a, b - a, a_i - a)$, $1 \leq i \leq n$ (са висинама једнаким x-координати левог краја $a_i - a$ и "поравнатим" десним крајевима $b - a$) одреди њихов максимум. Добијена функција $f_{(x_1,x_1),(x_2,x_2),\ldots(x_n,x_n)}(x)$ одређује сортирани редослед бројева $a_1, a_2, \ldots, a_n$: то је $x_1 + a, x_2 + a, \ldots x_n + a$ (леви горњи углови свих правоугаоника леже на правој $y = x$).

5.8. На крају сваког проласка кроз основну петљу алгоритма на на слици 8) тачно је следеће тврђење: $x[i] \leq pivot$ за $1 \leq i \leq L$ и $x[i] > pivot$ за $D \leq i \leq n$, што се лако доказује индукцијом по броју пролазака кроз петљу. Поред тога, у сваком проласку кроз петљу се L

повећава, а D смањује најмање за 1, па после највише $n/2$ пролазака кроз петљу мора да наступи услов за искакање из петље $L \geq D$. Прецизније, у том тренутку је $D = L - 1$, јер се L зауставља на вредности индекса D затеченој из претходног проласка кроз петљу, а онда се D смањује само за 1. Пошто је показивач L у последњем проласку "прешао" све чланове низа до индекса D, закључује се да је на крају $x[i] \leq pivot$ за $i \leq D$ и $x[i] > pivot$ за $i \geq D+1$.

5.9. Претпостављамо да се ради са промењеним алгоритмом *Razdvajanje* (слика 8), тако да ако пивот није једнак $X[Levi]$, онда се најпре замењује са $X[Levi]$. Нека је $X[1] = 1$, $X[n] = 2$, и нека су сви остали елементи већи од 2. У том случају јасно је да ће пивот бити $X[n]$. Промењени алгоритам *Razdvajanje* ће најпре заменити $X[1]$ са $X[n]$ (пивот на почетак низа), а онда ће заменити $X[n]$ са $X[2]$ (јер је сада $X[n] = 1 < 2$ и $X[2] > 2$). Пошто су сви остали елементи већи од 2, нема других замена. Резултат раздвајања је долазак 1 и 2 на њихова права места на почетку низа. Сортирање раздвајањем наставља се рекурзивно на низу од трећег до последњег елемента. Да би се наставило на исти начин, трећи најмањи елеменат треба да буде на позицији 3, а (пошто је елеменат на последњој позицији у низу после раздвајања на почетку био на другој позицији) четврти најмањи елеменат на позицији 2. Преме томе, ако је n парно, низ бројева $1, 4, 3, 6, 5, \ldots, n, n-1, 2$ (на крајевима су 1 и 2, а на позицијама $2i$, $2i+1$ налазе се бројеви $2i+2$, $2i+1$ за $1 \leq i \leq n/2 - 1$) се применом сортирања раздвајањем уређује помоћу $\Omega(n^2)$ упоређивања.

5.10. Поделимо скуп на два подскупа са једнаким бројем елемената. Решивши рекурзивно оба потпроблема, добијамо два минимална (m_1, m_2) и два максимална елемента (M_1, M_2) у подскуповима. Максимални елеменат у полазном скупу је $\max\{M_1, M_2\}$, а минимални је $\min\{m_1, m_2\}$, тј. за обједињавање два решења довољна су два упоређивања. Ако са $T(n)$ означимо број потребних упоређивања за улаз величине n, онда се за парно n добија $T(n) = 2T(n/2) + 2$, $T(2) = 1$. Према томе, $T(2^k) + 2 = 2(T(2^{k-1}) + 2) = 2^2(T(2^{k-2}) + 2) = \cdots = 2^{k-1}(T(2) + 2) = 3 \cdot 2^{k-1} - 2$, тј. за $n = 2^k$ је $T(n) = \frac{3}{2}n$.

5.11. Резултат је приказан следећом табелом (f_i је учестаност појављивања знака c_i, l_i је дужина одговарајуће кодне речи у битима, $i = 1, 2, \ldots, 9$). Укупан број употребљених бита је 64, у односу на 168 бита ако се употреби ASCII кôд (осмобитни), односно $21 \times 4 = 84$ бита ако се сваки знак кодира са по 4 бита.

i	c_i	f_i	l_i	кôд
1	a	4	2	00
2	r	4	2	01
3		4	3	111
4	b	2	3	100
5	v	2	4	1100
6	d	2	4	1101
7	n	1	4	1011
8	h	1	5	10100
9	m	1	5	10101

5.12. Матрица растојања подстрингова A и B, заједно са стрелицама које приказују едит операције најмање цене дата је табелом 1:

У доњем десном углу табеле проналазимо да је минимални број едит операција 6. До доњег десног угла долази се само операцијом →. Идући даље уназад, видимо да се од горњег левог до доњег десног угла табеле може доћи нпр. низом операција ↘↘↘↘↘↘↘↘→→, што одговара следећем низу трансформација:

a	a	b	c	c	b	b	a	ϕ	ϕ	
b	a	a	c	b	a	b	a	c	c	
1+	0+	1+	0+	1+	1+	0+	0+	1+	1	= 6.

5.13. Нека је $n_1 = \lfloor n/2 \rfloor$. Посматрајмо вредност $w = A[n_1]$. Ако је $w \geq z$, онда се z мора појавити у низу $A[1]$, $A[2]$,... $A[n_1]$ (разлика узастопних чланова низа је највише 1, па ни једна

B / A	0	1 b	2 a	3 a	4 c	5 b	6 a	7 b	8 a	9 c	10 c
0	**0**	→ 1	→ 2	→ 3	→ 4	→ 5	→ 6	→ 7	→ 8	→ 9	→ 10
1 a	↓ 1	↘ **1**	↘ 1	↘ → 2	→ 3	→ 4	↘ → 5	→ 6	↘ → 7	→ 8	→ 9
2 a	↓ 2	↘ ↓ 2	↘ **1**	↘ 1	→ 2	→ 3	↘ → 4	→ 5	↘ → 6	→ 7	→ 8
3 b	↓ 3	↘ 2	↓ 2	↘ ↓ **2**	↘ 2	↘ 2	→ 3	↘ → 4	→ 5	→ 6	→ 7
4 c	↓ 3	↓ 3	↘ ↓ 3	↘ ↓ 3	↘ **2**	↘ ↓ → 3	↘ 3	↘ → 4	↘ → 5	↘ 5	↘ → 6
5 c	↓ 5	↓ 4	↘ ↓ 4	↘ ↓ 4	↘ ↓ 3	↘ **3**	↘ ↓ → 4	↘ 4	↘ → 5	↘ 5	↘ 5
6 b	↓ 6	↘ ↓ 5	↘ ↓ 5	↘ ↓ 5	↓ 4	↘ 3	↘ → **4**	↘ 4	↘ → 5	↘ ↓ → 6	↘ ↓ 6
7 b	↓ 7	↘ ↓ 6	↘ ↓ 6	↘ ↓ 6	↓ 5	↘ ↓ 4	↘ 4	↘ **4**	↘ → 5	↘ → 6	↘ ↓ → 7
8 a	↓ 8	↓ 7	↘ 6	↘ 6	↓ 6	↓ 5	↘ 4	↘ ↓ → 5	↘ **4**	→ **5**	→ **6**

Таблица 1. Уз задатак 5.12

вредност не може бити прескочена). У противном, ако је $w < z$, онда се z мора појавити у низу $A[n_1 + 1]$, $A[n_1 + 2]$,... $A[n]$. У оба случаја се проблем своди на претрагу скупа од највише $\lceil n/2 \rceil$ елемената, сложеност $T(n)$ задовољава диференцну једначину $T(n) \le T(\lceil n/2 \rceil) + 1$, $T(1) = 0$, чије је решење $T(n) \le \lceil \log_2 n \rceil$ (видети решење задатка 2.10).

5.14. Може се искористити теоријско-информациона граница. Број могућих одговара на питање постављено у задатку 5.13 је n (z може бити једнако неком од бројева $A[1]$, $A[2]$,... $A[n]$). Према томе, стабло одлучивања које решава овај проблем мора имати n листова (где i-ти лист одговара решењу $z = A[i]$). Висина таког стабла одлучивања је најмање $\lceil \log_2 n \rceil$. Сложеност алгоритма из решења задатка 5.13 једнака је овој доњој граници, што значи да је тај алгоритам оптималан у најгорем случају.

5.15. Може се употребити број, за један већи од највећег броја у скупу S. Алгоритам сложености мање од $\Omega(n)$ не може да решава овај проблем за све улазе, јер не може да прегледа свих n елемената скупа S.

5.16. (а) Треба сортирати скуп S а затим за свако $z \in S$ извршити бинарну претрагу скупа S за бројем $x - z$.

(б) Проблем се може решити индукцијом, са следећом индуктивном хипотезом: умемо да решимо проблем за сортирани скуп од $< n$ елемената. Нека је дат растуће уређени низ S од n елемената. Посматрајмо збир $y = S[1] + S[n]$. Ако је $y = x$, проблем је решен. Ако је $y > x$, онда $S[n]$ не може бити део решења, јер за свако i важи $S[n]+S[i] > S[n]+S[1] = y > x$; $S[n]$ се дакле може елиминисати (извођењем $O(1)$ операција) и преостали проблем решити индукцијом. Ако је $y < x$, онда слично $S[1]$ не може бити део решења ($S[1]+S[i] < S[1]+S[n] = y < x$), па се елиминише $S[1]$. Размотавајући рекурзију, видимо да се у алгоритму користе два показивача, *Levi* (почетна вредност 1) и *Desni* (почетна вредност n). У петљи (чије извршавање траје све док је $Levi < Desni$ и $y \ne x$) се проверава сума $y = S[Levi] + S[desni]$; ако је $y = x$, проблем је решен, а ако је $y > x$ ($y < x$) декрементира се *Desni* (инкрементира се *Levi*).

5.17. Нека у првом скупу има k, а у другом $n - k$ елемената. Означимо елементе првог, односно другог скупа, после сортирања у растућем редоследу са $a_1, a_2, \ldots, a_k$, односно

$b_1, b_2, \ldots, b_{n-k}$. За свако i, $1 \le i \le n-k$, низ $a_1 + b_i$, $a_2 + b_i$, $\ldots$, $a_k + b_i$ је растући, па се бинарном претрагом помоћу $O(\log n)$ (прецизније $\lceil \log_2 k \rceil + 1$) упоређивања међу овим бројевима проналази x, или се утврђује да ни један од њих није једнак x. Значи, за проверу свих парова $a_i + b_j$ довољно је $O(n \log n)$ упоређивања; сложеност почетног сортирања је такође $O(n \log n)$, па је укупна сложеност алгоритма $O(n \log n)$.

Друго решење. Други део посла, после сортирање S_1 и S_2, може се обавити ефикасније, алгоритмом сложености $O(n)$. Посматрају се (и упоређују редом са x) суме $a_i + b_j$, при чему се најпре фиксира $i = 1$, а j узима вредности редом $n-k$, $n-k-1$, $\ldots$. Бира се такво j да је $a_1 + b_{j+1} > x \ge a_1 + b_j$; тада за све парове (r, s) такве да је $r \ge 1$ и $s > j$, важи $a_r + b_s > a_1 + b_j > x$, па остају само парови (r, s) за које је $1 \le r \le k$, $1 \le s \le j$. Ако такво j не постоји (тј. ако је $a_1 + b_{n-k} < x$ или $a_1 + b_1 > x$), посао је завршен, одговор је "не". У противном, помоћу $m = n - k - j + 1$ сабирања и упоређивања елиминисано је $m - 1$ највећих елемената b_{n-k}, b_{n-k-1}, $\ldots$, b_{j+1} скупа S_1. Настављајући на овај начин (повећавајући индекс r у суми $a_r + b_s$, па смањујући индекс s, итд.) долази се до решења. Укупан број сабирања и упоређивања при томе није већи од $2n$.

5.18. Низове елемената првог и другог скупа треба сортирати, па објединити, слично као при сортирању обједињавањем, само што се успут још и избацују поновљени елементи (они који се појављују у оба скупа).

5.19. Показаћемо да се тражени парови могу формирати тако да се за $k = 1, 2, \ldots, n/2$ упаре k-ти најмањи и k-ти највећи број из датог скупа; означимо овај распоред са A, а одговарајућу вредност s_{max} са $s_{max}(A)$. Да бисмо то доказали, претпоставимо супротно, да постоји неко друго упаривање (распоред B), такво да је $s_{max}(B) < s_{max}(A)$. Претпоставимо да је дати низ претходно сортиран у растућем редоследу. Ако у распореду B нису упарени x_1 и x_n, односно ако је x_1 у пару (x_1, x_i), а x_n у пару (x_j, x_n), онда се од распореда B може направити распоред B_1 заменом ова два пара паровима (x_1, x_n) и (x_i, x_j). Тада је $s_{max}(B_1) \le s_{max}(B)$, јер је $x_1 + x_n \le x_j + x_n \le s_{max}(B)$ и $x_i + x_j \le x_n + x_j \le s_{max}(B)$, тј. сви збирови парова у B_1 су мањи или једнаки од $s_{max}(B)$, па је и $s_{max}(B_1) \le s_{max}(B)$. Слично се од B_1 формира распоред B_2 у коме су упарени x_2 и x_{n-2}, при чему је $s_{max}(B_2) \le s_{max}(B_1)$, итд. После највише $n/2$ оваквих премештања парова долази се (преко низа распореда са нерастућим вредностима s_{max}) до распореда A, тј. $s_{max}(A) \le \cdots \le s_{max}(B_2) \le s_{max}(B_1) \le s_{max}(B)$, супротно претпоставци да је $s_{max}(B) < s_{max}(A)$.

5.20. Довољно је парове (a_i, x_i) сортирати (у месту) према "кључу" a_i (нпр. сортирање раздвајањем). Тиме пар (a_i, x_i) долази на позицију a_i, $i = 1, 2, \ldots, n$, што је и тражено,

5.21. На почетку треба d првих (минималних) елемената ставити у хип (тако да у корену буде најмањи елеменат). Затим се у сваком наредном кораку уклања из хипа најмањи елеменат (из корена) и умеће се следећи елеменат из оног низа коме је припадао уклоњени елеменат (због тога се у хипу мора за сваки елеменат чувати и податак из ког је низа потекао).

5.22. (а) Може се сваки елеменат уметнути у уравнотежено бинарно стабло претраге (нпр. АВЛ стабло). Сваки чвор у стаблу треба да садржи број појављивања елемената једнаких његовом кључу (при првом упису неког кључа у стабло бројач се поставља на нулу, а при сваком наредном упису истог елемента, бројач се инкрементира). Пошто различитих кључева има $O(\log n)$, број чворова у стаблу је $O(\log n)$, па је висина стабла $O(\log \log n)$. Затим се АВЛ стабло "исправља" (видети задатак 3.2), тј. његови елементи се у растућем редоследу копирају (и то толико пута колика је вредност одговарајућег бројача) у излазни низ дужине n.

(б) Захваљујући чињеници да је мали број различитих елемената, при сортирању су битно коришћене *вредности* елемената, па овај алгоритам није обухваћен моделом стабла одлучивања.

5.23. Нека је T_n уравнотежено стабло са n чворова, и нека је $f(n)$ збир висина свих чворова у T_n. Стабло T_{2n} добија се од стабла T_n додавањем листова $n+1, n+2, \ldots, 2n$ чије су висине 0. Висина чвора i $i = 1, 2, \ldots, n$, у T_{2n} тачно је за један већа од његове висине у T_n, па је $f(2n) = f(n) + n$. Поред тога, $f(2n+1) = f(2n)$. Помоћу ових диференцних једначина

лако се индукцијом доказује неједнакост $f(n) \leq n-1$, као и да једнакост важи акко је n степен двојке.

5.24. Сума висина је $T(n) = \sum_{i=1}^{n} \lfloor \log_2(n/i) \rfloor$. Претпоставимо да је $n = 2^k - 1$. Тада је $\lfloor \log_2(n/i) \rfloor = j$ за $i = 2^{k-j-1}, 2^{k-j-1}+1, \ldots 2^{k-j}-1$ (тј. у збиру се сабирак j појављује 2^{k-j-1} пут), $j = 0, 1, 2, \ldots, k-1$, па је $T(n) = \sum_{i=1}^{n} j 2^{k-j-1} = 2^k - k - 1 = O(n)$ (слична сума израчуната је у примеру 2.5). Ако $n+1$ није степен двојке, тј. $2^k - 1 < n < 2^{k+1} - 1$, онда се сваки сабирак у овој суми може повећати највише за један, па се цела сума повећава највише за n, и $T(n) = O(n)$.

5.25. Алгоритам се састоји од рекурзивног спуштања низ хип и бројања елемената већих од x, док се не прође цели хип, или се не изброји k елемената. Већих од x има k или више акко је k-ти највећи већи од x. Прецизније, користи се рекурзивна процедура која, примењена на (пот)хип са кореном у задатом чвору v полазног хипа

- упоређује вредност у корену v са x;
- ако је резултат поређења "веће", повећава се за један вредност промењиве m (која је статичка, односно заједничка за све нивое рекурзије);
- ако је $m > k$, излази се из процедуре;
- процедура се затим рекурзивно позива за оба сина чвора v.

Алгоритам најпре поставља почетну вредност $m := 0$ (број елемената хипа већих од x), а затим примењује описану процедуру на корен хипа. Број рекурзивних позива ограничен је са $O(k)$, независно од величине хипа. Меморијски простор $O(k)$ потребан је за стек, односно за памћење рекурзивних позива.

Друго решење. Описано решење се може формулисати и много једноставније, ако се прочита поглавље 6. Чворови са кључем већим од x чине једно подстабло полазног стабла, при чему, ако је кључ неког чвора $\leq x$, онда због особине хипа то исто важи и за све његове потомке. То подстабло може се обићи било којим поступком претраге, DFS или BFS, при чему се претрага завршава и пре комплетног обиласка ако се наброји више од k чворова са кључем већим од x (успут се прегледа највише k чворова са кључем $\leq x$).

5.26. Претпоставимо да алгоритам A проналази да је елеменат x_i k-ти највећи после низа упоређивања, чији резултати нису довољни да се установи да ли је неки други елеменат x_j мањи или већи од x_i. То значи да се резултати тих упоређивања неће променити ако се x_j замени бројем $y > x_i$ или бројем $z < x_i$. Међутим, у та два случаја не добија се исти k-ти највећи елеменат, што је у контрадикцији са претпоставком.

5.27. Идеја је посматрати $2k$ елемената истовремено. Почињемо са првих $2k$ елемената, и проналазимо њихову медијану (очекивани број корака је $O(2k) = O(k)$). Елементи већи од медијане се елиминишу. Затим посматрамо следећих k елемената и радимо исто. Поступак се састоји од око n/k итерација у којима се израчунава медијана $2k$ елемената, па је средња временска сложеност $O(n)$.

5.28. Рачунари P и Q могу да (без размене порука) сортирају своје скупове. Претпоставимо дакле да је $a_1 < a_2 < \cdots < a_n$ и $b_1 < b_2 < \cdots < b_n$. Означимо са $f(i)$ ранг елемента a_i у низу B, односно са $g(i)$ ранг елемента b_i у низу A. Елеменат a_i (односно b_i) је медијана уније акко је $i + f(i) = n$ (односно $i + g(i) = n$). Разумно је да поруке које рачунари размењују буду следећег типа: рачунар шаље један свој елеменат другом рачунару, а од њега добија ранг тог елемента у другом скупу. Функције $i + f(i)$, $i + g(i)$ су строго растуће, па се бинарном претрагом после највише $4 \log_2 n$ размењених порука проналази тражена медијана.

5.29. Различитих могућих одговора овде има $n+1$, па свако стабло одлучивања које решава овај проблем треба да има бар $n+1$ листова. Одатле непосредно следи тврђење задатка.

5.30. Идеја је конструисати таблицу h (видети одељак 5.6) за онај део узорка који је већ познат, и продужавати је по додавању наредних знакова. Ово није тешко, јер је конструкција таблице h слична обичном тражењу узорка у тексту.

5.31. Промена је минимална: сваки пут кад дође до неслагања, треба проверити да ли је уклопљен префикс узорка већи од дотле највећег: ако јесте, треба запамтити његову дужину и стартну позицију у тексту S.

5.32. Део (б) је уствари упутство за решавање дела (а). Најпре се израчунава едит растојање (и минимални број едит операција) између T и P, под претпоставком да замене низу дозвољене (може се користити алгоритам исти као на слици 5.25, изузев што се за цену замене узима вредност 2, као да је замена изведена једним уметањем и једним брисањем). LCS два низа је низ знакова који су преклопљени (који се не мењају, односно нису ни уметани ни брисани) при извођењу едит операција. SCS се од LCS добија додавањем свих уметнутих и обрисаних знакова.

5.33. Нови алгоритам се такође заснива на динамичком програмирању, односно израчунавању матрице свих едит растојања префикса $A(i)$ и $B(j)$, $0 \le i \le m$, $0 \le j \le n$, при чему се само мења диференцна једначина и прва врста матрице ($C[0,j] = 0$, $0 \le j \le n$),

$$C[i,j] = \begin{cases} \min\{C[i-1,j]+1, C[i,j-1]+1, C[i-1,j-1]+c(a_i,b_j)\}, & 1 \le i < m \\ \min\{C[i-1,j]+1, C[i,j-1], C[i-1,j-1]+c(a_i,b_j)\}, & i = m \end{cases} .$$

5.34. Проблем се може решити применом динамичког програмирања, уз помоћ тродимензионалне табеле M димензија $(m+1)\times(n+1)\times(p+1)$, чији су елементи M_{ijk} једнаки најмањем броју едит операција које префиксе $A(i)$, $B(j)$ и $C(k)$ трансформишу у исти низ, $0 \le i \le m$, $0 \le j \le n$, $0 \le k \le p$. Нека је

$$r(u,v) = \begin{cases} 0, & u = v \\ 1, & u \ne v \end{cases} ,$$

$$r(u,v,w) = \begin{cases} 0, & u = v = w \\ 1, & u = v \ne w \text{ или } u = w \ne v \text{ или } v = w \ne u \\ 2, & u \ne v \text{ и } v \ne w \text{ и } u \ne w \end{cases} .$$

Слично као у дводимензионалном случају, полази се од ивица ($M_{i,0,0} = i$, $M_{0,j,0} = j$, $M_{0,0,k} = k$), а онда се користи диференцна једначина

$$\begin{aligned} M_{i,j,k} = \min\{\, & M_{i,j,k-1}+1, M_{i,j-1,k}+1, M_{i-1,j,k}+1, \\ & M_{i,j-1,k-1}+r(b_j,c_k), M_{i-1,j,k-1}+r(a_i,c_k), M_{i-1,j-1,k}+r(a_i,b_j), \\ & M_{i-1,j-1,k-1}+r(a_i,b_j,c_k)\,\} . \end{aligned}$$

Прва група чланова одговара уметању истог знака у један од низова; друга група одговара уметању једног и евентуалној промени једног знака; трећа група одговара евентуалној промени једног или два знака. Елеменат $M_{m,n,p}$ једнак је траженом најмањем броју едит операција. Сложеност алгоритма је $O(n^3)$ јер је сложеност израчунавања појединих елемената табеле M $O(1)$. У општем случају за "уједначавање" k низова користи се k-димензионална табела, а сложеност алгоритма је $O(n^k)$ (прецизније, $O(n^k \cdot 2^k) = O((2n)^k)$).

5.35. Најједноставније је овај проблем решити обичним алгоритмом за упоређивање низова (слика 5.25), који је промењен тако да су почетне вредности $C[i,0]$ једнаке 0. Другим речима, брисање почетка A има цену 0 — управо оно што нам је потребно.

5.36. Тражени алгоритам може се конструисати индукцијом по n. Претпоставимо да се може добити случајна пермутација $p_1, p_2, \ldots, p_{n-1}$ скупа $\{1, 2, \ldots, n-1\}$ са униформном расподелом вероватноћа. Да би се добила случајна пермутација бројева $\{1, 2, \ldots, n\}$, треба најпре изгенерисати случајни број i, $1 \le i \le n$, кога треба ставити на последње место. Затим се на $n-1$ бројева $1, 2, \ldots, i-1, i+1, \ldots, n$ примењује пермутација $p_1, p_2, \ldots, p_{n-1}$. То обезбеђује да ће се све пермутације реда n добити са једнаком вероватноћом. Сложеност алгоритма је $O(n^2)$.

5.37. Нека су x_i, x_j два произвољна елемента E; ако је $x_i \ne x_j$ и оба ова елемента уклонимо из E, онда ако је у E постојао преовлађујући елеменат, постојаће и даље. Ако је $x_i = x_j$, онда памтимо само x_i и број његових пронађених копија, и узимамо из E један по један нови елеменат. Кад се наиђе на елеменат различит од x_i — уклањамо га, и број копија x_i смањујемо за 1 (све док евентуално не дођемо до нуле; тада се наставља на описани

начин, са два нова елемента). Према томе, у једном пролазу ($O(n)$ корака) елиминишу се сви елементи E сем евентуално једног. Провера овог кандидата за преовлађујући елемент захтева још $O(n)$ упоређивања, да би се одредила његова вишеструкост.

5.38. Може се искористити алгоритам сличан ономе из решења задатка 5.37, изузев што треба памтити три (уместо једног) различита кандидата. Нови елеменат упоређује се са кандидатима, па ако је различит од свих њих, њихове текуће вишеструкости се смањују за један (ако при томе вишеструкост неког од њих падне на нулу, он испада из списка кандидата). Ако тренутно има мање од три кандидата, онда се нови елеменат узима за новог кандидата, са текућом вишеструкошћу 1. На крају се преостали кандидати упоређују са свим осталим елементима, да би им се одредиле тачне вишеструкости.

13.6. Графовски алгоритми

6.1. Видети слику 6.

Улазна обрада:
    $v.Visina := 0$;
    $v.Faktor := 0$;
излазна обрада: {за чвор w, сина чвора v}
    $v.Visina := \max\{v.Visina, 1 + w.Visina\}$;
    if w је леви син чвора v **then**
        $v.Faktor := v.Faktor + 1 + w.Visina$
    else
        $v.Faktor := v.Faktor - 1 - w.Visina$;

Рис. 6. Уз задатак 6.1.

6.2. Проблем се може решити помоћу варијанте обиласка DFS (претраге у дубину). Рекурзивна процедура стартована из неког чвора v, означава v и проверава за све суседе v, сем онога из кога се дошло у v, да ли су означени. Ако је било који од њих означен, излази се из рекурзије са одговором "не". У противном се из свих суседа процедура стартује рекурзивно.

6.3. Нека је за $v \in V$ са $d(v)$, односно $d_T(v)$ означен степен чвора v у G, односно у T, и нека је $\delta(v) = d(v) - d_T(v)$. Проласком кроз E израчунавају се све вредности $d(v)$, $d_T(v)$, и формира се листа чворова са $\delta(v) = 0$ — само ти чворови могу бити "слепе улице" у DFS. Ако таквих чворова нема, T није регуларно DFS стабло графа G. У противном се v уклања из G, T и листе, уз одговарајуће поправке вредности d, d_T суседних чворова, и евентуално проширивање листе новим чворовима w са $\delta(w) = 0$. Тиме се проблем своди на мањи, који се по индуктивној хипотези може решити.

6.4. Нека је T стабло које задовољава услове задатка, и нека је e нека грана графа ван T. Пошто је T DFS стабло, грана e је повратна грана у односу на њега, тј. e повезује неки чвор са својим потомком. С друге стране, пошто је T истовремено и BFS стабло, грана e повезује два чвора на истом или суседним нивоима хијерархије у стаблу. Дошли смо до контрадикције, што значи да у графу не постоје гране ван стабла, односно граф који задовољава услове задатка је стабло.

6.5. Основа овог алгоритма је алгоритам за тополошко сортирање, који чува листу чворова улазног степена нула и са ње скида један по један чвор (заједно са суседним гранама, смањујући за један улазне степене чворова на њиховим крајевима, и додајући на листу евентуалне нове чворове улазног степена нула). Ако се у току извршавања испразни листа, а

нису елиминисани сви чворови графа, у преосталом графу сви чворови имају улазни степен бар један, што омогућује једноставно проналажење циклуса у њему за време $O(|E|)$. Заиста, бира се произвољни чвор v, означава, и проналази произвољан чвор w из кога води грана у v; поступак се понавља са чвором w, итд, све до наиласка на означен чвор, који затвара циклус. Показивачи "уназад" лако се формирају за $O(|E|)$ корака, обрадом једне по једне гране из листе повезаности.

6.6. Дијкстрин алгоритам за сваки наредни чвор додаје тачно једну грану која припада неком најкраћем путу — грану којом се продужује неки претходни најкраћи пут. На крају се добија повезани граф са $|V|$ грана, дакле стабло. Оријентисањем грана стабла ка новододатим чворовима, добија се коренско стабло са кореном v.

6.7. Није; видети пример на слици 7.

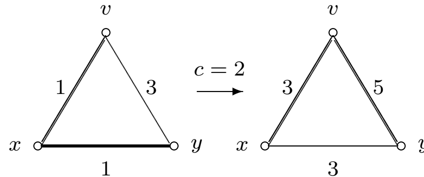


Рис. 7. Уз задатак 6.7.

6.8. Нетачно. У примеру на слици 8 алгоритам *Najkr_putevi* за најкраћи пут до z проглашава грану (v,z) дужине 3. Међутим, пут v,x,y,z има мању дужину 2. При томе граф нема циклус негативне тежине.

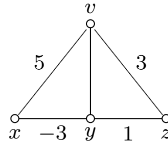
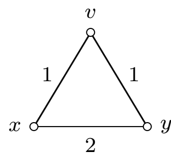


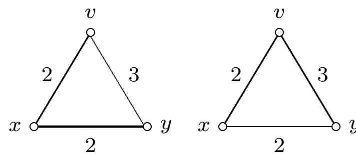
Рис. 8. Уз задатак 6.8.

6.9. У алгоритму за налажење MCST довољно је сваку појаву "∞","најмања","$<$", заменити редом са "0","највећа","$>$".

6.10. (а), (б) Видети примере на слици 9.



(а) MCST је истовремено и стабло најкраћих путева из v



(б) пример кад се MCST разликује од стабла најкраћих путева из v

Рис. 9. Уз задатак 6.10.

(ц) Не. Најкраћа грана из v припада и једном и другом стаблу. Ако она не би припадала MCST, онда би њеним додавањем у MCST био затворен циклус, у коме постоји бар једна

дужа грана из чвора v; заменом те гране најкраћом граном из v добија се повезујуће стабло мање цене, супротно претпоставци.

6.11. Најпре се формира максимално упаривање и листа неупарених чворова. Затим се упаривање, ако није савршено, повећава у петљи за по једну грану узимајући произвољна два неупарена чвора v_1 и v_2, тражећи грану упаривања (w_1, w_2) такву да од v_1, v_2 ка w_1, w_2 воде бар три гране ($O(|V|)$ операција, видети одељак 6.9.1), и замењујући грану (w_1, w_2) са две нове од v_1, v_2 ка w_1, w_2. Пошто се полазно упаривање мора повећати за мање од n грана, сложеност алгоритма је $O(|V|^2) = O(|V| + |E|)$.

6.12. Чворови произвољног стабла се могу индукцијом уредити на описани начин: за v_1 се узима произвољни лист, а затим се из стабла уклања v_1 са својом једином суседном граном; остали чворови се по индуктивној хипотези могу нумерисати. Обрнуто, граф који има ову особину, не може да садржи циклус, јер сви чворови у циклусу имају степен бар два; поред тога, индукцијом по броју чворова показује се да граф са овом особином мора бити повезан. Дакле, граф са овим особинама је стабло.

6.13. Полази се од циклуса формираног као у доказу тврђења да у Ојлеровом графу увек постоји Ојлеров циклус, одељак 6.2 (формира се тако што се, полазећи из произвољног чвора насумице бирају гране које чине пут, све док се не дође до полазног чвора). Нека је број чворова графа означен са n. Означе се гране циклуса, за сваки чвор се израчуна у посебном низу број њему суседних неозначених грана (његов Ц-степен, који је увек паран), и формира се листа чворова са позитивним Ц-степеном. Затим се, све док се та листа не испразни, узима неки чвор са листе, формира помоћни циклус од неозначених грана, полазећи од тог чвора (успут се означавају гране укључене у помоћни циклус и смањују Ц-степени чворова на њиховим крајевима, а ако Ц-степен неког чвора постане нула, чвор се уклања са листе), и обједињује са полазним циклусом — што није тешко, јер имају један заједнички чвор. У тренутку кад је листа празна, све гране графа су укључене у циклус. Споменуту листу згодно је реализовати у посебном вектору дужине n, у коме за елементе листе стоји јединица, а за остале чворове нула. Нови елементи листе узимају се пролазећи вектор слева удесно, па је довољан један пролазак кроз вектор.

6.14. По услову задатка граф је Ојлеров, или се састоји од компоненти повезаности које су Ојлерови графови. У свакој компоненти повезаности конструише се Ојлеров циклус, и све гране циклуса усмере се у истом смеру дуж пута (тако да се "надовезују"). Тада ће за сваки чвор графа његов улазни степен бити једнак његовом излазном степену.

6.15. Доказ се изводи индукцијом по броју грана графа, на исти начин као и доказ тврђења да у Ојлеровом графу увек постоји Ојлеров циклус, одељак 6.2 (разлика је само у томе што се полази од неког усмереног циклуса у графу).

6.16. (а) Збир степена свих чворова графа је паран број (једнак двоструком броју грана), јер се у тој суми свака грана рачуна два пута, за оба чвора на њеним крајевима. После избацивања парних сабирака, збир је и даље паран. Према томе, број непарних сабирака (тј. чворова са непарним степеном) мора бити паран.

(б) Прва фаза алгоритма је повезивање парова непарних чворова (чворова са непарним степеном) било каквим путевима (полазећи из непарног чвора и бирајући насумице суседне гране, раније или касније долази се до другог непарног чвора; уклањањем, односно означавањем изабраних грана, ова два чвора постају парни). Друга фаза је проширивање ових $k/2$ путева помоћним циклусима, добијеним на исти начин као у задатку 6.13, све дотле док има чворова степена већег од нуле, односно неозначених грана.

6.17. Може се конструисати произвољно повезујуће стабло графа, (нпр. DFS стабло) и у том стаблу изабрати произвољан лист.

6.18. Из произвољног чвора $a_1a_2\ldots a_{n-1}$ графа G_n излазе тачно две гране (ка чворовима $a_2a_3\ldots a_{n-1}0$ и $a_2a_3\ldots a_{n-1}1$); у њега улазе тачно две гране (из чворова $0a_1a_2\ldots a_{n-2}$ и $1a_1a_2\ldots a_{n-2}$). У графу G_n је сваки чвор $b_1b_2\ldots b_{n-1}$ достижан из сваког чвора $a_1a_2\ldots a_{n-1}$ — преко чворова редом $b_2b_3\ldots b_{n-1}a_1$, $b_3\ldots b_{n-1}a_1a_2$, $\ldots$, $b_{n-1}a_1a_2\ldots a_{n-2}$. Према томе, граф G_n је усмерени Ојлеров граф. Последица ове чињенице је да за сваки природан број n постоји бинарни де Бруијнов низ од N бита, јер се сваком Ојлеровом путу $a_0a_1\ldots a_{n-2}$,

$a_1a_2\dots a_{n-1},\dots,\ a_Na_0\dots a_{n-3}$ у графу G_n (он садржи сваку грану G_n тачно једном) једнозначно придружује бинарни де Бруијнов низ $a_0a_1\dots a_N$ — свакој грани у G_n одговара јединствена n-торка бита у овом низу.

6.19. Базни случај $n = 2$ је једноставан: тада је обавезно $d_1 = d_2 = 1$. Циљ је свести проблем на мањи, са $n-1$ бројева и ограничењима истог типа. Пошто су бројеви d_i позитивни и збир им је $2n-2$, за неко k је $d_k = 1$ (у противном би било $\sum d_i \geq 2n > 2n-2$). На сличан начин, за неко j је $d_j > 1$ (у противном би било $\sum d_i \leq n < 2n-2$ за $n > 2$). Због тога се може d_k удаљити из скупа, а d_j смањити за један, и тако добити регуларан проблем са $n-1$ бројева — сви бројеви су и даље природни, а збир им је $2n-4$. Стабло добијено применом индуктивне хипотезе на ове бројеве проширујемо једним листом степена $d_k = 1$ привезаним за чвор степена $d_j - 1$, после чега степен тог чвора постаје d_j. Тиме је решен задати проблем. Сложеност алгоритма је $O(n)$.

6.20. Формирају се две листе тројки (j, u_j, i_j): L_0 од тројки са $i_j = 0$ и L_1 од осталих тројки, уређених по опадајућим вредностима индекса j (због чега је $1, u_1, i_1$) последња тројка у L_1). За $n = 2$ могућ је само низ парова $(0, 1)$, $(1, 0)$, коме одговара коренско стабло од два чвора. Случај низа од n парова се на случај $n-1$ парова своди тако што се скине прва тројка (k, u_k, i_k) са листе L_0, а прва тројка (l, u_l, i_l) из L_1 замењује се са $(l, u_l, i_l - 1)$; ако је $i_l - 1 = 0$, онда се та тројка премешта из L_1 у L_0; памти се грана стабла од оца l ка сину k. Збир излазних степена је сада $n-2$, па се са конструкцијом стабла наставља рекурзивно. Сложеност алгоритма је $O(n)$.

6.21. Да би у G постојала оваква нумерација, мора да постоји бар један чвор v са излазним степеном 0; њему се може придружити редни број $n = |V|$. Чвор v уклањамо из графа заједно са гранама које улазе у њега (смањујемо излазне степене одговарајућих чворова). На преостали граф G' рекурзивно примењујемо исту процедуру (односно његови чворови се на основу индуктивне хипотезе могу нумерисати на жељени начин, или се може установити да је таква нумерација немогућа). Ако постоји нумерација G', добија се нумерација G. У противном се ни G не може нумерисати: ако би постојала нумерација G, онда би се она могла променити тако да у њој чвор v има редни број n (чворови који у G имају излазни степен 0 могу се нумерисати са неколико највећих бројева на произвољан начин), чиме се добија нумерација за G', супротно претпоставци.

6.22. Може се искористити претрага у ширину (BFS). Полази се од произвољног чвора, који се боји црвеном бојом и ставља у листу. Затим се, док се листа не испразни, скида наредни чвор v са листе (обојен!), означава, његови означени суседи се проверавају (да ли им је боја супротна од његове; ако некоме није, онда G није 2-обојив), а неозначени суседи се стављају на листу и боје бојом супротном од боје v. Ако граф није повезан, онда се описани алгоритам примењује посебно на сваку компоненту повезаности. Сложеност алгоритма је линеарна, $O(|V| + |E|)$.

6.23. Ако је G повезан, алгоритам из задатка 6.22 после избора боје произвољног чвора једнозначно одређује боје свих осталих чворова. Ако претпоставимо да G није једнозначно 2-обојив, односно да постоји прави подскуп $U \subset V$ чворова који се могу обојити било једном, било другом бојом, онда између чворова у U и $V \setminus U$ не може да постоји ни једна грана, тј. G није повезан.

6.24. Може се искористити алгоритам KMP за налажење узорка у тексту, у комбинацији са претрагом стабла у дубину. У току обиласка се за сваки чвор чува његов статус у односу на тренутно потенцијално уметање узорка. Прецизније, за чвор v његово поље $v.Stat$ једнако је величини максималног префикса узорка који се слаже са суфиксом пута од корена до v (управо ти подаци чувају се и у алгоритму KMP). Улазна обрада за чвор v обухвата додавање знака из чвора v (као у наредном кораку алгоритам KMP, и израчунавање $v.Stat$. Излазна обрада треба да $v.Stat$ уредно врати у претходно стање после напуштања v. Временска сложеност алгоритма је линеарна у односу на број грана стабла.

6.25. Да нема циклуса, имали бисмо посла са стаблом, за које се проблем лако решава (у њему се неки чвор изабере за корен, а онда се све гране усмере од корена). Циклус се може пронаћи претрагом у дубину, у тренутку кад се наиђе на прву (и јединУ) грану која води

ка већ посећеном (означеном) чвору. Циклус се може усмерити тако да постане усмерени циклус, и да сви његови чворови имају улазни степен 1. Све остале гране треба усмерити у смеру од циклуса. То се може постићи покретањем нове претраге у дубину од произвољног чвора на циклусу и усмеравањем грана у смеру којим су прелажене први пут у току претраге. Једина повратна грана усмерена је према корену стабла претраге, чији је улазни степен 1 (као и свих осталих чворова).

6.26. Ако је G неповезан граф, онда се посебно обрађује свака његова компонента повезаности, па се без смањења општости може претпоставити да је G повезан. Ако је G стабло, одговор је *не*, јер онда G има мање грана него чворова, недовољно да се сваком чвору обезбеди грана која улази у њега. У противном у G постоји циклус, који се може пронаћи, на пример, алгоритмом са слике 6.13. Нека су v, w произвољна два суседна чвора у том циклусу; после уклањања гране (v, w) из G, G остаје и даље повезан. Полазећи од чвора v стартује се DFS графа G, конструише се DFS стабло, а у том стаблу све гране се усмеравају од корена v. Грана (v, w) усмерава се од w ка корену v. Преостале грана могу се произвољно оријентисати.

6.27. На G треба применити претрагу у дубину, и формирати DFS стабло T. Пошто су све гране графа повратне у односу на T, могу се све гране стабла T усмерити од корена, а све остале гране ка чворовима ближим корену (тј. ка прецима).

6.28. Нека се тополошким сортирањем G добија редослед чворова $v_1, v_2, \dots, v_n$. Ако знамо најдужи усмерени пут чији је последњи чвор v_i за $i < m \leq n$, онда се најдужи међу путевима који се завршавају у чвору v_m добија продужавањем пута до неког чвора v_i, $i < m$, граном (v_i, v_m) — ако она постоји. Дакле, приликом преласка на наредни чвор по тополошкој нумерацији, треба проверити све гране које воде у њега. Тако се после $O(|E| + |V|)$ корака добијају дужине најдужих путева до чворова v_i, $i = 1, 2, \dots, n$. Од тих дужина бира се највећа. Да би се реконструисао најдужи пут, при додавању новог чвора v_m треба памтити не само дужину најдужег пута до v_m, него и последњу грану на најдужем (или једном од најдужих) путу. Сложеност алгоритма је $O(|E| + |V|)$.

6.29. Тражени алгоритам може се конструисати прилагођавањем алгоритма за тополошко сортирање, тако што се сви чворови улазног степена 0 заједно уклањају из листе (чворова степена 0), и обједињују у прву групу. Обрада ових чворова врши се на исти начин као и у алгоритму за тополошко сортирање: уклањају се гране које из њих излазе и декрементирају се улазни степени чворова на крајевима тих грана, а затим се уклањају нови чворови улазног степена 0. Са преосталим графом наставља се даље на исти начин, па се добијају друга, трећа, …група. Није тешко доказати да произвољни чворови v, w из исте групе (са редним бројем i) не могу бити повезани путем. Заиста, у првој групи такав пут не може да постоји, јер су улазни степени свих чворова у њој једнаки нули. Ако се ради о групи $i > 1$, претпоставка да постоји пут од v до w је неодржива: из v нема грана ка "нижим" групама, а ако се из v пређе граном у чвор u у "вишој" групи, немогућ је повратак у групу i; остаје могућност да постоји пут од v до w преко чворова из групе i — која такође отпада, према конструкцији. Даље, по конструкцији у сваки чвор из групе i улази бар једна грана из неког чвора претходне групе (у противном би тај чвор морао припасти некој претходној групи). Полазећи из произвољног чвора последње групе уназад, уверавамо се да у G постоји усмерени пут дужине k.

6.30. Може се искористити претрага у ширину полазећи од чвора v. Сваком чвору w придружује се поље — бројач $w.BNP$ са текућим бројем најкраћих путева до w. За све $w \in V$ на почетку се ставља $w.BNP := 0$. Произвољна грана (x, y) на коју се наиђе у току претраге, део је најкраћег пута од v до y ако је y нови (неозначени) чвор; тада се ставља $y.BNP := x.BNP$. Ако y није нови чвор (тј. већ је означен), а x и y нису у истој генерацији (на истом растојању од корена; растојања од корена се лако успут израчунавају), онда се ставља $y.BNP := y.BNP + x.BNP$, јер се сваки најкраћи пут до x граном (x, y) продужује до најкраћег пута до y.

6.31. Довољно је не користити хип. Уместо тога, нови најкраћи пут се налази провером *свих* чворова. За то је потребно време $O(|V|)$, али је зато за обраду сваке гране довољно константно време (јер није потребна поправка хипа).

6.32. Циклус је пут који почиње и завршава у истом чвору. Алгоритам за налажење дужина свих најкраћих путева са слике 6.25 може се искористити за тражење најкраћег циклуса — пута од v_i до v_i преко чворова $v_1, v_2, \ldots, v_m$ (најкраћег m-пута), редом за $m = 0, 1, 2, \ldots, n = |V|$. Дакле, по извршењу алгоритма са слике 6.25, дужина најкраћег циклуса једнака је величини најмањег елемента на дијагонали матрице A. Да би се поред дужина најкраћих путева могли да одреде и сами најкраћи путеви, за сваки елеменат (i, j) матрице треба памтити последњи чвор на дотле најкраћем путу од v_i до v_j; на крају је онда лако реконструисати саме најкраће путеве, у овом случају најкраћи циклус.

6.33. Најпре формирамо квадратну матрицу $A = A_0$ (реда пет) дужина најкраћих 0-путева, тј. (симетричну) матрицу суседности графа G — са великим бројевима (∞) на позицијама које одговарају непостојећим гранама, и дужином гране на позицији у пресеку врсте првог и колоне другог краја гране:

$$A = A_0 = \begin{pmatrix} \infty & 10 & \infty & 1 & 5 \\ 10 & \infty & 5 & \infty & 4 \\ \infty & 5 & \infty & 3 & \infty \\ 1 & \infty & 3 & \infty & 4 \\ 5 & 4 & \infty & 4 & \infty \end{pmatrix}$$

При томе врсте, односно колоне редом $1, 2, 3, 4, 5$ одговарају чворовима редом a, b, c, d, e. Затим се редом формирају матрице A_m дужина најкраћих m-путева (путева који воде само преко чворова са индексима мањим или једнаким од m), $m = 1, 2, 3, 4, 5$:

$$A_m[i, j] = \min \{A_{m-1}[i, j], A_{m-1}[i, m] + A_{m-1}[m, j]\}, \quad 1 \leq i, j \leq 5.$$

Израчунавање матрица A_m поједностављује њихова симетрија. Редом се за матрице A_1, A_2, A_3, A_4, A_5 добија

$$\begin{pmatrix} \infty & 10 & \infty & 1 & 5 \\ 10 & 20 & 5 & 11 & 4 \\ \infty & 5 & \infty & 3 & \infty \\ 1 & 11 & 3 & 2 & 4 \\ 5 & 4 & \infty & 4 & 10 \end{pmatrix}, \quad \begin{pmatrix} 20 & 10 & 15 & 1 & 5 \\ 10 & 20 & 5 & 11 & 4 \\ 15 & 5 & 10 & 3 & 9 \\ 1 & 11 & 3 & 2 & 4 \\ 5 & 4 & 9 & 4 & 8 \end{pmatrix}, \quad \begin{pmatrix} 20 & 10 & 15 & 1 & 5 \\ 10 & 10 & 5 & 8 & 4 \\ 15 & 5 & 10 & 3 & 9 \\ 1 & 8 & 3 & 2 & 4 \\ 5 & 4 & 9 & 4 & 8 \end{pmatrix},$$

$$\begin{pmatrix} 2 & 9 & 4 & 1 & 5 \\ 9 & 10 & 5 & 8 & 4 \\ 4 & 5 & 6 & 3 & 7 \\ 1 & 8 & 3 & 2 & 4 \\ 5 & 4 & 7 & 4 & 8 \end{pmatrix}, \quad \begin{pmatrix} 2 & 9 & 4 & 1 & 5 \\ 9 & 8 & 5 & 8 & 4 \\ 4 & 5 & 6 & 3 & 7 \\ 1 & 8 & 3 & 2 & 4 \\ 5 & 4 & 7 & 4 & 8 \end{pmatrix}.$$

Последња добијена матрица A_5 је тражена матрица дужина најкраћих путева.

6.34. Нека је n број чворова у графу. Грани e задате тежине $w(e)$ може се придружити нова тежина $w'(e) = n^2 w(e) + 1$. Ако су p_1, p_2 два произвољна пута, дужина редом $|p_1|$, $|p_2|$, онда из $w(p_1) > w(p_2)$ следи да је

$$\begin{aligned} w'(p_1) &= n^2 w(p_1) + |p_1| \geq n^2(w(p_2) + 1) + |p_1| > n^2(w(p_2) + 1) + n^2 > \\ &> n^2(w(p_2) + 1) + |p_2| = w'(p_2) \end{aligned}$$

(тежина пута једнака је збиру тежина његових грана). Поред тога, ако је $w(p_1) = w(p_2)$, онда је $w'(p_1) > w'(p_2)$ акко p_1 има више грана од p_2. Према томе, довољно је применити обичан алгоритам за налажење најкраћих путева, али са промењеним тежинама w'.

6.35. Применићемо индукцију по k, и одредити не само путеве до w, него и до свих осталих чворова. Због једноставности, разматраћемо само дужине путева, а не и саме путеве. Случај $k = 1$ је јасан. Нека за $e \in E$ $d(e)$ означава дужину гране e, и нека је $f(w, k)$ дужина најкраћег пута од v до w, под условом да пут садржи тачно k грана. Претпоставимо да

знамо дужине најкраћих путева до свих чворова, под условом да се путеви састоје од по тачно $k-1$ гране, тј. да знамо $f(w,k-1)$ за све чворове w. Најпре се за свако w ставља $f(w,k)=\infty$. Затим се разматра свака грана (x,y) и вредност $f(y,k)$ замењује се вредношћу $f(x,k-1)+d(x,y)$ ако је мања. Временска сложеност алгоритма је дакле $O(|V||E|)$.

6.36. Алгоритам *Acikl_najkr_putevi2* (слика 6.18) ради коректно и ако су цене неких грана негативне. За сваки чвор се тим алгоритмом упоређују дужине свих путева до њега од v, што се може доказати индукцијом по редном броју чвора у тополошком редоследу.

6.37. Граф мора да буде Ојлеров, јер циклус повећава за два степен сваког чвора кроз који пролази. С друге стране, гране Ојлеровог циклуса увек се могу разложити у скуп циклуса. Довољно је приметити да у Ојлеровом графу постоји бар један прости циклус, и да се уклањањем тог циклуса из графа добија један или више повезаних Ојлерових графова.

6.38. Посматрајмо граф $G=(V,E)$ са n чворова и $2n-2$ гране, који се може нацртати као правилни $(n-1)$-угао са додатним чвором v у центру, страницама–гранама $e_1,e_2,\ldots,e_{n-1}$, и још $n-1$ граном f_1, f_2, $\ldots$, f_{n-1}, које повезују v са теменима $(n-1)$-угла. За произвољан непразни подскуп S скупа $\{e_1,e_2,\ldots,e_{n-1}\}$ постоји циклус у G који од грана $e_1,e_2,\ldots,e_{n-1}$ садржи тачно гране из S (добија се обједињавањем циклуса које чини v са путевима формираним од узастопних грана из S); пошто подскупова има $2^{n-1}-1$ (што је веће од $2^{n/2}$ за $n>2$), значи да и циклуса има више од $2^{0.5n}$, односно $2^{\Omega(n)}$.

6.39. Решење се може конструисати индукцијом по броју чворова. База је једноставна. Нека је v било који лист у стаблу. Волели бисмо да уклонимо v, решимо проблем за остатак стабла, и одредимо $S(v)$ тако да буду испуњени услови 1), 2). Ако је грана суседна чвору v усмерена ка v, редукција је једноставна: нека је у питању грана (w,v); тада се може ставити $S(v)=S(w)\cup\{\lambda(w,v)\}$, па се лако проверава да су услови испуњени. Тежи је случај ако је грана (v,w). Тада стављамо $S(v)=S(w)$, а свим осталим скуповима додајемо број $\lambda(v,w)$, и услови су такође испуњени.

6.40. Замислимо да су чворови у језгру обојени црно, а остали бело. Због дефиниције језгра, не постоји грана која води из црног у црни чвор, а у сваки бели чвор води бар једна грана из неког црног чвора. Чворови са улазним степеном 0 морају бити у језгру (односно обојени црно) — они не могу бити ван језгра, јер у њих не води ни једна грана. Чворови до којих воде гране из црно обојених чворова морају се обојити у бело (они не могу бити у језгру). Сада се могу уклонити бело обојени чворови, заједно са гранама које воде у или из њих, јер то не утиче на бојење преосталих чворова графа (у чвор језгра може да води грана из неког чвора ван језгра; исто важи и за чвор ван језгра). Са бојењем се наставља рекурзивно: пошто је преостали граф у општем случају шума, постоји бар један чвор улазног степена 0.

6.41. Нека је $v\in V$ произвољан фиксирани чвор; ставимо $p(v)=0$. По претпоставци, до сваког другог чвора $w\in V$ постоји пут $v=v_0,v_1,v_2,\ldots,v_k=w$; за $i=1,2,\ldots,k$ стављамо

$$p(v_i)=p(v_{i-1})+f(v_{i-1},v_i)=\sum_{j=1}^{i} f(v_{j-1},v_j).$$

На тај начин дефинисана је (бар једном) вредност $p(w)$ за свако $w\in V$. Та дефиниција је непротивречна. Заиста, ако постоји и неки други пут $v=v_0,w_1,w_2,\ldots,w_l=w$ од v до w, онда је

$$v=v_0,v_1,v_2,\ldots,v_k=w=w_l,w_{l-1},\ldots,w_1,w_0=v$$

циклус, па је по услову задатка за функцију f

$$\sum_{j=1}^{k} f(v_{j-1},v_j)+\sum_{j=1}^{l} f(v_j,v_{j-1})=0,$$

односно

$$\sum_{j=1}^{k} f(v_{j-1},v_j)=\sum_{j=1}^{l} f(v_{j-1},v_j)=p(w).$$

6.42. Нека је дат граф $G = (V, E)$ Најпре се скуп грана E сортира по растућим тежинама, што захтева $O(|E| \log|V|)$ операција. Користи се структура података за формирање унија (одељак 3.7); полази се од $|V|$ једночланих подскупова — чворова G, зачетка повезујућег стабла G. Гране се узимају једна по једна из уређеног низа. За изабрану грану проверава се у $O(\log|V|)$ корака да ли њени крајеви припадају различитим скуповима (различитим стаблима повезујуће шуме); ако не — грана се прескаче, а у противном се формира унија два подскупа и грана додаје у скуп изабраних грана ($O(1)$ корака). После додавања сваке гране број стабала у повезујућој шуми се смањује за 1; на крају се добија MCST. Заиста, претпоставимо да добијено стабло није MCST, и да је e, k-та грана у горњем низу изабраних грана, прва која не припада MCST. Тада би додавањем e у MCST настао циклус; ако са T_1, T_2 означимо два стабла која је грана e повезала, споменути циклус мора да садржи бар још једну грану e' која спаја нека два стабла повезујуће шуме која постоји у тренутку пре додавања e. По претпоставци цена e' већа је од цене e. Избацујући e' из MCST и додајући e, добијамо повезујуће стабло мање цене, супротно претпоставци.

6.43. Треба кренути од F као текућег "минималног" повезујућег стабла и додавати му гране према алгоритму из задатка 6.42.

6.44. Нека је T неко MCST у G које садржи грану $e = (v, w)$ највеће тежине у неком циклусу C. Уклањањем гране e стабло T се распада на два стабла T_1 и T_2, тако да је нпр. v у T_1, а w у T_2. Посматрајмо пут P од v до w, који се добија уклањањем e из C. На том путу постоји грана e' чији је један крај у T_1, а други крај у T_2 — која дакле повезује T_1 и T_2, а тежина јој је једнака тежини e (ако би тежина e' била мања од тежине e, онда би повезујуће стабло $(T \setminus \{e\}) \cup \{e'\}$ имало мању цену од T). Према томе, стабло $(T \setminus \{e\}) \cup \{e'\}$ је такође MCST, а не садржи грану e.

6.45. Најпре треба одредити MCST T графа G; за све гране стабла T одговор је очигледно T. За сваки пар чворова u, v треба одредити грану са најмањом ценом на (јединственом) путу од u до v у T; ово је предобрада. Ова информација није неопходна за сваки пар чворова, али је једноставније израчунати је за сваки пар. За произвољну грану (x, y) MCST $T(x, y)$ добија се додавањем гране (x, y) стаблу T, и уклањањем гране највеће цене у јединственом циклусу који је тиме формиран (или друге највеће, ако је цена гране (x, y) највећа). Та грана је грана највеће цене на путу од x до y кроз T, која се зна после предобраде. Предобрада се може извести покретањем претраге у дубину на стаблу T за сваки од његових $|V|$ чворова.

6.46. Повећање цена свих грана за исту константу не мења MCST, јер је битан само релативан однос цена, а он се не мења после повећавања свих цена за исти износ.

6.47. Основна идеја је решавати проблем уназад. Уместо да се чворови уклањају задатим редоследом, они се додају обрнутим редоследом. При томе се користи структура података за формирање унија (видети одељак 3.7), да би се знао текући скуп компоненти повезаности графа. Започињемо са чвором n и додајемо га као једну компоненту у структури података. Затим разматрамо чворове $n-1$, $n-2$, итд. За сваки чвор i, најпре га додајемо као компоненту, па онда проверавамо све гране (i, j) такве да је $j > i$. За сваку грану (i, j) проверавамо да ли гране i, j припадају истој компоненти (користи се операција проналажења скупа коме припада елеменат), а ако не, онда обједињујемо одговарајуће компоненте (операција формирања уније). Заустављамо се оног тренутка кад нека компонента садржи више од $n/2$ чворова.

6.48. Пошто сваки циклус у G мора да садржи једну грану из скупа повратних грана F, скуп грана $E \setminus F$ не може да садржи ни један циклус. Минимизација величине скупа F еквивалентна је максимизацији величине $E \setminus F$. Максимални скуп грана без циклуса у графу одговара повезујућем стаблу (или повезујућој шуми код неповезаног графа). Према томе, минимални скуп повратних грана у неусмереном графу је комплемент било ког повезујућег стабла графа, а његова величина је $|E| - |V| + 1$ (под претпоставком да је граф повезан).

6.49. Претпоставка да нема циклуса негативне тежине користи се у доказу исправности алгоритма за налажење свих најкраћих путева са слике 6.25. При преласку са најкраћих $(m-1)$-путева на најкраће m-путеве између произвољна два чвора u, v упоређује се најкраћи $(m-1)$-пут од u до v са збиром најкраћих $(m-1)$-путева од u до m и од m до v; мања од

те две вредности је дужина најкраћег m-пута u до v. При томе се користи претпоставка да се на најкраћем m-путу од u до v чвор m може наћи само једном — пролазак кроз тај чвор два пута додаје путу од u до v циклус — за који се претпоставља да има ненегативни збир тежина грана (у противном би сваки пролазак те петље смањивао дужину пута од u до v!).

6.50. Треба проверити све гране ван стабла. Нека за произвољни чвор x $x.SP$ означава дужину пута од v до x кроз стабло. Ако постоји грана (u, w) таква да је $u.SP + dužina(u, w) < w.SP$, онда T очигледно није стабло најкраћих путева. Остаје само да се докаже да ако таква грана не постоји, онда T јесте стабло најкраћих путева.

6.51. Довољно је употребити алгоритам са слике 6.25, коригован тако да прекине рад кад се наиђе на негативни дијагонални елеменат матрице.

6.52. Треба одредити тополошки редослед чворова графа и проверити да ли су узастопни чворови у том редоследу повезани (чине пут). Ако јесу, онда тополошки редослед даје Хамилтонов пут. С друге стране, ако постоји Хамилтонов пут, онда тополошко сортирање мора да доведе управо до оног редоследа чворова којим се они нижу у путу.

6.53. За граф $G = (V, E)$, $V = \{1, 2, 3, 4\}$, $E = \{(2, 3), (3, 1), (1, 4)\}$ промењени алгоритам даје на излазу граф са скупом грана без гране $(2, 4)$, која припада транзитивном затворењу графа G.

6.54. (а) Претпоставимо да v припада бази чворова B. Ако v не припада ни једном циклусу и улазни степен му је већи од нуле, онда постоји чвор w такав да је (w, v) грана и w није достижан из v. Да би скуп B био база чворова, морало би да буде $w \in B$ или да постоји пут од неког чвора из B до w. У оба случаја, међутим, постоји такође и пут од неког чвора у B до v, па се v може уклонити из B, што је у контрадикцији са минималношћу скупа B.

(б) Пошто у ацикличком графу ни један чвор не припада циклусу, из (а) следи да само чворови улазног степена 0 могу бити у бази чворова. Исто тако закључујемо да сваки чвор улазног степена 0 *мора* бити у свакој бази чворова, јер до њега не води ни један пут. Према томе, јединствена база чворова ацикличког графа је скуп чворова са улазним степеном 0 (у одељку 6.4 доказано је да је тај скуп непразан).

6.55. Посматрајмо дато стабло као коренско стабло са произвољним чвором у улози корена. Узимамо произвољан лист v и упарујемо га са својим оцем w; затим уклањамо оба чвора из стабла (заједно са осталим синовима w, који остају неупарени). Преостали проблем се по индуктивној хипотези може решити. Потребно је доказати да грана (v, w) заиста припада неком оптималном упаривању. Нека је M оптимално упаривање. Ако M не садржи грану (v, w), онда чвор v није упарен (јер је повезан једино са w). Ако је грана (u, w) у оптималном упаривању, онда се она може заменити са (v, w); тако се добија оптимално упаривање (оно садржи исти број грана као и M) које садржи грану (v, w).

6.56. Може се поћи од упаривања $M_0 = \{(a, 1), (b, 2), (c, 3), (e, 4), (f, 5), (g, 6)\}$. Да бисмо пронашли евентуалне алтернирајуће путеве у G у односу на M_0, усмеравамо гране G, и то оне које су у упаривању — удесно, а остале — улево. У добијеном усмереном графу тражимо обичан пут од неупареног чвора 7 до неупареног чвора d. Такав пут међутим не постоји: од 7 се може доћи само дао c, и даље само до 3. Према томе, M_0 је оптимално упаривање.

Ако бисмо пошли од полазног упаривања $M_1 = \{(a, 5), (b, 2), (c, 3), (e, 4), (g, 6)\}$, онда би се искоришћавањем алтернирајућег пута $f, 5, a$ дошло до истог оптималног упаривања M_0.

6.57. Може се поћи од тока са слике 10(а). Резидуални граф у односу на тај ток приказан је на слици 10(б). У њему се могу издвојити два независна (дисјунктна) увећавајућа пута: s, a, d, t капацитета $\min\{4, 5, 4\} = 4$ и s, c, f, t капацитета $\min\{6, 7, 3\} = 3$. Узимајући ово у обзир, добијамо ток са слике 10(ц). Овај ток је очигледно оптималан, јер су токови кроз гране од b, d и f ка t (које чине пресек одређен скупом $V \setminus \{t\}$) једнаки њиховим капацитетима.

6.58. (а) Граф који се може покрити звездама не сме да садржи чвор степена 0 (изоловани чвор). После уклањања произвољног чвора може се догодити да нека компонента повезаности буде изоловани чвор.

(б) Претпоставимо сада да граф нема изолованих чворова, и покушајмо да обезбедимо да и после уклањања чвора v не буде изолованих чворова. Проблеме могу да проузрокују само чворови степена 1. Због тога, ако граф садржи неки чвор v степена 1, онда v уклањамо

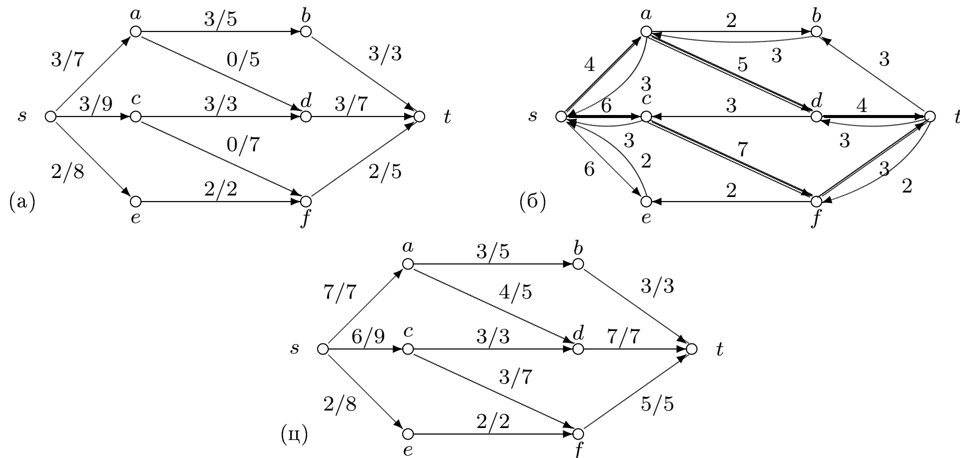


Рис. 10. Уз задатак 6.57.

заједно са граном, и звезду са кореном у v прикључујемо покривачу. На овај начин се само једном чвору степен смањује за 1; пошто је граф повезан, степен тог чвора морао је бити бар 2, па је ова редукција исправна — не доводи до појаве изолованих чворова.

6.59. Може се искористити комбинација бинарне претраге и алгоритма за налажење оптималног упаривања из одељка 6.9. Позабавимо се најпре провером да ли, за задато x, постоји оптимално упаривање такво да су тежине свих његових грана $\leq x$. Овај проблем може се решити уклањањем из графа свих грана чије су тежине веће од x, и применом алгоритма за налажење оптималног упаривања, да би се проверило да ли оптимално упаривање у смањеном графу има исти број грана као и оптимално упаривање у полазном графу. Сложеност ове провере одређена је сложеношћу алгоритма за налажење оптималног упаривања, која је $O(m\sqrt{n})$. У G има m грана, па и различитих тежина има највише m. Бинарном претрагом тражимо најмање x, такво да је x тежина неке гране, и да постоји оптимално упаривање у коме све гране имају тежину $\leq x$.

6.60. Пошто се тражи алгоритам сложености $O(n \log n)$, природно је покушати са декомпозицијом. Играчи се деле у две једнаке групе, па се за обе групе рекурзивно конструишу тражени редоследи. Затим се два редоследа обједињују на исти начин као код обичног обједињавања два сортирана низа у један. Да се ово схвати, довољно је уверити се да се произвољни играч може укључити у произвољан редослед (ланац) играча тако да се добије дужи ланац: место његовог укључивања је иза последњег играча у низу од кога је изгубио, односно испред првог играча, ако га је победио.

6.61. Јасно је да је за бојење оваквог графа неопходно бар k боја, јер се гране суседне истом чвору морају обојити различитим бојама; показаћемо да је k боја довољно. Циљ је искористити декомпозицију, тако да потпроблеми буду истог типа као и полазни проблем. Ако се граф подели на два мања подграфа, мора се обезбедити да степени чворова у оба подграфа буду међусобно једнаки и да буду неки степен двојке. Ово се може постићи конструкцијом Ојлеровог циклуса графа (он постоји јер су степени свих чворова парни). Затим се пролази циклус, а његове гране прикључују наизменично првој, односно другој групи. Свака група дефинише подграф са непромењеним скупом чворова и половином полазних грана, при чему је степен сваког чвора $k/2$ (јер кад год грану која улази у чвор сврстамо у једну групу, следећу грану, која излази из истог чвора, стављамо у другу групу). Подскупови се по индуктивној хипотези могу обојити са $k/2$ боја; ако се употреби k различитих

боја, онда се два бојења директно обједињују. Тиме је специфициран рекурзивни алгоритам за решавање овог проблема; за $k = 2$ имамо граф који се састоји од једног простог циклуса, и очигледно се може обојити са две боје. Сложеност алгоритма одређена је сложеношћу налажења Ојлерових циклуса, која је линеарна; при томе треба узети у обзир да се Ојлеров циклус мора наћи у оба подграфа. Број итерација у овом алгоритму је $\log_2 k$ (k се полови после сваке итерације); према томе, сложеност алгоритма је $O(|E| \log k)$.

6.62. Дато стабло, ако се фиксира његов произвољни чвор, може се схватити као коренско стабло. Посматрајмо произвољни лист v и његовог оца w. Грана (v, w) може се покрити чвором v, али је боље покрити је чвором w: v покрива само (v, w), а w покрива и неке друге гране. Прецизније, *постоји минимални покривач грана који садржи чвор* w. Према томе, ако изаберемо чвор w (на описани начин), уклонимо га са свим њему суседним гранама, и индукцијом решимо преостали проблем, долазимо до минималног покривача. Временска сложеност алгоритма је линеарна (пропорционална броју грана).

6.63. Јасно је да тежина покривача грана мора бити већа или једнака од броја грана (јер све гране морају бити покривене, а свака од њих повећава цену покривача најмање за један). Према томе, покривач који садржи тачно по један крај сваке гране биће минимални покривач грана. Показаћемо да такав покривач увек може бити пронађен.

Прво решење. Изврши се претрага у ширину полазећи од произвољног чвора v као корена. Затим се у покривач грана укључују сви чворови на (на пример) непарним нивоима (растојањима од корена). Добијени скуп чворова је покривач грана, јер свака грана повезује чворове са два суседна нивоа, па један од њих има непарни ниво. Поред тога, свака грана је покривена тачно једним чвором, па је покривач минималан.

Друго решење. Индуктивна хипотеза је да умемо да решимо проблем за сва стабла са $< n$ чворова. У датом стаблу са n чворова изаберимо произвољан лист v и уклонимо га (заједно са граном) из стабла. Нека је w јединствени чвор у G суседан са v, и нека је G' преостало стабло. За граф G' решавамо проблем индукцијом. Посматрајмо поново полазно стабло G. Ако је чвор w коришћен у минималном покривачу грана за G', онда имамо исправан покривач за G; у противном прикључујемо v покривачу G. Треба доказати да је у оба случаја покривач G минималан. Међутим, величина минималног покривача грана G мора бити бар за 1 већа од величине минималног покривача G', јер се додата грана мора покрити или са v или са w (а цена се повећава за 1 у оба случаја, јер се повећава степен чвора w). Та граница се достиже, па добијени покривач јесте минималан.

6.64. Нека је v произвољан чвор у G. Означимо са $N(v)$ скуп чворова суседних са v. Ако v припада покривачу грана, онда ни један чвор из $N(v)$ не може припадати покривачу (јер покривач треба да буде независан скуп). Поред тога, сви чворови суседни чворовима из $N(v)$ *морају* припадати покривачу, јер само они могу покрити гране ка чворовима из $N(v)$. Специјално, ако су било која два чвора из $N(v)$ суседна, онда се грана између њих не може покрити, па покривач грана који садржи v не постоји, тј. процедура "одустаје". Са процедуром се наставља све до неког "одустајања" (тада v не припада покривачу грана), или док се не пронађе покривач грана. Сви кораци процедуре одређени су избором чвора v, па је добијени покривач грана једини који садржи v и задовољава услове задатка. Ако добијени покривач грана има мање од k чворова, проблем је решен; у противном v не припада покривачу. Међутим, ако установимо да v не може да припада покривачу, онда покривачу морају припадати сви његови суседи, па се примењује иста процедура.

6.65. Претпоставимо да у G постоји такав подскуп $U \subset V$. Између чворова у U не сме постојати грана (U је независан скуп), а сваки чвор у $V \setminus U$ мора бити повезан са неким чвором из U (U је максимални независни скуп). Између чворова у $V \setminus U$ не сме бити грана (U је покривач грана), а сваки чвор из U мора бити повезан бар једном граном са неким чвором из $V \setminus U$ (U је минимални покривач грана). Дакле, граф G мора бити бипартитни, и то без изолованих чворова. Обрнуто, ако је $G = (U_1, E, U_2)$ бипартитни граф без изолованих чворова, онда се било U_1, било U_2 може узети за тражени скуп U. За проверу да ли је G бипартитни граф постоји полиномијални алгоритам (видети задатак 6.22), а провера да ли постоје изоловани чворови је једноставна.

6.66. Интервал I_j може се представити са два броја (l_j, d_j), својим левим и десним крајем. Уредимо интервале по растућим вредностима d_j; нека је I_1, I_2, ..., I_n тако уређени низ интервала; d_1 је најмањи од десних крајева. Тврдимо да постоји максимални независни скуп који садржи I_1. Да бисмо то доказали, узмимо произвољни максимални независни скуп и посматрајмо такав интервал I_j да је d_j најмањи од десних крајева у независном скупу. Пошто је d_1 глобални минимум, биће $d_j \geq d_1$. Из тога следи да се I_1 не сече са било којим другим интервалом у скупу (сем евентуално са I_j; ако се неки интервал I_k сече са I_j, онда због $d_k > d_j$ мора бити и $l_k > d_j \geq d_1$, па се I_k не сече ни са I_1). Према томе, I_1 може да замени I_j, а промењени скуп је и даље максимални независан скуп. Пошто се са интервалом I_1 из скупа искључе сви интервали који се секу са њим (односно лева граница им је мања од d_1), остаје мањи скуп интервала, који се даље обрађује на исти начин (без сортирања десних граница интервала — оно се извршава само једном, на почетку). Сложеност алгоритма је $O(n \log n)$.

6.67. (а) За сваки пар чворова $u, v \in V$ броји се са колико чворова су они истовремено суседни. Оног тренутка кад за неке u, v тај број достигне два, пронађен је квадрат у G, а ако је тај број за сваки пар u, v мањи или једнак од један, у G нема квадрата. Сложеност алгоритма је $O(|V|^3)$.

(б) Алгоритам се заснива на истој идеји као у (а): за сваки пар чворова $u, v \in V$ овог пута се броје заједнички елементи листа суседа чворова u и v. Дакле користи се листа повезаности за G, у којој се најпре за сваки чвор листа његових суседа сортира према растућим редним бројевима ($O(|E| \log |V|)$ корака). Бројање заједничких суседа чворова u и v може се извести за време пропорционално збиру дужина одговарајучих листа, односно $d(u) + d(v)$, па је за обраду свих парова u, v довољно време

$$\sum_{u \in V} \sum_{v \in V} (d(u) + d(v)) = 2 \sum_{u \in V} \sum_{v \in V} d(v) = 4 \sum_{u \in V} |E| = 4|E||V| = O(|E||V|).$$

6.68. Постоје фамилије графова у којима је број квадрата асимптотски већи од $O(|E||V|)$. Тако комплетан граф са n чворова има $\binom{n}{4} = O(n^4)$ квадрата, а код њега је $|E||V| = O(n^3)$. За комплетан граф већ само набрајање свих квадрата има сложеност већу од $O(|E||V|)$.

13.7. Геометријски алгоритми

7.1. Ако полуправа садржи теме P_k, треба проверити да ли су темена P_{k-1}, P_{k+1} (P_n, P_2 за $k = 1$, односно P_{n-1}, P_1 за $k = n$) са исте стране L: ако јесу, односно нису, пресек не треба, односно треба рачунати. На сличан начин решава се случај кад L садржи страницу (P_k, P_{k+1}): треба проверити да ли су темена P_{k-1} и P_{k+2} са исте стране L.

7.2. Тачке $P_1, P_2, \ldots, P_n$ су колинеарне акко су колинеарне све тројке тачака (P_1, P_2, P_i), $i = 3, 4, \ldots, n$. Сложеност алгоритма је $O(n)$.

7.3. На слици 11 су приказане четири тачке P_1, P_2, P_3, P_4, и два различита проста четвороугла $P_1P_2P_4P_3$, $P_1P_4P_2P_3$ са теменима у тим тачкама.

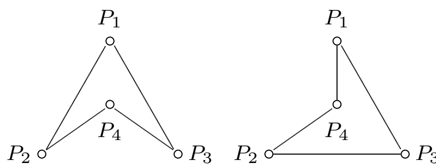


Рис. 11. Уз задатак 7.3.

7.4. Индексу $i \in \{0, 1, \ldots, n-1\}$ придружујемо величину угла $\alpha_i = \angle P_0AP_i$. Јасно је да је $-\pi < \alpha_i < \pi$; нека је $\alpha_n = 0$. Ако су сви углови α_i истог знака (односно позитивни, што се може претпоставити без смањења општости), онда је AP_0 права ослонца, а друга права

ослонца AP_m одређена је теменом P_m за које угао α_m има највећу апсолутну вредност. Теме P_m налази се бинарном претрагом интервала $1, 2, \ldots, n-1$, јер низ $|\alpha_i|$ најпре расте, па опада. У две "средње" тачке i, $i+1$ интервала индекса посматра се разлика углова $\alpha_{i+1} - \alpha_i$, па

- ако је позитивна, онда је $m \geq i+1$,
- ако је негативна, онда је $m \leq i$,
- ако је $\alpha_{i+1} = \alpha_i$, онда је нпр. $m = i$, али права ослонца P_0P_m садржи целу страницу P_mP_{m+1}.

Ако пак нису сви углови α_i истог знака, онда постоје две симетричне могућности: углови расту, опадају па расту, или опадају, расту па опадају. У оба случаја се бинарном претрагом налазе два суседна темена са угловима супротног знака. Та два темена раздвајају интервал индекса на два подинтервала, у којима се на описани начин (тражећи максимални угао по апсолутној вредности) проналази бинарном претрагом по једна права ослонца.

7.5. Нека је C центар поменутог круга. Полази се од дужи CP_1 (односно три тачке C, P_1, P_2), а онда се наставља као у обичном Грахамовом алгоритму. Кад се поново дође до тачке P_1 (пошто је конвексни пут "обухватио" све тачке $P_1, P_2, \ldots, P_n$), нека је j највећи, а i најмањи индекс неке тачке на путу у том тренутку. Могуће је да буде $\angle P_jP_1P_i > \pi$ (у противном је после избацивања тачке C завршен конвексни омотач). Тада се, слично као у Грахамовом алгоритму, искључује тачка C, и неке тачке са почетка или краја конвексног пута, све док не постану конвексни оба угла у теменима гране која повезује једно теме при крају са једним теменом са почетка пута.

7.6. У примеру на слици 12 тачке P_3, P_4,..., P_{p+3}, леже редом на дужи P_3P_{p+3}. Грахамов алгоритам ће избацити узастопце p тачака P_{p+3}, P_{p+2},..., P_4.

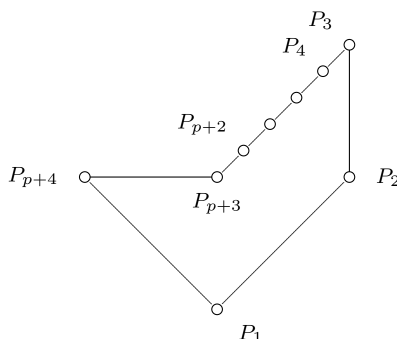


Рис. 12. Уз задатак 7.6.

7.7. Довољно је конструисати такав распоред n тачака у равни да у сваком рекурзивном позиву број тачака у средњим тракама буде пропорционалан са укупним бројем обрађиваних тачака. Такав распоред се може рекурзивно конструисати ако се пође од $n = 4 \cdot 3^k$ тачака, подељених на два једнака подскупа вертикалном симетралом, правом $x = 0$. Десни подскуп састоји се од два подскупа од по 3^{k-1} тачака на две вертикалне праве (лева од њих је $x = 1$), на вертикалним размацима по 1, и $4 \cdot 3^{k-1}$ тачака између вертикалних правих, чији се распоред конструише рекурзивно. Према томе, број тачака у средњој траци је $2 \cdot 3^{k-1} = n/6$ (овај однос важи за сваки ниво рекурзије), па је сложеност $T(n)$ решење диференцне једначине $T(n) = 2T(n/2) + cn$, односно $T(n) = \Omega(n \log n)$. Захтев да x-координате тачака буду различите лако је остварити померајући врло мало у хоризонталном правцу тачке са истом x-координатом (равномерно, увек за исти износ, пропорционално расту y-координате).

7.8. Дужи $(a_i, y_i) - (b_i, y_i)$ могу се сортирати лексикографски по паровима (y_i, a_i) (дакле по y-координатама, а у оквиру групе са истим y-координатама, у оквиру којих су једино

могући пресеци, по x-координатама левог краја дужи). Приликом проласка групе (дужи са истом y-координатом), памте се у повезаној листи кандидати — дужи чији је леви крај лево, а десни крај десно од покретне вертикалне праве; у посебном вектору се за сваку дуж памти локација левог краја у сортираном вектору. Ако је покретна права наишла на леви крај дужи, онда се региструју пресеци те дужи са свим кандидатима, и дуж се убацује у листу кандидата. Ако се наиђе на десни крај дужи, та дуж се (по налажењу позиције у листи) избацује са листе кандидата. Обрада групе од k дужи са истом y-координатом траје $O(k)$, па је сложеност укупне обраде после сортирања $O(n)$.

7.9. Нека је $y = ak + b$ једначина праве p. Права q паралелна са p има једначину $y = ak+b'$. Растојање тачке (x_k, y_k) од своје "пројекције" (у вертикалном смеру) (x_k, ax_k+b') на праву q у правцу y-осе једнако је $y_k - ax_k - b'$ (оно је позитивно, односно негативно, ако је тачка изнад, односно испод праве q), $1 \leq k \leq n$. Потребно је изабрати b' тако да половина бројева $y_k - ax_k - b'$ буде већа или једнака од нуле, односно тако да половина бројева $y_k - ax_k$ буде већа или једнака од b'. Проблем се дакле своди на налажење медијане бројева $y_k - ax_k$, $1 \leq k \leq n$.

7.10. Треба сортирати темена многоугла $p_1, p_2, \ldots, p_n$ циклички према угловима који чине дужи $q - p_i$ са неком фиксираном правом. Тим редоследом (нпр. $p_{i_1}, p_{i_2}, \ldots, p_{i_n}$) треба проћи темена. Започиње се са теменом p_{i_1}: преброје се ивице које се секу са полуправом $q - p_{i_1} -$, за шта је довољно $O(n)$ провера ивица многоугла. При преласку са темена p_{i_k} на теме $p_{i_{k+1}}$ довољно је упоредити број пресека полуправих $q - p_{i_k} -$ и $q - p_{i_{k+1}} -$ са страницама многоугла суседним са теменима p_{i_k}, $p_{i_{k+1}}$, да би се видело за колико се број пресека променио у односу на претходни положај полуправе ($O(1)$ провера). Успут се памти теме p за које је најмањи број пресека полуправе $q - p -$ са ивицама многоугла. Сложеност алгоритма је $O(n \log n)$.

7.11. Решење се заснива на примени бинарне претраге. Основна операција је провера, ако је дата права p и тачке A и B, да ли су A и B са исте стране p. Следећа елементарна операција (која се лако изводи помоћу неколико операција првог типа), је провера за четири задате тачке A, B, C и Q, да ли се Q налази унутар угла $\angle ABC$, односно унутар троугла $\triangle ABC$. Идеја на којој се заснива решење је следећа: ако знамо да се Q налази унутар угла $\angle P_i P_1 P_j$, $i < j$, онда се после провере да ли се Q и P_j налазе са исте стране праве $P_1 P_m$, $m = \lfloor (i+j)/2 \rfloor$, зна да ли је Q унутар угла $\angle P_m P_1 P_j$ (ако је одговор "да", или ако је Q на правој $P_1 P_m$), или унутар угла $\angle P_i P_1 P_m$ (у противном). На почетку се проверава да ли је Q унутар угла $\angle P_2 P_1 P_n$ (ако није, онда Q не припада многоуглу). После највише $O(\log n)$ корака закључује се или да Q не припада многоуглу, или да је унутар неког угла $\angle P_i P_1 P_{i+1}$; у другом случају је Q у многоуглу акко припада троуглу $\triangle P_i P_1 P_{i+1}$.

7.12. Прва тачка може се узети за почетак поларног координатног система; за темена текућег конвексног омотача памте се дужине потега и углови које они заклапају са x-осом, сортирано по угловима. За новододату тачку се бинарном претрагом могу пронаћи два узастопна угла–потега између којих лежи њен потег, и проверити да ли она припада конвексном омотачу; ако не, онда треба пронаћи две праве ослонца кроз ту тачку ($O(\log n)$ операција, видети задатак 7.4), искључити темена која су унутар новог конвексног омотача, и укључити ново теме ($O(\log n)$ операција). Сложеност алгоритма је дакле $O(n \log n)$.

7.13. Помоћу алгоритма из задатка 7.9 проналази се права која дати скуп тачака дели на два подскупа, тако да се бројеви тачака у подскуповима разликују највише за један. Лако се може постићи да на деобној правој не лежи ни једна тачка из скупа. Алгоритам који конструишемо користимо рекурзивно за добијање конвексних омотача P_1 и P_2 ова два подскупа (они имају мање од n тачака); црном кутијом се конвексни омотачи P_1 и P_2 обједињују у конвексни омотач свих n тачака. Сложеност алгоритма $T(n)$ (по претпоставци о црној кутији) задовољава диференцну једначину $T(n) = 2T(n/2) + cn$, па је $T(n) = O(n \log n)$.

7.14. Раван се може изделити у вертикалне траке ширине d, почевши од тачке p са минималном x-координатом, до тачке q са максималном x-координатом; број трака је $O(X/d)$. Дате тачке се разврставају по тракама; да се за једну тачку установи којој траци припада

довољно је $O(1)$ операција. За сваку траку проналазе се тачке са са минималном и максималном y-координатом. Слично као у Грахамовом алгоритму повезују се редом све "горње" и све "доње" тачке (из омотача се избацују оне које са два суседа чине угао већи од π; $O(n)$ корака). Тако се добијају два конвексна пута, "горњи" и "доњи", који повезују тачке p и q (ако је нпр. доњи пут цео изнад дужи $p-q$, он се избацује и замењује се дужи $p-q$). Ова два конвексна пута заједно чине d-приближни омотач скупа P, што се лако доказује.

7.15. (а) Тачке скупа сортирају се по x-координатама. Тачка (x_1, y_1) са највећом x-координатом је очигледно максимална. Остале тачке којима је y-координата мања од y_1 нису максималне, јер над њима доминира тачка (x_1, y_1). Дакле, тачке се разматрају редом према опадајућим x-координатама и елиминишу до наиласка на прву тачку (x_2, y_2) за коју је $y_2 > y_1$ — која је максимална. Поступак се наставља даље на исти начин, елиминишући тачке са y-координатама мањим од y-координате последње пронађене максималне тачке. Део алгоритма после сортирања има сложеност $O(n)$.

(б) Слично као у (а) тачке се сортирају по опадајућим x-координатама; за обрађене тачке чувају се сортирано пројекције пронађених максималних тачака у раван yOz. Нова тачка се елиминише ако над њеном пројекцијом доминира нека пројекција (провера се врши бинарном претрагом); у противном, нова тачка елиминише неке од пројекција после уметања своје пројекције.

7.16. Најпре се тачке сортирају према растућим x-координатама. За тачку са најмањом x-координатом је број тачака "испод ње" (тачака над којима она доминира) нула. Долазећи до наредне тачке, умећемо је у уравнотежено бинарно стабло претраге са кључевима једнаким y-координатама тачака. Пошто нова тачка има већу x-координату од свих претходних тачака, она доминира над свим тачкама у стаблу којима је y-координата мања од њене. Уметање тачке у стабло је сложености $O(\log n)$, па је укупан број операција $O(n \log n)$. Да би се лако рачунао број тачака испод новододате тачке, уз сваки чвор v у стаблу чува се информација $v.B$ о броју чворова у његовом левом подстаблу. При уметању новог чвора у стабло, најпре стављамо $s := 0$, а онда у сваком чвору w кроз који прођемо, ако се иде улево инкрементирамо вредност $w.B$, а ако се иде удесно, суму s повећавамо за $w.B + 1$. На тај начин, кад пронађемо место за уметање новог елемента, израчунат је број s тачака испод њега. Приликом уравнотежавања стабла ротацијом или двоструком ротацијом, мењају се вредности поља B за време $O(1)$, што се непосредно проверава.

7.17. Тачке се најпре сортирају по својим x-координатама. Затим се међу нагибима дужи које повезују *узастопне* тачке проналази највећи, и то је решење проблема. Коректност алгоритма последица је чињенице да ако између тачака A_0 и A_k у сортираном редоследу постоје још неке тачке A_1, A_2,..., A_{k-1}, онда постоје такве две узастопне тачке A_i и A_{i+1} да је нагиб дужи A_iA_{i+1} већи или једнак од нагиба дужи A_0A_k. Ово тврђење може се доказати индукцијом по $k \geq 1$. Базни случај $k = 1$ је тривијалан. Нека је тврђење тачно за неко k. Посматрајмо произвољних узастопних $k + 2$ тачака A_0, A_1,..., A_k, A_{k+1}. Ако је тачка A_1 изнад дужи A_0A_{k+1}, или лежи на њој, онда је нагиб дужи A_0A_1 већи или једнак од нагиба A_0A_{k+1}; у противном, нагиб A_1A_{k+1} је већи од нагиба A_0A_{k+1}, па по индуктивној хипотези између A_1 и A_{k+1} постоје две узастопне тачке A_i, A_{i+1} такве да је нагиб A_iA_{i+1} већи или једнак од нагиба A_1A_{k+1}, а тиме и већи од нагиба A_0A_{k+1}.

7.18. У свакој листи тражи се најмањи модуо разлике узастопних чланова. Кад се за парове суседних листа тражи најближи пар тачака, довољно је рачунати растојање сваке тачке из једне листе од само две тачке (једне испод, једне изнад) из суседне листе; парови тачака за које се тражи растојање добијају се обједињавањем (које има линеарну сложеност) двеју суседних листа према y-координатама. Укупан број парова тачака за које се тражи растојање (или квадрат растојања, да би се избегло рачунање квадратног корена) је дакле $O(n)$. Алгоритам не гарантује налажење пара тачака на најмањем растојању, јер две најближе тачке не морају припадати суседним листама или једној листи (нпр. тачке $(0, 5)$, $(1, 0)$ и $(2, 5)$: алгоритам не би нашао пар $(0, 5) - (2, 5)$ на растојању 2).

7.19. Једини тежак случај је налажење пресека свих вертикалних дужи са дужима под нагибом $\pi/4$ (сви остали случајеви своде се на овај, или се решавају алгоритмом из

одељка 7.6). Користи се исти приступ као у алгоритму *Preseci* из одељка 7.6. Сортирају се x-координате свих крајева дужи. Алгоритам са покретном вертикалном правом примењује се на исти начин. Дуж под нагибом $\pi/4$ убацује се у списак кандидата кад покретна права дође до њеног левог краја, и уклања се из њега кад покретна права дође до њеног десног краја. Потребно је још одредити пресеке између нове вертикалне дужи и неколико кандидата, дужи под нагибом $\pi/4$. То се може обавити на следећи начин. За сваку косу дуж рачуна се x-координата пресека њене праве са x-осом. Ове вредности користе се при претрази једнодимензионалног опсега, чији су крајеви x-координате пресека са x-осом правих под углом $\pi/4$ (у односу на x-осу) кроз крајеве вертикалне дужи. Сложеност комплетног алгоритма због тога остаје непромењена.

7.20. Задатак се може решити слично као у специјалном случају кад су дужи хоризонталне или вертикалне, методом покретне праве. Због једноставности се може претпоставити да међу дужима нема паралелних, нити делимично преклопљених, нити вертикалних дужи. Најпре се сортирају сви крајеви дужи према x-координатама. У сортираном редоследу се за сваку тачку поред њених координата памти и редни број дужи којој она припада, и да ли је она леви или десни крај те дужи. Тиме се обезбеђује познавање редоследа којим покретна права наилази на крајеве дужи. За дужи $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ које покретна права L сече у неком тренутку ("кандидате") претпоставља се да y-координате њихових пресека са L чине растући низ, као и да су x-координате пресека правих — носача парова суседних дужи редом v_{i_1}, v_{i_2}; $v_{i_2}, v_{i_3}; \ldots; v_{i_{k-1}}, v_{i_k}$ смештене у посебан хип H. На врху хипа налази се x-координата првог следећег пресека нека два суседна кандидата. У тренутку кад права L наиђе на нови крај дужи са x-координатом x_i, онда се најпре проверава да ли је најмањи елеменат H мањи од x_i (ако јесте, онда је покретна права у међувремену прешла преко пресека нека два суседна кандидата, и проблем је решен). Даље, ако је у питању

- леви крај дужи v_p — нови кандидат: проналази се бинарном претрагом место y-координате левог краја v_p у сортираном редоследу (према y-координатама пресека са L — тај редослед се није променио ако се неке две суседне дужи нису пресекле до доласка L на x_i) y-координата пресека кандидата са L; дуж се умеће у сортирани редослед кандидата (за ту сврху може се користити АВЛ стабло) између неке две дужи v_{i_m}, $v_{i_{m+1}}$; x-координата пресека носача v_{i_m}, $v_{i_{m+1}}$ избацује се из H, а у H се убацују x-координате пресека носача v_{i_m}, v_{i_p}, односно v_{i_p}, $v_{i_{m+1}}$ (уз одговарајуће преуређење хипа H).
- десни крај дужи v_{i_m}: из H се избацују x-координате пресека носача $v_{i_{m-1}}, v_{i_m}$, односно $v_{i_m}, v_{i_{m+1}}$, па се у H убацује x-координата пресека носача $v_{i_{m-1}}, v_{i_{m+1}}$; при томе се v_{i_m} избацује из АВЛ стабла са кандидатима.

Поправке АВЛ стабла и хипа H су сложености $O(\log n)$ у најгорем случају, укупан број обрађених крајева дужи је $2n$, па је сложеност алгоритма $O(n \log n)$.

7.21. Најпре се сортирају сви крајеви дужи, при чему се за сваку тачку памти податак да ли је леви или десни крај, црвене или плаве дужи. Вредности бројача p и c поставе се на нулу. Крајеви дужи се пролазе слева удесно овим сортираним редоследом, и при наиласку на сваку нову тачку, ако је то

- почетак плаве (црвене) дужи, инкрементира се број p (c) "отворених" плавих (црвених) дужи,
- крај плаве (црвене) дужи, декрементира се број p (c) "отворених" плавих (црвених) дужи.

Проверава се да ли је истовремено $p > 0$ и $c > 0$; ако јесте, пронађен је пресек плаве и црвене дужи. Ако се у току прегледа крајева дужи не догоди истовремено $p > 0$ и $c > 0$, онда не постоји пресек неке плаве и неке црвене дужи.

7.22. Најпре се интервали сортирају према растућим x-координатама својих левих крајева. Проблем се може решити помоћу следеће индуктивне хипотезе: у скупу са $< n$ интервала умемо да означимо све интервале садржане у другим интервалима, и да одредимо највећи десни крај тих интервала. Ако је x-координата десног краја n-тог интервала већа

од до тог тренутка највеће x-координате десног краја неког интервала, онда n-ти интервал није садржан у неком другом интервалу; његов десни крај преузима улогу највећег десног краја. У противном је он део неког другог интервала, па се означава. Последњи интервал има највећу x-координату левог краја, па не садржи ни један други интервал. Сложеност алгоритма је $O(n \log n)$, због почетног сортирања.

7.23. Сортирају се ДЛ (доњи леви) и ГД (горњи десни) углови датих правоугаоника према x-координатама. Користи се следећа структура података S: за сваки ДЛ угао памти се низ ГД углова који доминирају над њим (видети задатак 7.15), и то сортирано, нпр. у АВЛ стаблу. Кад се наиђе на нови ДЛ угао, проверава се да ли је његов ГД угао испод неког другог у S; ако јесте, означи се, а ако није, онда се убацује у S, при чему се неке тачке избацују из S (користи се бинарна претрага, а укупан број избацивања у току извршења целог алгоритма мањи је од n). Кад се наиђе на ГД угао, одговарајући првоугаоник се избацује из S. Укупна сложеност је $O(n \log n + n) = O(n \log n)$.

7.24. Пресек два правоугаоника са страницама паралелним осама проналази се за константно време. Тај пресек је или празан, или је неки правоугаоник. Према томе, проблем се може решити за време $O(n)$, налазећи пресеке једног по једног правоугаоника са пресеком свих претходних правоугаоника.

7.25. За сваку кружницу посматрамо вертикалне праве које је додирују. Све x-координате ових правих се сортирају, па се праве пролазе слева удесно покретном вертикалном правом L. Између два узастопна положаја L у траци између њих постоји низ лукова који повезују једну тачку леве са једном тачком десне праве. Свака кружница–кандидат (она која сече обе праве) даје два оваква лука. Задатак је проверити да ли се нека два од ових лукова секу у траци; ако нема пресека, онда се лукови могу сортирати према растућим y-координатама пресека са левом правом. Дакле, за сваког кандидата треба чувати два лука (они су задати центром, полупречником и податком да ли је у питању горња или доња полукружница), и то сортирано према y-координатама пресека са L. Тај редослед се не мења при преласку L у нови положај ако нема пресека; ако се пак нека два лука секу, онда увек постоји пресек нека два узастопна лука у сортираном редоследу. Према томе, ако L наиђе на десни крај неког кандидата, његова оба лука се елиминишу (погодна структура података је зато уравнотежено бинарно стабло претраге); ако наиђе на леви крај неке кружнице, онда у сортирани редослед треба уметнути оба лука и проверити да ли се нова кружница сече са кружницама–суседима у сортираном редоследу кандидата (две кружнице полупречника r_1 и r_2 са растојањем центара d секу се акко је $|r_1 - r_2| \leq d \leq r_1 + r_2$). Сложеност алгоритма је $O(n \log n)$.

7.26. Проблем се може решити слично као у задацима 7.25, 7.20. У овом случају се сортира свих nk темена многоуглова по x-координатама. Текући скуп кандидата чине ивице многоуглова које секу покретну вертикалну праву; оне се могу (ако нема пресека) чувати сортирано према y-координатама својих пресека са покретном правом. За кандидате се памте индекси многоуглова којима припадају, да се не би као пресеци рачунала темена многоуглова — пресеци суседних ивица. Укупан број кандидата је највише nk, па је сложеност алгоритма $O(nk \log(nk))$.

7.27. Идеја је изделити оба многоугла вертикалним правама на трапезе, и онда пронаћи пресеке парова трапеза. Најпре се сва темена сортирају према x-координатама. Ово се може урадити за време $O(n)$, јер се зна циклички редослед темена оба многоугла. Кроз свако теме повуче се вертикална права и на тај начин су оба многоугла издељена на трапезе (са основама паралелним y-оси; троугао је специјалан случај трапеза). У једној по једној вертикалној траци, слева удесно, проналазе се (за време $O(1)$, јер оба трапеза имају по највише четири темена) пресеци по два трапеза, дела два многоугла. Затим се за време $O(n)$ одговарајући пресеци обједињују у многоугао.

7.28. Пресек троуглова је конвексан многоугао. Пресек два троугла проналази се за константно време, а пресек два конвексна многоугла за линеарно време (видети задатак 7.27).

Према томе, алгоритам заснован на декомпозицији дели троуглове на два подскупа, израчунава рекурзивно пресеке троуглова у оба подскупа, а онда проналази пресек два добијена конвексна многоугла. Сложеност алгоритма је $O(n \log n)$.

7.29. Нека горња граница d за дијаметар многоугла (дужину највеће дужи коју садржи) може се одредити за $O(n)$ корака — то је нпр. дијагонала правоугаоника који садржи многоугао, а ивице су му паралелне осама. Затим се за свако теме P_i многоугла конструише једнакокраки троугао коме је P_i теме, странице му садрже странице многоугла које се сустичу у темену P_i, а висина из темена P_i има дужину d (тај троугао садржи цели многоугао!). Пресек ових n троуглова је задати многоугао.

7.30. За сваку тачку P_i, $1 \leq i \leq n$, сортирају се нагиби од P_i ка осталим тачкама, и провери се да ли међу њима има једнаких (ако има, онда су пронађене три колинеарне тачке). Сложеност алгоритма је $O(n \cdot n \log n) = O(n^2 \log n)$.

7.31. Нека су дате тачке $A_i = (x_i, y_i)$, $i = 1, 2, \ldots, n$. Свих $n(n-1)/2$ дужи A_iA_j могу се лексикографски сортирати ($O(n^2 \log n)$ корака) према четворки (x_i, y_i, x_j, y_j); при томе је због одређености редослед тачака у сваком пару A_iA_j дефинисан тако да буде $x_i < x_j$ или $x_i = x_j$ и $y_i < y_j$. На тај начин могуће је за произвољну дуж бинарном претрагом за $O(\log n)$ корака проверити да ли се налази или не у овом скупу дужи. За сваку од $n(n-1)/2$ дужи се проверава да ли скуп садржи три дужи које са њом затварају квадрат на један или други начин — за ово је потребно највише шест бинарних претрага, односно још укупно $O(n^2 \log n)$ корака.

7.32. Нека су дате тачке $P_i = (x_i, y_i)$, $i = 1, 2, \ldots, n$. Формирају се четири скупа бројева $A = \{x_i | 1 \leq i \leq n\}$, $B = \{y_i | 1 \leq i \leq n\}$, $C = \{x_i - y_i | 1 \leq i \leq n\}$ и $D = \{x_i + y_i | 1 \leq i \leq n\}$. Сортирањем бројева у овим скуповима утврђује се број различитих елемената у сваком од њих, који је управо једнак броју паралелних правих које садрже све дате тачке, а паралелне су y-оси (A), x-оси (B), правој $y = x$ (C) и правој $y = -x$ (D). Најмањи од ова четири броја одређује решење задатка.

13.8. Алгебарски алгоритми

8.1. Нека је $(k_{m-1}k_{m-2}\ldots k_0)_2$ бинарни запис броја k. Резултату z додељује се почетна вредност $z := 1$. Пошто је $k = ((k_{m-1} \cdot 2 + k_{m-2}) \cdot 2+) \ldots \cdot 2 + k_0$, бити k_i пролазе се редом за $i = m-1, m-2, \ldots, 0$: текући резултат z се квадрира, а ако је $k_i = 1$, онда се и множи са n: $z := z^2 \cdot n^{k_i}$.

8.2. Алгоритам *Stepen_kvadriranjem* са слике 8.2 за степеновање експонентом 15 захтева 6 множења, $n^{15} = (((n^2 \cdot n)^2 \cdot n)^2 \cdot n)$. Међутим, n^{15} може се израчунати помоћу 5 множења: $n^{15} = ((n^2)^2 \cdot n)^3$.

8.3. Потребно је Еуклидовим алгоритмом одредити $d = \mathrm{NZD}\,(a, b)$, после чега је $a' = a/d$, $b' = b/d$.

8.4. Доказује се индукцијом по k, где је $n = 2^k$.

8.5. За разлику од множења полинома, ако се покуша са свођењем производа PQ, $0 \leq P, Q < 2^n$, на производе мањих бројева помоћу $P = P_0 + 2^{n/2}P_1$, $Q = Q_0 + 2^{n/2}Q_1$, $0 \leq P_0, P_1, Q_0, Q_1 < 2^{n/2}$, $PQ = P_0Q_0 + 2^{n/2}(P_0Q_1 + P_1Q_0) + 2^nP_1Q_1$, и $P_0Q_1 + P_1Q_0 = (P_0 + P_1)(Q_0 + Q_1) - P_0Q_0 + P_1Q_1$, појављује се израчунавање производа $(P_0 + P_1)(Q_0 + Q_1)$ бројева $0 \leq P_0 + P_1, Q_0 + Q_1 \leq 2 \cdot 2^{n/2} - 2$, који излазе из полазног опсега $[0, 2^n - 1]$. Проблем се може решити на следећи начин: израчунавање производа два броја од по $n = 2^k + 2$ бита своди се на израчунавање три производа бројева од по највише $m = \frac{n}{2} + 1 = 2^{k-1} + 2$ бита. Заиста, ако је $0 \leq P, Q < 2^n$, онда је $P = P_0 + 2^{m-1}P_1$, $Q = Q_0 + 2^{m-1}Q_1$, где је $0 \leq P_0, P_1, Q_0, Q_1 < 2^{m-1}$, и $0 \leq P_0 + P_1, Q_0 + Q_1 < 2^m$. Сложеност $T(n)$ задовољава диференцну једначину $T(2^k + 2) = 3T(2^{k-1} + 2) + c2^k$. Индукцијом се показује да је горња граница њеног решења $T(2^k + 2) < 3c(3^k - 2^k)$, па за $n = 2^k + 2$ важи $T(n) < 3c3^k = 3c2^{k \log_2 3} = 3c(n-2)^{\log_2 3}$, односно $T(n) = O(n^{\log_2 3})$. Уопштење на случај произвољне основе $b \geq 2$ је директно — и даље се производ бројева од $n = 2^k + 2$ цифара своди на три производа бројева од по $m = n/2 + 1$ цифара.

Друго решење. Ако је $n = 2^k$, може се поћи од $P = P_0 + 2^{n/2}P_1$, $Q = Q_0 + 2^{n/2}Q_1$, $0 \leq P_0, P_1, Q_0, Q_1 < 2^{n/2}$, и идентитета

$$PQ = (2^n + 2^{n/2}) + 2^{n/2}(P_1 - P_0)(Q_0 - Q_1) + (2^{n/2} + 1)P_0Q_0,$$

којим се израчунавање производа n-битних бројева P, Q своди на израчунавање три производа $n/2$-битних бројева и неколико једноставних операција, сабирања и дописивања нула здесна у систему са основом 2. Уопштење на случај произвољне базе $b \geq 2$ такође је директно.

8.6. Довољно је израчунати производе $p_1 = ac$, $p_2 = bd$ и $p_3 = (a+b)(c+d) = p_1 + p_2 + (ad + bc)$, јер је тада

$$(a + bi)(c + di) = ac - bd + i(ad + bc) = p_1 - p_2 + i(p_3 - p_1 - p_2).$$

8.7. Ако се ограничимо на случај $n = 4^k$, и ако са $T(n)$ означимо сложеност израчунавања производа две матрице реда n, онда је $T(n) = kT(n/4) + cn^2$. Решење ове диференцне једначине је (видети одељак 2.5.4) $T(n) = O(n^{\log_4 k})$. Да би алгоритам био асимптотски бржи од Штрасеновог, требало би да буде $n^{\log_4 k} < n^{\log_2 7}$, или $\log_4 k < \log_2 7 = \log_4 49$, односно $k < 49$.

8.8. Узастопна множења: n^i има id цифара, па је укупан број корака $d^2 + 2d^2 + \ldots + (k-1)d^2 = (k^2 - k)d^2/2$.

Узастопна степеновања: укупан број корака је

$$d^2 + (2d)^2 + (4d)^2 + \ldots + (kd/2)^2 = (k^2 - 1)d^2/3.$$

Дакле, под овим претпоставкама оба алгоритма имају исту асимптотску сложеност $O(k^2d^2) = O(k^2 \log^2 n)$.

8.9. НЗД бројева $a_1, a_2, \ldots, a_k$ се не мења ако се (претпоставимо да је a_1 најмањи од њих) бројеви $a_2, a_3, \ldots, a_k$ замене својим остацима при дељењу са a_1, и ако се затим из скупа избаце нуле. Овим операцијама се величина највећег броја у скупу смањује, па се (као и код обичног Еуклидовог алгоритма) после коначног броја корака долази до система од само једног броја; тај број је тражени НЗД бројева $a_1, a_2, \ldots, a_k$.

8.10. Нека је $d = \mathrm{NZD}\,(a, b)$; тада је $a = a_1d$, $b = b_1d$, где су a_1 и b_1 узајамно прости бројеви, па је $\mathrm{NZS}\,(a, b) = a_1b_1d = ab/\mathrm{NZD}\,(a, b)$. Тиме је одређивање $\mathrm{NZS}\,(a, b)$ сведено на одређивање $\mathrm{NZD}\,(a, b)$.

8.11. Проблем се може решити индукцијом по k. Ако су дати бројеви $a_1, a_2, \ldots, a_k$ и израчунат је $A_{k-1} = \mathrm{NZS}\,(a_1, a_2, \ldots, a_{k-1})$, онда је

$$A_k = \mathrm{NZS}\,(a_1, a_2, \ldots, a_k) = \mathrm{NZS}\,(A_{k-1}, a_k) = A_{k-1}a_k/\mathrm{NZD}\,(A_{k-1}, a_k),$$

односно потребно је још једном извршити Еуклидов алгоритам, множење и дељење.

8.12. Да се сваки природан број n може представити у облику збира различитих Фибоначијевих бројева може се доказати индукцијом по n. За $n = 1, 2$ тврђење је тачно. Нека је тврђење тачно за бројеве мање од n (индуктивна хипотеза). Нека је k такав индекс да је $F(k) \leq n < F(k+1)$. Тада је $F(k) > n/2$ (у противном би било $F(k+1) = F(k) + F(k-1) \leq 2 \cdot n/2 = n$). Према индуктивној хипотези се $n - F(k)$ може представити у облику збира највише $\log_2(n - F(k))$ различитих Фибоначијевих бројева. Међутим, како је $n - F(k) < F(k)$, у тој репрезентацији $F(k)$ не може да учествује, па је и за број $n = (n - F(k)) + F(k)$ добијена репрезентација у облику збира различитих Фибоначијевих бројева; број сабирака је мањи од $\log_2(n - F(k)) + 1 = \log_2(n + (n - 2F(k))) < \log_2 n$. Да бисмо ефективно одредили репрезентацију датог броја n, израчунавамо Фибоначијеве бројеве док не дођемо до неког који је већи од n, а онда настављамо као у индуктивном доказу: од n се одузима претходни Фибоначијев број, па се са разликом наставља на исти начин, користећи већ израчунате Фибоначијеве бројеве.

8.13. Слично као код природних бројева, ако се изврши дељење са остатком $a = bq + r$, $\deg r < \deg b$, онда је $\mathrm{NZD}\,(a, b) = \mathrm{NZD}\,(b, r)$. Тако се може полазећи од $r_0 = a$ и $r_1 = b$ извршити низ дељења са остатком $r_{i-1} = qr_i + r_{i+1}$, $\deg r_{i+1} < \deg r_i$, $i = 1, 2, \ldots, k$ (низ дељења је коначан, јер степени полинома r_i чине строго опадајући низ). Ако је r_k последњи

остатак различит од нуле, онда је $r_k = \mathrm{NZD}\,(a, b)$, јер сваки заједнички делилац a и b дели r_k, и сваки делилац r_k дели и a и b. НЗД се може изразити у облику $r_k = au + bv$, где су u и v полиноми; то се постиже на исти начин као и код природних бројева. Ако су два полинома d_1 и d_2 истовремено НЗД полинома a и b, онда $d_1|d_2$ и $d_2|d_1$, па пошто су им најстарији коефицијенти јединице, мора бити $d_1 = d_2$.

8.14. Нека је $p_1 = a_1a_2$, $p_2 = b_1b_2$, $p_3 = c_1c_2$, $p_4 = d_1d_2$, $p_{12} = (a_1 + b_1)(a_2 + b_2)$, $p_{13} = (a_1+c_1)(a_2+c_2)$, $p_{14} = (a_1+d_1)(a_2+d_2)$, $p_{23} = (b_1-c_1)(b_2+c_2)$, $p_{24} = (d_1-b_1)(b_2+d_2)$, $p_{34} = (c_1 - d_1)(c_2 + d_2)$ (укупно 10 множења). Тада је

$$\begin{aligned}
& (a_1 + b_1i + c_1j + d_1k)(a_2 + b_2i + c_2j + d_2k) = \\
& (a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2) \\
+\quad & (a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2)i \\
+\quad & (a_1c_2 - b_1d_2 + c_1a_2 + d_1b_2)j \\
+\quad & (a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2)k \\
=\quad & (p_1 + p_2 + p_3 + p_4) + (p_{12} - p_1 - p_2 + p_{34} - p_3 + p_4)i \\
+\quad & (p_{13} - p_1 - p_3 + p_{24} - p_4 + p_2)j + (p_{14} - p_1 - p_4 + p_{23} - p_2 + p_3)k.
\end{aligned}$$

Према томе, израчунавање производа два кватерниона своди се на 10 (уместо 16) реалних множења.

8.15. Пошто је

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} a^2 + bc & b(a+d) \\ c(a+d) & bc + d^2 \end{pmatrix},$$

види се да је довољно израчунати пет производа a^2, bc, d^2, $c(a + d)$ и $b(a + d)$.

8.16. (а) Матрица A је пермутациона акко се резултат множења $y = Ax$ произвољног вектора колоне x са A добија пермутовањем компоненти x (на начин који не зависи од x: $y[i] = x[j] = x[P[i]]$, ако је $P[i] = j$, $1 \le i \le n$). Производ две пермутационе матрице A_1A_2 на произвољан вектор колону x делује тако што му пермутује компоненте, јер је $(A_1A_2)x = A_1(A_2x)$, па је и A_1A_2 пермутациона матрица.

(б) Нека је $y = A_2x$, $z = A_1y = A_1A_2x$, и нека су P_1, P_2 вектори који представљају пермутационе матрице редом A_1, A_2. Тада је $y[i] = x[P_2[i]]$, $z[i] = y[P_1[i]] = x[P_2[P_1[i]]]$, па је i-та компонента вектора који представља пермутациону матрицу A_1A_2 једнака $P_2[P_1[i]]$, $1 \le i \le n$.

8.17. Може се применити декомпозиција. Булова функција $f(x_1, x_2, \ldots, x_n)$ од n променљивих може се представити као линеарни полином по x_n:

$$\begin{aligned}
& x_nf(x_1, x_2, \ldots, 1) \vee \bar{x}_nf(x_1, x_2, \ldots, 0) \\
=\quad & x_nf(x_1, x_2, \ldots, 1) + (1 + x_n)f(x_1, x_2, \ldots, 0) \\
=\quad & x_n(f(x_1, x_2, \ldots, 1) + f(x_1, x_2, \ldots, x_{n-1}, 0)) + f(x_1, x_2, \ldots, x_{n-1}, 0) \\
=\quad & x_nh(x_1, x_2, \ldots, x_{n-1}) + g(x_1, x_2, \ldots, x_{n-1}),
\end{aligned}$$

где је

$$\begin{aligned}
g(x_1, x_2, \ldots, x_{n-1}) &= f(x_1, x_2, \ldots, x_{n-1}, 0), \\
h(x_1, x_2, \ldots, x_{n-1}) &= f(x_1, x_2, \ldots, x_{n-1}, 1) + f(x_1, x_2, \ldots, x_{n-1}, 0);
\end{aligned}$$

коришћен је идентитет $\bar{x}_n = 1 + x_n$. Вектор од 2^n коефицијената полинома f добија се према томе надовезивањем два вектора од по 2^{n-1} коефицијената полинома g, h од $n - 1$ променљиве. Ако број операција потребних за добијање коефицијената полинома функције од n променљивих означимо са $T(n)$, онда је $T(n) = 2T(n - 1) + c2^n$. Члан $c2^n$ потиче од израчунавања таблице h сабирањем две половине таблице f. Решење ове диференцне једначине је $T(n) = O(n2^n)$ (добија се нпр. увођењем смене $R(n) = 2^{-n}T(n)$, која диференцну једначину трансформише у $R(n) = R(n - 1) + c$, одакле је $R(n) = O(n)$).

8.18. Задатак се решава налажењем растојања f од свих линеарних функција $L_a(x) = a_0+\sum_{i=1}^{n} a_i x_i$, $a = (a_0, a_1, \ldots, a_n) \in B^{n+1}$. Сложеност тривијалног алгоритма (израчунавање свих 2^n вредности сваке од 2^{n+1} линеарних функција, и налажење растојања тих функција од f) је $O(n2^n \cdot 2^n) = O(n4^n)$. До уштеде се може доћи применом декомпозиције

$$f = x_n f''(x_1, x_2, \ldots, x_{n-1}) + \bar{x}_n f'(x_1, x_2, \ldots, x_{n-1}),$$

где су

$$\begin{aligned} f'(x_1, x_2, \ldots, x_{n-1}) &= f(x_1, x_2, \ldots, x_{n-1}, 0), \\ f''(x_1, x_2, \ldots, x_{n-1}) &= f(x_1, x_2, \ldots, x_{n-1}, 1) \end{aligned}$$

две Булове функције од $n-1$ промењљиве (таблица f добија се надовезивањем таблица f', f''). На исти начин се произвољна линеарна функција од n промењљивих $L_a(x_1, x_2, \ldots, x_n) = a_0 + \sum_{i=1}^{n} a_i x_i$ представља преко две линеарне функције од $n-1$ промењљиве

$$\begin{aligned} L'_{a_0,a_1,\ldots,a_{n-1}}(x_1, x_2, \ldots, x_{n-1}) &= a_0 + \textstyle\sum_{i=1}^{n-1} a_i x_i, \\ L''_{a_0,a_1,\ldots,a_{n-1}}(x_1, x_2, \ldots, x_{n-1}) &= a_0 + a_n + \textstyle\sum_{i=1}^{n-1} a_i x_i : \end{aligned}$$

$$\begin{aligned} & L_a(x_1, x_2, \ldots, x_n) \\ = \; & x_n L''_{a_0,a_1,\ldots,a_{n-1}}(x_1, x_2, \ldots, x_{n-1}) + \bar{x}_n L'_{a_0,a_1,\ldots,a_{n-1}}(x_1, x_2, \ldots, x_{n-1}). \end{aligned}$$

При томе је

$$L''_{a_0,a_1,\ldots,a_{n-1}} = a_n + L'_{a_0,a_1,\ldots,a_{n-1}} = \begin{cases} \bar{L}'_{a_0,a_1,\ldots,a_{n-1}}, & a_n = 1 \\ L'_{a_0,a_1,\ldots,a_{n-1}}, & a_n = 0 \end{cases},$$

што значи да је друга половина таблице функције $L_{a_0,a_1,\ldots,a_n}$ једнака првој ако је $a_n = 0$, односно њеном комплементу ако је $a_n = 1$. Свакој линеарној функцији од $n-1$ промењљиве одговарају на овај начин две линеарне функције од n промењљивих (за $a_n = 0$, односно $a_n = 1$). Полазећи од једнакости

$$\begin{aligned} & D(f, L_{a_0,a_1,\ldots,a_n}) = D(f', L'_{a_0,a_1,\ldots,a_{n-1}}) + D(f'', L''_{a_0,a_1,\ldots,a_{n-1}}) \\ = \; & \begin{cases} D(f', L_{a_0,a_1,\ldots,a_{n-1}}) + D(f'', L_{a_0,a_1,\ldots,a_{n-1}}), & a_n = 0 \\ D(f', L_{a_0,a_1,\ldots,a_{n-1}}) + 2^{n-1} - D(f'', L_{a_0,a_1,\ldots,a_{n-1}}), & a_n = 1 \end{cases}, \end{aligned}$$

могу се израчунати сва растојања f од линеарних функција, ако се знају сва растојања функција f', f'' (од $n-1$ промењљиве) од линеарних функција: довољно је извршити $O(2^n)$ сабирања. Ако са $T(n)$ означимо временску сложеност овако дефинисаног рекурзивног алгоритма, онда је $T(n) = 2T(n-1) + O(2^n)$, и коначно $T(n) = O(n2^n)$ (видети задатак 8.17).

8.19. Елеменат c_{ij} може се израчунати за *константно* очекивано време. Заиста, $2n$ случајних догађаја $a_{ik} = 0$, $b_{kj} = 0$, $k = 1, 2, \ldots, n$, међусобно су у паровима независни. Због тога су независни и догађаји $d_k = 0$, где је $d_k = a_{ik} \vee b_{kj}$; вероватноћа $P\{d_k = 0\}$ је 1/4. Независни догађаји $A_k = \{d_1 = d_2 = \cdots = d_{k-1} = 1, d_k = 0\}$, $k = 1, 2, \ldots, n$, и $A_{n+1} = \{d_1 = d_2 = \cdots = d_n = 1\}$, чине комплетан систем догађаја (са збиром вероватноћа 1). Ако је $d_1 = 0$, ставља се $c_{ij} := 0$ (догађај A_1 са вероватноћом $\frac{1}{4}$; c_{ij} је израчунато применом једне Булове операције, тј. после једног корака); ако је $d_1 = 1$ и $d_2 = 0$ ставља се такође $c_{ij} := 0$ (догађај A_2 са вероватноћом $\frac{3}{4} \cdot \frac{1}{4}$; израчунавање c_{ij} завршено је после два корака), итд. Уопште, ако је $d_1 = d_2 = \cdots = d_{k-1} = 1$ и $d_k = 0$, ставља се $c_{ij} := 0$ (догађај A_k са вероватноћом $(\frac{3}{4})^{k-1} \cdot \frac{1}{4}$; израчунавање c_{ij} завршено је после k корака, $k = 1, 2, \ldots, n$). У противном се после n корака ставља $c_{ij} := 1$ (догађај A_{n+1} са вероватноћом $(\frac{3}{4})^n$). Очекивани

(средњи) број корака потребних за израчунавање c_{ij} је(видети пример 2.4)

$$\sum_{k=1}^{n} k\left(\tfrac{3}{4}\right)^{k-1}\tfrac{1}{4}+n\left(\tfrac{3}{4}\right)^{n}=\sum_{k=1}^{n} k\left(\tfrac{3}{4}\right)^{k-1}\tfrac{1}{4}+\sum_{k=n+1}^{\infty} n\left(\tfrac{3}{4}\right)^{k-1}\tfrac{1}{4}$$
$$<\ \tfrac{1}{4}\sum_{k=1}^{\infty} k\left(\tfrac{3}{4}\right)^{k-1}=\tfrac{1}{4}\tfrac{1}{1-\frac{3}{4}}=4.$$

Очекивани број корака за израчунавање производа $C = AB$ је дакле мањи од $4n^2 = O(n^2)$.

8.20. Нека је $C = AB$, $A = (a_{ij})$, $B = (b_{ij})$, $C = (c_{ij})$. Довољно је претпоставити да су врсте матрице A отворени Грејови кодови. Најпре се израчунавају елементи прве врсте матрице C, $c_{1j} = \sum_{k=1}^{n} a_{1k}b_{kj}$ ($O(n^2)$ операција) и проналазе индекси $R[i]$ — позиције на којима се разликују i-та и $(i+1)$-а врста матрице A, $i = 1, 2, \ldots, n-1$ (такође $O(n^2)$ операција). Пошто је

$$\begin{aligned} c_{i+1,j}-c_{ij} &= \sum_{k=1}^{n} a_{i+1,k}b_{kj}-\sum_{k=1}^{n} a_{ik}b_{kj}=\sum_{k=1}^{n}(a_{i+1,k}-a_{ik})b_{kj} \\ &= (a_{i+1,R[i]}-a_{i,R[i]})b_{k,R[i]}, \end{aligned}$$

сви остали елементи матрице c_{ij}, израчунавају се кориговањем вредности елемента изнад себе ($c_{i-1,j}$) помоћу једног множења, сабирања и одузимања, $i \geq 2$, $1 \leq j \leq n$. Према томе, при израчунавању производа извршено је укупно $O(n^4)$ операција.

8.21. Решићемо проблем индукцијом по n. Случај $n = 2$ је тривијалан. Претпоставимо да знамо решење за $n \geq 2$ матрица, и нека је дато $n + 1$ матрица које треба помножити. Означимо са $M[i..j]$ производ $M_iM_{i+1}\ldots M_j$. Најбољи начин да се израчуна производ $M[1..n+1]$ је да се израчунају производи $M[1..i]$ и $M[i+1..n+1]$ за неко (за сада непознато) i, па да се онда ти производи помноже. Најбоља вредност за i одређује се израчунавањем цене у свим варијантама. Према индуктивној хипотези знамо најбољи начин за израчунавање $M[1..i]$ (као и одговарајућу цену). Међутим, мора се пронаћи најбољи начин за израчунавање $M[i + 1..n + 1]$. Због тога је згодно појачати индуктивну хипотезу: знамо најбољи начин за израчунавање $M[i..j]$ за све $1 \leq i < j \leq n$. Да би се индуктивна хипотеза проширила на $n + 1$, треба израчунати $M[i..n + 1]$ за све $1 \leq i \leq n$. Овај проблем решава се другом (уметнутом) индукцијом, при чему i узима вредности $n, n - 1, \ldots, 1$. За $i = n$ проблем је тривијалан. Претпоставимо да знамо најбољи начин израчунавања $M[i..n+1]$ и размотримо производ $M[i - 1..n + 1]$. Сада се може извести комплетно свођење: за свако j, $i < j < n + 1$, проверава се цена израчунавања $M[i - 1..j]$ (што знамо по основној индуктивној хипотези), $M[j + 1..n + 1]$ (што знамо по унутрашњој индуктивној хипотези), и цена множења ова два производа; бира се оно j за које је цена израчунавања најмања. Ако се погледа комплетан алгоритам, индуктивна конструкција одговара двема петљама — спољашњој величине n, и унутрашњој величине $n - i$, па се унутрашња петља састоји од $n - i$ корака. Укупан број корака је дакле

$$\sum_{k=1} n \sum_{i=k} n \sum_{j=i} nO(1) = O(n^3).$$

13.9. Примене у криптографији

9.1. Провером могућих помака редом, са помаком 5 добија се порука
CEZAROVAŠIFRASAPOMAKOMPET

За све остале помаке добијају се бесмислени текстови.

9.2. Број могућих кључева је 26^8, па одређивање кључа у најгорем случају траје $0.001 \cdot 26^8/3600 = 58000$ сати. Ако једно шифровање траје $1\mu s$, онда одређивање кључа траје највише 58 сати, односно у просеку 29 сати.

9.3. Ако се упореде шифровање порука P, односно $\bar{P}$ кључевима редом K, односно $\bar{K}$, запажа се да су одговарајући излази после i-те рунде комплементарни, $i = 1, 2, \ldots, 16$.

Заиста, блокови L_0 и R_0 у ова два шифровања су очигледно комплементарни, јер су бити ових блокова копије улазних бита (видети слику 9.4). Даље, због структуре функције f (слика 9.5) важи $f(R_{i-1}, K_i) = f(\bar{R}_{i-1}, \bar{K}_i)$. Према томе, ако је улаз у i-ту рунду $\bar{L}_{i-1}, \bar{R}_{i-1}$, онда је према (9.1) излаз из ње $\bar{R}_{i-1} = \bar{L}_i$ и

$$\bar{L}_{i-1} \oplus f(\bar{R}_{i-1}\bar{K}_i) = \bar{L_{i-1}} \oplus f(R_{i-1}K_i) = \overline{(L_{i-1} \oplus f(R_{i-1}K_i))} = \bar{R}_i.$$

9.4. Могу се формирати два низа a и b дужине 2^{56} чији елементи

$$a[i] = (DES_i(P_1), DES_i(P_2), \ldots, DES_i(P_{10}))\,,$$

$$b[i] = \left(DES_i^{-1}(C_1), DES_i^{-1}(C_2), \ldots, DES_i^{-1}(C_{10})\right),$$

су величине $10 \cdot 8 = 80$ бајтова. Овде је $DES_i(P)$, односно $DES_i^{-1}(C)$ резултат шифрованја P, односно дешифровања C, кључем i, $0 \le i \le 2^{56} - 1$. Низови a и b се сортирају у месту, после чега се једним проласком кроз оба низа (слично као при обједињавању) проналази њихов заједнички елеменат $a[j] = b[k]$, одакле је $DES_j(P_l) = DES_k^{-1}[C_l]$, односно $DES_k(DES_j(P_l)) = C_l$ за свако $l = 1, 2, \ldots, 10$. Другим речима, пронађен је пар кључева (j, k) за DES који десет датих порука трансформишу у десет датих шифрата. При томе је DES за шифровање или дешифровање коришћен $2^{56} \cdot 2 \cdot 10 < 2^{61}$ пута, број упоређивања блокова од 80 бајтова је око $2^{56} \cdot 56 \cdot 2 < 2^{63}$ (што значи да је овај део посла једноставнији од шифровања/дешифровања алгоритмом DES).

9.5. Једини начин да сви поткључеви K_i буду једнаки је да се обе група од по 28 бита кључа (које се ротирају после сваке рунде) буду састављене од једнаких бита, да их ротације не би мењале. Постоје четири такве могућности. Користећи табелу 9.2 добијамо да су то следећи кључеви (хексадецимално):

Кључ	Испермутовани кључ	
0101010101010101	00000000	00000000
ФЕФЕФЕФЕФЕФЕФЕФЕ	ФФФФФФФФ	ФФФФФФФФ
1Ф1Ф1Ф1Ф1Ф1Ф1Ф1Ф	00000000	ФФФФФФФФ
ЁЁЁЁЁЁЁЁ	ФФФФФФФФ	00000000

9.6. До факторизације n може се доћи пробабилистичким алгоритмом типа Лас Вегас. Најпре се израчунава $k = de - 1$, број који је умножак $\phi(n)$. Пошто је $\phi(n) = (p-1)(q-1)$ паран број, биће $k = 2^t r$, где је $t \ge 1$ и r је непаран број. За произвољан број g, узајамно прост са n важи $g^k \equiv 1 \pmod{n}$, па је $g^{k/2}$ корен из јединице по модулу n. Корена из јединице по модулу n има четири: њихови остаци по модулу p и q могу бити $(1,1)$, $(-1,-1)$, $(1,-1)$ или $(-1,1)$; прва два од њих су ± 1, а остала два су $\pm x$, где x задовољава конгруенције $x \equiv 1 \pmod{p}$ и $x \equiv -1 \pmod{q}$. Познавање било ког од ова два последња корена из јединице по модулу n, омогућава налажење факторизације n израчунавањем $\text{NZD}\,(x-1, n)$. За случајно изабрани број g, $1 < g < n$, са вероватноћом бар $1/2$ међу бројевима $g^{k/2}$, $g^{k/4}, \ldots$, $g^{k/2^t} \pmod{n}$, налази се бар један од коренова из јединице који открива факторизацију n.

9.7. Полазимо од чињенице да $\phi(n)|ed-1$; потребно је да за нови јавни кључ e' одредимо неко d' тако да је $\phi(n)|e'd'-1$. Најпре израчунавамо резултат дељења $ed-1$ свим заједничким делиоцима са e', тј. $g = (ed-1)/\text{NZD}\,(ed-1, e')$. Пошто су ти заједнички делиоци узајамно прости са $\phi(n)$ (начин израчунавња кључева за RSA подразумева да су e' и $\phi(n)$ узајамно прости), биће $\phi(n)|g$. Затим се помоћу Еуклидовог алгоритма одређује такво d' да $g|e'd'-1$, а тиме и $\phi|e'd'-1$. Експонент d' може се користити за дешифровање порука шифрованих експонентом e'.

Друго решење. Особа B може, знајући n, e и d, да факторише n (видети задатак 9.6), после чега лако може да израчуна тајни кључ d' који одговара јавном кључу e'.

Напомена. Ово запажање показује да се RSA модуо n не сме користити два пута.

9.8. Јавни експонент e бира се тако да буде узајамно прост са $\chi(n)$ и мањи од $\chi(n)$. Тајни експонент d добија се помоћу Еуклидовог алгоритма, као решење конгруенције $de \equiv 1$

(mod χ)(n). Шифровање блока p даје $c = p^e \pmod{n}$, а дешифровањем c добија се $p^{ed} \equiv p \pmod{n}$, јер је ed истовремено и облика $1 + l_1(p-1)$ и облика $1 + l_2(q-1)$, где су l_1 и l_2 природни бројеви.

9.9. Једно могуће решење је да се на почетку поруке резервише простор за поље у коме се записује дужина поруке; претпоставка је да су дужине свих порука мање од неке фиксиране вредности.

13.10. Редукције

10.1. Проблем се решава свођењем на проблем налажења најкраћих путева из датог чвора у посебном графу $G = (V, E)$ одређеном низовима A и B. Скуп чворова G је $V = \{(i,j) \mid 0 \le i \le n,\ 0 \le j \le m\}$, а у чвор (i,j) воде гране из чворова

- $(i, j-1)$ ако је $0 \le i \le n$, $1 \le j \le m$ (уметање знака a_i, цена ci),
- $(i-1, j)$ ако је $1 \le i \le n$, $0 \le j \le m$ (брисање знака b_j, цена cj),
- $(i-1, j-1)$ ако је $1 \le i \le n$, $1 \le j \le m$ (замена a_i са b_j; цена је 0 ако $a_i = b_j$, односно 1 ако $a_i \ne b_j$).

У графу G проналазе се најкраћи путеви од чвора $(0,0)$ до свих осталих чворова, специјално до чвора (n,m); тај пут одговара низу едит операција најмање цене који A трансформише у B.

10.2. Показаћемо да се проблем инклузије интервала може решити помоћу алгоритма за налажење максималних тачака. Сваком интервалу $I_j = (L_j, D_j)$ придружујемо тачку $A_j = (-L_j, D_j)$. Тада је интервал I_j садрзан у интервалу I_k акко је тачка A_j мања од одговарајуће тачке A_k. Другим речима, интервал I_j је садржан у неком другом интервалу акко одговарајућа тачка A_j није максимална.

10.3. Сваки чвор који одговара факултету заменимо са два чвора, повезана гранама са свим студентима који су се квалификовали за пријем на тај факултет. Тиме је проблем сведен на обичан проблем бипартитног упаривања.

10.4. Проблем се може решити свођењем на обичан проблем бипартитног упаривања. Полазећи од бипартитног графа $G = (U, V, E)$ са скупом наставника U, курсева V и грана E, формира се граф $G' = (U', V', E')$ тако да сваком наставнику $u \in U$ одговарају два чвора $u_0, u_1 \in U'$, сваком курсу $v \in V$ одговарају два курса $v_0, v_1 \in V'$, а свакој грани $(u,v) \in E$ одговарају четири гране $(u_0, v_0), (u_0, v_1), (u_1, v_0), (u_1, v_1) \in E'$. Лако се показује да сваком обичном бипартитном упаривању у G' одговара такво додељивање курсева наставницима у G, да сваки наставник држи највише два курса, и сваки курс држе највише два наставника. При томе је укупан број курсева једнак величини одговарајућег упаривања у G'.

10.5. За дати граф $G = (V, E)$ формира се нови граф $G' = (V', E')$ на следећи начин. Сваки чвор $v_i \in V$ замењује се са $b(v_i)$ чворова $v_{i,1}, v_{i,2}, \ldots, v_{i,b(v_i)}$. Поред тога, за сваку грану $(v_i, v_j) \in E$ уводе се још два чвора $v_{i,j,0}$ и $v_{j,i,0}$. За сваку грану $(v_i, v_j) \in E$ у G' постоје гране $(v_{i,k}, v_{i,j,0})$, $1 \le k \le b(v_i)$, $(v_{i,j,0}, v_{j,i,0})$, и $(v_{j,k}, v_{j,i,0})$, $1 \le k \le b(v_j)$, видети слику 13. Да би се одредило оптимално b-упаривање M у G, може се одредити оптимално упаривање M' у G', и затим га трансформисати у оптимално b-упаривање у G на следећи начин: $(v_i, v_j) \in M$ акко су од грана G' придружених грани $(v_i, v_j) \in E$ у упаривање M' укључене две гране (од ових грана у упаривање M може бити укључена или једна или две гране). Заиста, најпре се уверавамо да је скуп грана M b-упаривање. Упаривање M' у G' трансформише се у упаривање M'', у коме се свим комплетима грана придруженим гранама (v_i, v_j) из G са једном граном у упаривању M' бира за M'' "средња" грана $(v_{i,j,0}, v_{j,i,0})$ из комплета. Из осталих комплета грана за M' су изабране по две гране, по једна грана са сваке стране средње гране; за фиксирани чвор $v_i \in V$ у упаривању M је највише $b(v_i)$ грана њему суседних, јер у G' скуп чворова $v_{i,1}, v_{i,2}, \ldots, v_{i,b(v_i)}$ може да обезбеди највише $b(v_i)$ комплета са по две гране у упаривањуM': сваки од тих комплета грана покрива по један од чворова — реплика чвора v_i.

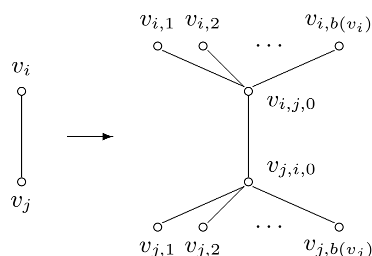


Рис. 13. Илустрација уз задатак 10.5.

Да бисмо се уверили да је упаривање M оптимално b-упаривање у G, претпоставимо да у њему има k грана (односно да у упаривању M' има $|E|+k$ грана), и да у G постоји упаривање M''' са $k' > k$ грана. Тада се може, полазећи од M''', формирати (обично) упаривање M^{IV} у G' које из комплета грана придружених грани из M (односно $E \setminus M$) садржи две (односно једну) грану; број грана у M^{IV} је $2k' + (|E| - k') = |E| + k' > |E| + k$, супротно претпоставци да је M' оптимално упаривање у G'. Према томе, M јесте оптимално b-упаривање у G.

10.6. Полазећи од графа G конструише се нови граф H, у коме сваком чвору w из G одговарају два чвора w_1 ("завршни") и w_2 ("полазни"). За сваку грану (w, u) из G додајемо грану (w_2, u_1) цене 0 у H (другим речима, преправљамо све гране да воде од полазног ка завршном чвору). Поред тога, додајемо грану (u_1, u_2) цене $(c(u)$ за сваки чвор u из G. Тиме је дати проблем сведен на проблем одређивања најкраћих путева од v_2 до свих полазних чворова у H.

10.7. Довољно је ограничења са неједнакостима типа "$\geq$" помножити са (-1).

10.8. Нека је $n = |V|$, $m = |E|$ и нека је $x_i = w(e_i)$ тежина i-те гране $e_i \in E$, $1 \leq i \leq m$. Потребно је максимизирати вредност $z = \sum_{i=1}^{m} x_i$, под условом да је

$$\sum_{e_i \text{ је суседна са } v} x_i \leq h \quad \text{за сваки чвор } v \in V,$$

и $x_i \geq 0$, $1 \leq i \leq m$.

10.9. Могу се преправити неједнакости – ограничења у основној варијанти. Лакше је приметити да свака особа треба да поклони свој максимални износ.

10.10. Доказ се изводи тако што се покаже да се проблем сортирања (за који се зна доња граница сложености — под претпоставком да се користи модел стабла одлучивања) може свести на овај проблем, слично као што је то урађено за проблем конструкције простог многоугла са задатим скупом темена. Задатих n бројева, улаз за проблем сортирања, трансформишу се у n тачака $P_1, P_2, \ldots P_n$ на кружници са центром O у координатном почетку, таквих да дуж OP_i са x-осом заклапа угао $ax_i + b$; константе a и b бирају се тако да све вредности $ax_i + b$ припадају интервалу $[0, 2\pi)$. На тачке $P_1, P_2, \ldots P_n$ примени се алгоритам за уређење темена конвексног многоугла (све тачке леже на кружници, па су темена конвексног многоугла). На тај начин (читајући тачке са кружнице супротно од казаљке на сату, полазећи од x-осе) добија се сортирани низ тачака $P_{i_1}, P_{i_2}, \ldots, P_{i_n}$, а тиме и уређење $x_{i_1} < x_{i_2} < \cdots < x_{i_n}$ — решење проблема сортирања. Пошто је доња граница сложености проблема сортирања $\Omega(n \log n)$, тиме је доказано да је и доња граница сложености уређења темена конвексног многоугла $\Omega(n \log n)$.

10.11. Да би се израчунао квадрат матрице, довољно је пет множења матрица двоструко мање димензије. Међутим, то су производи две *произвољне*, а не *једнаке* матрице.

10.12. Нека су A и B две произвољне квадратне матрице реда n. Потребно је свести израчунавање производа AB на израчунавање производа неких доњих и горњих троугаоних матрица. У том циљу се A и B могу представити (не једнозначно) у облику збира једне доње

и једне горње троугаоне матрице, $A = A_d + A_g$, $B = B_d + B_g$. Одатле се добија

$$AB = (A_d + A_g)(B_d + B_g) = A_dB_d + A_dB_g + A_gB_d + A_gB_g.$$

Потребно је још пронаћи начин да се доње троугаоне матрице "претварају" у горње троугаоне и обрнуто. Циљ се постиже помоћу специјалне пермутационе матрице P, која има јединице само на споредној дијагонали, а остали елементи су јој нуле. Множењем слева матрицом P произвољне матрице C са врстама редом $c_1, c_2, \ldots, c_n$, добија се матрица C' са врстама редом $c_n, c_{n-1}, \ldots, c_1$, па је C' горња троугаона акко је C' доња троугаона, и обрнуто. Ако је E јединична матрица реда n, онда је очигледно $P^2 = E$, односно $P^{-1} = P$. Матрица P^2 је јединична матрица. Друго средство је за постизање истог циља је операција транспоновања, која такође доњу троугаону матрицу преводи у горњу троугаону и обрнуто. Помоћу ових двеју операција сложености $O(n^2)$ сва четири горња производа троугаоних матрица своде се на израчунавање производа доње и горње троугаоне матрице:

- $A_dB_d = (P((PB_d^T)A_d^T))^T$ (два пермутовања врста и три транспоновања),
- A_dB_g (производ жељеног облика),
- $A_gB_d = (B_d^TA_g^T)^T$ (три транспоновања),
- $A_gB_g = P((PA_g)B_g)$ (два пермутовања врста).

До ових израза се лако долази после мало експериментисања.

10.13. Слично као у решењу задатка 10.12, користи се пермутациона матрица P реда n таква да је њено множење матрице слева еквивалентно замени прве и последње врсте, друге и претпоследње врсте, итд, па доњу тругаону матрицу претвара у горњу троугаону и обрнуто. Матрице A и B које треба помножити разлажу се у збирове једне доње и једне горње троугаоне матрице, $A = A_d + A_g$, $B = B_d + B_g$, а онда се производ $AB = A_dB_d + A_dB_g + A_gB_d + A_gB_g$ израчунава тако што се сва четири производа са десне стране сведу на множење две труогаоне матрице, и неке допунске операције сложености $O(n^2)$:

- $A_gB_d = P((PA_g)B_d)$ (два пермутовања врста),
- $A_gB_g = (B_g^TA_g^T)^T$ (два транспоновања),
- $A_dB_g = P(B_g^T(PA_d)^T)^T$ (два пермутовања врста и три транспоновања).

10.14. Показаћемо како се наведени алгоритам за налажење MCST може искористити за сортирање. Нека је дат низ $x_1, x_2, \ldots, x_n$ различитих бројева које треба сортирати. Сваком броју x_i придружујемо тачку P_i на x-оси са x-координатом x_i. Лако је видети да MCST у овом случају мора да повезује сваку тачку са њеним суседима на правој, односно стабло је пут — низ тачака без гранања. Стабло има два листа — минимални и максимални елеменат низа. Знајући минимално повезујуће стабло, лако долазимо до уређења бројева за линеарно време: проналазимо најлевију тачку, нпр. x_i, а затим низ тачака одређен стаблом даје уређени низ бројева. Тиме је доказано да је доња граница за проблем налажења MCST по моделу стабла одлучивања (по том моделу је доказана доња граница за проблем сортирања) једнака $\Omega(n \log n)$.

10.15. Потребно је наћи такво пресликавање бројева из $A \cup B$ у скуп S тачака у равни, да се одговор на питање о дисјунктности скупова A и B добија налажењем дијаметра S. Нека сви бројеви из A и B припадају интервалу (m, M) (бројеви m и M одређују се помоћу $O(n)$ упоређивања). Сваки број $z \in A \cup B$ пресликава се у тачку $D = \phi(z)$ на јединичној кружници са центром у координатном почетку O, тако да полуправа OD са x-осом гради угао $\pi(z-m)/(M-m)$, ако је $z \in A$, односно $\pi+\pi(z-m)/(M-m)$, ако је $z \in B$. Све добијене тачке припадају кружници пречника 2, па је дијаметар добијеног скупа тачака мањи или једнак од 2. Прецизније, дијаметар је једнак 2 акко у скупу постоје две дијаметрално супротне тачке, односно акко постоје два броја $z_1 \in A$ и $z_2 \in B$ таква да је $\pi + \pi(z_2 - m)/(M - m) = \pi(z_1 - m)/(M - m) + \pi$, тј. $z_1 = z_2$. Дакле, пресликавањем $\leq n$ тачака из $A \cup B$ и решавањем проблема ДМ за добијене тачке, добија се решење проблема ДС.

13.11. NP-комплетност

11.1. Нека су B и C два нетривијална проблема из P (тј. за оба постоје улази са одговором "да" и улази са одговором "не"). Нека су u_0 и u_1 два улаза за проблем C, таква да је за u_0 решење "не", а за u_1 "да" (односно $u_0 \notin L_C$ и $u_1 \in L_C$). За свођење проблема B на проблем C може се искористити следећи полиномијални алгоритам: улазу v за проблем B придружује се $\phi(v) = u_0$ ако $v \notin L_B$, односно $\phi(v) = u_1$ ако $v \in L_B$. Тада за произвољан улаз v за проблем B важи $v \in L_B$ акко $\phi(v) \in L_C$.

11.2. Функција n^k је полином по n, али експоненцијално зависи од k. Пошто је k део поставке проблема (и може да достигне n), временска сложеност овог алгоритма није полиномијална.

11.3. Тражени 3SAT израз је

$$
\begin{aligned}
&(x + y + a_1)(\bar{a}_1 + a_2 + \bar{z})(\bar{a}_2 + a_3 + w)(\bar{a}_3 + u + \bar{v}) \\
&(\bar{x} + \bar{y} + a_4)(\bar{a}_4 + a_5 + z)(\bar{a}_5 + a_6 + \bar{w})(\bar{a}_6 + u + v) \\
&(x + \bar{y} + a_7)(\bar{a}_7 + a_8 + \bar{z})(\bar{a}_8 + a_9 + w)(\bar{a}_9 + u + \bar{v}) \\
&(x + \bar{y} + a_{10})(x + \bar{y} + \bar{a}_{10}).
\end{aligned}
$$

11.4. Доказ се заснива на редукцији са обичног проблема покривач грана. Нека је $G = (V, E)$ произвољни неусмерени граф, и нека је U скуп чворова непарног степена у G. Додавањем три нова чвора x, y, и z, међусобно повезана у троугао, и повезујући x са свим чворовима из U, добијамо нови граф G', чији сви чворови имају паран степен. Граф G' има покривач грана величине k акко G има покривач грана величине $k-2$. Заиста, од произвољног покривача G величине $k-2$ се додавањем чворова x и y добија покривач грана графа G' величине k. Обрнуто, ако у G' постоји покривач грана величине k, онда тај покривач мора да садржи бар два од три чвора x, y и z; тада је и скуп од k чворова који се добија заменом та два чвора са x, y такође покривач грана G'; пошто x, y не покривају ни једну грану из G, њиховим уклањањем из скупа добија се покривач грана графа G величине $k-2$.

11.5. Проблем је у класи NP, јер се за сваки од највише $\binom{n}{k}^2$ избора два подскупа од k чворова, полиномијалним алгоритмом проверава да ли је први подскуп клика, а други – независан скуп. С друге стране, проблем клика, за који се зна да је NP-комплетан, своди се на овај проблем. Заиста, за дати граф $G = (V, E)$ и природни број k (произвољни улаз за проблем клика), формира се граф $G' = (V', E)$ са скупом чворова V' који се од V добија додавањем k изолованих чворова. Решавањем проблема налажења k-клике и k-независног скупа за пар G', k, проналази се клика величине k у G', која је истовремено и клика величине k у G (јер клика не може да садржи ни један од додатих изолованих чворова). Тиме је доказана NP-комплетност проблема из формулације задатка.

11.6. Проблем је у класи NP, јер се за изабрани подскуп од k чворова, за полиномијално време може проверити да ли је индуковани граф ациклички. На овај проблем може се свести проблем 3SAT. Нека је $E = C_1C_2\ldots C_n$ произвољни улаз за 3SAT. Полазећи од E, конструишемо граф G са $4n+1$ чворова на следећи начин. За сваку клаузу C_i додајемо четири чвора — један који одговара клаузи и три који одговарају њеним литералима. Та четири чвора повезана су са свих шест грана. Поред тога, свака два чвора који одговарају некој променљивој x и њеном комплементу $\bar{x}$ повезују се граном. На крају додајемо још један чвор r и повезујемо га са свим чворовима придруженим литералима (не клаузама). Тврдимо да граф G има подскуп од $2n+1$ чворова који индукује ациклички граф акко је израз E задовољив.

- Ако је израз задовољив, онда постоји придруживање вредности променљивим такво да је у свакој клаузи тачан бар један литерал. Подскуп који се састоји од чвора r, n чворова који одговарају клаузама и n чворова (по један из групе сваке клаузе) који одговарају тачним литералима, индукује ациклички граф (прецизније, стабло у коме је корен r повезан са чворовима тачних литерала, а чворови клауза

су листови; по конструкцији између чворова тачних литерала не постоји ни једна грана).

- Ако постоји подскуп S од $2n+1$ чворова, који индукује ациклички граф, онда S садржи r и по тачно два чвора из сваке клаузе (ако би из неке клаузе садржао три чвора, онда би садржао троугао, јер су сви чворови у групи једне клаузе међусобно повезани). Пошто S садржи r, онда S не може да садржи истовремено два чвора који припадају некој променљивој x и њеном комплементу $\bar{x}$. Према томе, S одређује такво придруживање вредности променљивима, које израз E чини тачним.

11.7. Згодно је уопштити претпоставке о изразу E: нека се у њему свака променљива x_i и њен комплемент $\bar{x}_i$, $i = 1, 2, \ldots, n$, појављују највише по једном. Укупан број литерала у E је због тога највише $2n$. У зависности од тога да ли се и где појављују x_n и $\bar{x}_n$, израз E може се написати у једном од следећих облика:

- $(x_n + A)(\bar{x}_n + B)E'$ (ако се x_n и $\bar{x}_n$ појављују у различитим клаузама);
- $(x_n + A)E'$ (ако се само x_n појављује у некој клаузи);
- $(\bar{x}_n + B)E'$ (ако се само $\bar{x}_n$ појављује у некој клаузи).

Овде су A, B остаци клауза у којима се појављују x_n, односно $\bar{x}_n$, а E' је остатак израза E, у коме се не појављују ни x_n ни $\bar{x}_n$. Изрази $(x_n + A)E'$, односно $(\bar{x}_n + B)E'$ су задовољиви акко је задовољив израз E'. Слично, израз $(x_n + A)(\bar{x}_n + B)E'$ је задовољив акко је задовољив израз CE', где је C клауза која се од $A + B$ добија избацивањем поновљених литерала, односно 1, ако се у $A + B$ појављује и нека променљива y и њена негација $\bar{y}$. У сва три случаја добијени израз (E', E', CE') зависи само од променљивих $x_1, x_2, \ldots, x_{n-1}$, и сваки литерал се у преосталом изразу појављује највише једном. Дакле, овим поступком се (за полиномијално време) елиминише из израза једна променљива, па је одговарајући алгоритам полиномијалне сложености.

11.8. Искористићемо редукцију са проблема 3SAT. Нека је $C = x + y + z$ нека клауза у произвољном 3SAT изразу. Ову клаузу замењујемо са следеће три (при чему су сви симболи a_i нове променљиве, које се уводе само за клаузу C): $(x + a_1 + a_2)$, $(y + a_3 + a_4)$, $(z + a_5 + a_6)$. У разматраној варијанти проблема 3SAT у овим клаузама треба да вредност 1 има по тачно једна променљива. Да би се обезбедило да вредност 1 има бар један од литерала x, y, z, додајемо још шест клауза, које гарантују да од променљивих a_1, a_3, a_5, односно a_2, a_4, a_6, вредност 1 има највише једна: $(a_1 + a_3 + a_7)$, $(a_3 + a_5 + a_8)$, $(a_5 + a_1 + a_9)$, $(a_2 + a_4 + a_{10})$, $(a_4 + a_6 + a_{11})$, $(a_6 + a_2 + a_{12})$. Сада се може показати да је бар један од литерала x, y, z тачан акко се новим променљивим $a_1, a_2, \ldots, a_{12}$ могу доделити такве вредности, да у свакој новој клаузи тачно један литерал има вредност 1.

- Ако је бар један од литерала x, y, z тачан, онда се могу разликовати три случаја, кад је број тачних међу њима 1, 2 и 3. У сва три случаја се могу вредности нових променљивих изабрати тако да у свакој новој клаузи тачно један литерал има вредност 1. (видети табелу 2; она због симетрије обухвата све могуће случајеве).
- Претпоставимо да у свакој новој клаузи тачно један литерал има вредност 1, и да је $x = y = z = 0$. Тада у групама (a_1, a_2), (a_3, a_4), (a_5, a_6) тачно по једна променљива има вредност 1, а у групама (a_1, a_3, a_5) и (a_2, a_4, a_6), највише по једна променљива има вредност 1 (због друге, односно треће групе додатих клауза). Ако је на пример $a_1 = 1$, онда је $a_2 = 0$; $a_3 = a_5 = 0$; $a_4 = a_6 = 1$ – контрадикција. Случај $a_1 = 0$, $a_2 = 1$ је симетричан. Према томе, ако у свакој од девет нових клауза тачно један литерал има вредност 1, онда је бар један од литерала x, y, z тачан.

11.9. Свешћемо проблем клика на овај проблем. Нека је $G = (V, E)$, k, произвољан улаз за проблем клика. Циљ је трансформисати граф G у регуларан граф R, такав да се проблем клика на G може решити решавањем проблема клика на регуларном графу R. Јасно је да се не могу просто додавати гране у G док не постане регуларан, јер се тако могу појавити нове, веће клике. Графу G морају се додавати нови чворови и гране, тако да се не формирају нове клике. Нека је d' максимални степен неког чвора у G, и нека је $d = d'$ ако је d' парно,

x	y	z	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
1	0	0	0	0	1	0	0	1	0	0	1	1	0	0
1	1	0	0	0	0	0	1	0	1	0	0	1	1	1
1	1	1	0	0	0	0	0	0	1	1	1	1	1	1

Таблица 2. Уз задатак 11.8.

односно $d = d' + 1$ у противном. За сваки чвор v у G степена $d(v) < d$ додајемо $d - d(v)$ нових чворова и све њих граном повезујемо са v. Ако је $|V| = n$, онда је укупан број новододатих чворова $dn - \sum_{i=1}^{n} d(v_i) = dn - 2|E|$. Овај број је паран, јер је d по конструкцији парно. Сви стари чворови у новом графу имају сада исти степен d, а нове клике се нису појавиле (јер је сваки нови чвор повезан само са једним чвором). Остаје још проблем нових чворова, јер они имају степен 1. Њихови степени могу се повећати на d, не додајући нове клике, повезујући их гранама на следећи начин. Скуп нових чворова поделимо на два једнака подскупа, а онда сваки чвор из једног подскупа повезујемо са тачно $d - 1$ чворова из другог подскупа (при томе само треба водити рачуна о томе да степени нових чворова из другог скупа не пређу d, што није тешко). Нови чворови сада индукују бипартитни граф, који не садржи клику већу од два.

11.10. За свако могуће разлагање скупа $n = |V|$ чворова на k подскупова (којих има мање од $n!\binom{n-1}{k-1}$), за полиномијално време се може проверити да ли је то разлагање на независне скупове. С друге стране, на овај проблем може се свести проблем k-обојивости графа. Произвољан алгоритам који решава постављени проблем може се искористити за k-бојење графа G: чворови који одговарају истовремено обављеним пословима чине независан скуп (између њих нема грана), па се могу обојити једном истом бојом. Проблем k-обојивости графа је NP-комплетан, јер је у класи NP, а на њега се своди (као специјални случај) проблем 3-обојивостграфа.

11.11. Посматрајмо фамилију графова приказану на слици 14. Алгоритам бојења претрагом (слика 11.12) мора да провери редом свих $3 \cdot 2^{n-4}$ исправних бојења подграфа који чине чворови $1, 2, \ldots, n-3$. Сваки тај покушај завршава се неуспехом кад се дође до чвора n, који има три суседна чвора $n-1, n-2$ и $n-3$ обојена различитим бојама. Запажа се да је у овом алгоритму битна нумерација чворова: ако би чворови $n, n-1, n-2, n-3$ били пренумерисани редом са $1, 2, 3, 4$, онда би се после $O(1)$ корака установило да не постоји 3-бојење овог графа.

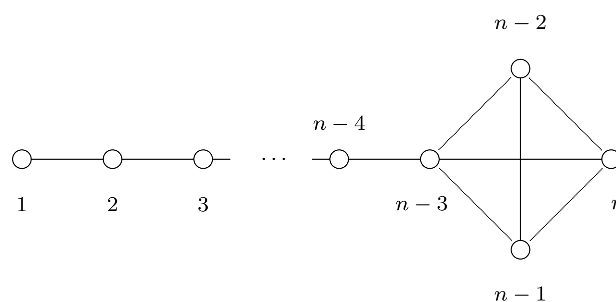


Рис. 14. Илустрација уз задатак 11.11.

13.12. Паралелни алгоритми

12.1. У сваком кораку треба сабирати по два броја (број бита збира је $O(n + \log k)$, па је број корака за свако сабирање $O(\log(n + \log k))$. У првој фази се k бројева деле на парове и израчунава се збир сваког пара, што се паралелним алгоритмом (одељак 12.3.1) на $kn/2$ процесора извршава за $O(\log n)$ корака. Даље се на исти начин наставља са добијеним збировима; број збирова после сваке фазе постаје двоструко мањи, па је укупан број фаза $O(\log k)$. Сложеност алгоритма је дакле $O(\log n \log k)$, ако се занемари да међурезултати имају више од n бита (тачнија оцена је $O(\log(n + \log k) \log k)$).

12.2. Довољно је да сваки процесор P_i израчунава своју сопствену копију $D[i]$ променљиве D. Дакле, иницијализација је паралелни корак $D[i] := 1$, а остала појављивања D замењују се са $D[i]$. Других конфликата при приступању меморији нема: само процесор P_i чита $R[N[i]]$ (ранг свог суседа) и евентуално позицију његовог суседа $N[N[i]]$; процесори чијим се елементима пронађе ранг, искључују се и не изазивају конфликте.

12.3. Ако у првом кораку неки процесор наиђе на два једнака броја, онда он уписује јединицу у променљиву придружену броју са мањим индексом. Сложеност алгоритма је и даље $O(1)$.

12.4. (а) Користи се метод удвостручавања. У првом кораку процесор P_1 копира $A[1]$ у $A[2]$. Индуктивна хипотеза: пре почетка i-тог корака $A[1]$ је ископирано на позиције од 1 до 2^{i-1} (што је за $i = 2$ тачно). У кораку i учествује 2^{i-1} процесора, који у једном паралелном кораку копирају елементе $A[1], A[2], \ldots, A[2^{i-1}]$ у елементе $A[2^{i-1}+1]$, $A[2^{i-1}+2]$, $\ldots$, $A[2^i]$.

(б) Да би се поправила ефикасност алгоритма, користи се Брентова лема. Прецизније, нека је $n = 2^k$, нека је на располагању $\lfloor n/\log_2 n \rfloor = \lfloor 2^k/k \rfloor$ процесора, и нека је i такав број да је $2^{i-1} < 2^k/k < 2^i$. Тада се у $i - 1$ корака $A[1]$ копира на локације од 1 до 2^{i-1} описаним алгоритмом. Затим се помоћу $2^{i-1} < 2^k/k$ процесора сваки од 2^{i-1} тих елемената паралелно копира $2^k/2^{i-1} - 1$ пут (елеменат са локације j, $0 \le j < 2^{i-1}$ процесор P_j копира редом на локације $j + m \cdot 2^{k-i}$, $m = 1, 2, \ldots, 2^k/2^{i-1} - 1$); при томе је $i - 1 < k = \log_2 n$ и $2^k/2^{i-1} - 1 = 2 \cdot 2^{k-i} - 1 < 2k - 1 < 2\log_2 n$, па је укупно време извршавања мање од $3\log_2 n$, односно $O(\log n)$.

12.5. Овај проблем своди се на проблем паралелног префикса за операцију $+$, при чему су почетне вредности $A[i] = 1$ за $2 \le i \le n$ (ова иницијализација извршава се у једном паралелном кораку — константу 1 може генерише независно сваки процесор).

12.6. Са $n+1$ процесора $P_0, P_1, \ldots, P_n$ на располагању, вредност полинома $\sum_{i=0}^n a_i x^i$ степена n може се израчунати паралелно за $O(\log n)$ корака на следећи начин. Најпре се сваком процесору обезбеди копија аргумента x (нпр. алгоритмом паралелни префикс сложености $O(\log n)$, при чему је операција сабирање, почетно стање процесора P_0 је x, а осталих 0). Затим се још једном изврши паралелни префикс за множење, чиме се постиже да процесор P_i има израчунату вредност x^i, $0 \le i \le n$. У наредном паралелном кораку P_i израчунава производ $a_i x^i$, $0 \le i \le n$. На крају се ови производи сабирају трећом применом паралелног префикса — после тога се у процесору P_n налази израчуната вредност полинома. Ефикасност овог алгоритма је $O(1/\log n)$, а и не користи се Хорнерова шема. Ефикасност се може поправити применом Брентове леме тако са се користи само $k = n/\log n$ процесора. Полином се представља у облику полинома степена k од x^l, $l = n/k$, чији су коефицијенти "мали" полиноми степена $l - 1$, чије вредности поједини процесори израчунавају секвенцијално Хорнеровом шемом за време $O(l) = O(\log n)$. Вредност "великог" полинома степена $k = n/\log n$ израчунава се на већ описани начин на k процесора за време $O(\log k) = O(\log n)$. Ефикасност конструисаног алгоритма је $O(n/((n/\log n)\log n)) = O(1)$.

12.7. Проблем се решава слично као проблем налажења максимума. Променљивим z, $v_1, v_2, \ldots, v_n$ на почетку се додељује вредност 0. Процесор P_{ij} упоређује x_i и x_j, па ако су различити, уписује 1 у z, $1 \le i < j \le n$. После овог паралелног корака z ће имати вредност 1 акко за бар неко i важи $x_i = 1$, и за бар неко j важи $x_j = 0$; тада је вредност дисјункције 1. У противном, ако је $z = 0$, онда P_1 копира x_1 у z (тада је $x_1 = x_2 = \cdots = x_n = z$).

12.8. Слика траженог кола за $n = 8$ приказана је на слици 15. За $n = 2^k$ коло се састоји од k група по n процесора P_{0j}, P_{1j}, ..., $P_{n-1,j}$, $j = 0, 1, \ldots, k-1$. Процесор P_{ij} у групи j за $0 \leq i < 2^j$ преузима већ израчунату вредност i-тог префикса од процесора $P_{i,j-1}$; за $2^j \leq i < n$ процесор P_{ij}, узимајући резултате x_i, x_{i-2^j} од процесора $P_{i,j-1}$, $P_{i-2^j,j-1}$, израчунава производ $x_i \star x_{i-2^j}$, и резултат смешта у (своју) променљиву x_i. Индукцијом по $j = 0, 1, \ldots, k-1$ доказује се да променљива x_i после проласка кроз процесор P_{ij} садржи вредност $x_0 \star x_1 \star \ldots \star x_i$ ако је $i < 2^j$, односно $x_{i-2^j+1} \star x_{i-2^j+2} \star \ldots \star x_i$, ако је $2^j \leq i < n$. За $j = k-1$ одатле се добија да променљива x_i после изласка из кола садржи вредност одговарајућег префикса.

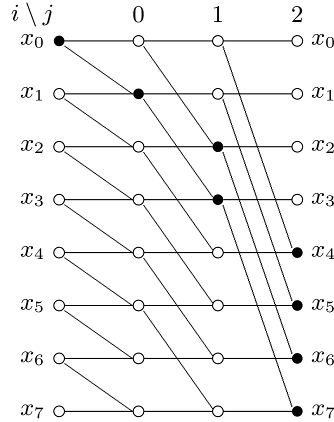


Рис. 15. Илустрација уз задатак 12.8.

12.9. Пошто је на располагању довољан број процесора, а дозвољено је истовремено читање са исте локације, може се најпре формирати матрица A са елементима

$$a_{ij} = \begin{cases} 1, & x_i < x_j \\ 0, & x_i > x_j \end{cases},$$

за шта је потребно $O(1)$ паралелних корака. Ранг i_r елемента x_r (позиција x_r у сортираном редоследу) једнак је броју елемената x_j мањих од x_r увећаном за један, односно $s_r + 1$, где је s_r збир елемената r-те врсте матрице A. Помоћу n^2 процесора збирови врста у A могу се паралелно израчунати за $O(\log n)$ корака (посебним турниром за сваку врсту). Затим се у једном паралелном кораку x_r копира на позицију i_r излазног вектора, $1 \leq r \leq n$..

12.10. Сваком пару елемената $A[i]$, $B[j]$, $1 \leq i < j \leq n$, додељује се по један процесор P_{ij}. Процесор P_{ij} најпре упоређује $A[i]$ са $B[j]$, а онда $A[i]$ са $B[j+1]$. Ако је $A[i]$ између $B[k]$ и $B[k+1]$, онда процесор P_{ik} то открива и закључује да у излазном вектору елеменат $A[i]$ треба да буде на позицији $i+k$. На исти начин утврђују се позиције елемената вектора B. Сва копирања улазних елемената у излазни вектор изводе се у једном паралелном кораку.

12.11. Посао се може поделити у два дела. Први део је "проширивање" вектора A, B у векторе–матрице A', B' дужине nm, $A'[km+r] = A[k]$, $B'[km+r] = B[r]$ (ово је неопходно да би се избегло истовремено читање истог елемента вектора од стране више процесора). Други део је паралелно множење $C[i] = A'[i]B'[i]$, $0 \leq i \leq nm-1$. Проширивање се са p паралелних процесора може извршити за $O(\log p)$ корака у оквиру једне од mn/p група по p елемената, па је трајање прве фазе $O(mn \log p/p)$ (претпоставља се да је $p \leq mn$). Множење се извршава за $O(mn/p$ паралелних корака (обрада сваке од mn/p група извршава се за један паралелни корак).

12.12. Проблем се може решити елегантно свођењем на проблем паралелног префикса (иако га је могуће решити и директно). Најпре се паралелно израчунају префикси на копији вектора $Oznaka$, са операцијом $+$. Вредност префикса за сваки слог коме $Oznaka$ има вредност 1 биће једнака његовој позицији у препакованом вектору. Затим се вредности вектора $Oznaka$ могу комплементирати ($0 \to 1$, $1 \to 0$), и на исти начин израчунати позиције преосталих слогова (оних за које је $Oznaka$ имала вредност 0). Кад су сви индекси израчунати, онда се стварно копирање извршава у једном паралелном кораку, јер нема конфликата.

12.13. Нека је $N[i]$ адреса $(i+1)$-ог елемента листе, односно показивач придружен i-том елементу, $1 \le i \le n$. У првом паралелном кораку (ако је $N[i] \ne 0$) процесор P_i проверава $F[N[i]]$: ако је 0, онда се "премошћује" $(i+1)$-ти елеменат помоћу $N[i] := N[N[i]]$, а помоћу $N[N[i]] := 0$ сигнализира процесору $P_{N[i]}$, да од следећег корака престане са радом, јер је његов елеменат искључен из листе. Тиме је преполовљена дужина сваког низа узастопних елемената листе са вредношћу $F[i]$ једнаком 0. Настављајући даље на исти начин, после $O(\log n)$ корака постиже се да за свако i, такво да је $F[i] = 1$, $N[i]$ показује на следећи необрисани елеменат листе (тј. $F[N[i]] = 1$). На крају се, ако је $F[1] = 0$, показивач на почетак листе преусмерава на $N[1]$.

12.14. Ако се сваком слогу придружи процесор, онда у једном паралелном кораку сваки процесор уписује индекс (адресу) свог слога у одговарајуће поље следећег слога у листи. После тога сваки процесор зна свог претходника у листи.

12.15. Користићемо индукцију. Ако је висина стабла (број слојева процесора) 2, онда леви лист шаље свој број корену, који га затим шаље наниже десном листу, а који га додаје свом броју x_2. Претпоставимо да имамо алгоритам за висину $h \ge 2$, и размотримо стабло висине $h+1$ (разматрамо само случај комплетних бинарних стабала; алгоритам се лако преправља да ради у општем случају). Нека је R корен, и нека су R_L и R_D његов леви и десни син. Суму свих листова подстабла зваћемо *сумом тог подстабла.* Најједноставније је решити проблем независно за лево и десно подстабло, а затим преко корена послати суму левог подстабла свим листовима у десном подстаблу. Сваки лист десног подстабла просто додаје суму левог подстабла израчунатом префиксу. Проблем са овим решењем је у томе, што му је време извршавања $O(h^2)$, због диференцне једначине $T(h+1) = T(h) + h$ (ако је $T(h)$ време извршавања на стаблу висине h). Ово решење може се поправити, ако запазимо да нема потребе чекати да лево стабло заврши израчунавање. Десно подстабло треба да добије суму левог подстабла, а та сума може бити расположива у корену у кораку $h+1$. Према томе, захтев који R_L треба да испуни је да прими суму свог подстабла и пошаље је навише ка корену. Захтев за R_D је да ту суму прими од корена и пошаље је наниже својим потомцима. Закључујемо да треба да важе следећа правила:

- сваки лист почиње слањем свог броја навише (треба дакле променити просто решење за стабло висине 2, јер корен треба да зна суму);
- унутрашњи чвор, кад добије вредности од синова, сабира их и шаље навише;
- унутрашњи чвор, кад добије вредност од оца, шаље ту вредност и левом и десном сину;
- унутрашњи чворови такође играју улогу корена својих подстабала, и вредност добијену од левог сина шаљу десном сину.

Препуштамо читаоцу доказ да је овај алгоритам коректан, и да му је време извршавања $2h$.

12.16. Једна итерација процеса селекције, као и у основној верзији, састоји се од четири фазе, тј. четири проласка стабла (горе,доле,горе,доле). У првој фази се бира пивот, који доспева у корен стабла. У другој фази се пивот доставља наниже свим (активним) листовима, исто као и у основној верзији. У трећој фази сваки лист упоређује свој број са пивотом и шаље навише резултат поређења, али овог пута прецизнији — да ли је већи од пивота, односно да ли је једнак пивоту. На путу до корена стабла се тако добија број v листова већих од пивота и број j листова једнаких пивоту. У четвртој фази се ова два броја достављају свим активним листовима. Тада сваки активни лист упоређује k са v и $v+j$, па ако је

- $k \leq v$, онда се лист искључује, ако је његов број мањи или једнак од пивота, а међу неискљученим листовима у наредној итерацији тражи се k-ти највећи,
- $v < k \leq v + j$, онда се лист искључује, ако је његов број мањи или већи од пивота, а међу неискљученим листовима у наредној итерацији тражи се $(k - v)$-ти највећи,
- $k > v + j$, онда се лист искључује, ако је његов број већи или једнак од пивота, а међу неискљученим листовима у наредној итерацији тражи се $(k - v - j)$-ти највећи.

12.17. У првом кораку процесор P_i израчунава производ $a_{ii}b_i$ и чува резултат; затим шаље b_i процесору P_{i-1}, $1 \leq i \leq n$. Уопште, у кораку k, $1 \leq k \leq n$, процесор P_i израчунава производ $a_{i,i+k-1}b_{i+k-1}$ и додаје га претходно израчунатом збиру, а затим шаље b_{i+k-1} процесору P_{i-1}, а од процесора P_{i+1} прима b_{i+k} (сви индекси рачунају се по модулу n). После n-тог корака процесор P_i садржи жељену суму $x_i = \sum_{k=0}^{n-1} a_{ik}b_k$, $0 \leq i \leq n-1$.

12.18. Пошто се b_i множи са елементима i-те колоне матрице, процесор P_i може да израчуна за n корака све производе $a_{ji}b_i$, $0 \leq j < n$. Остаје да се у наредних n корака паралелно израчунају збирови врста матрице.

Претпоставимо најпре да треба израчунати само x_0. Процесор P_1 израчунава $a_{01}b_1$, што је његов допринос вредности x_0, и шаље то процесору P_2; процесор P_2 тој вредности додаје $a_{02}b_2$ и прослеђује то процесору P_3, итд. После $n-1$ корака P_0 добија од P_{n-1} вредност $x_0 - a_{00}b_0$, па је у стању да заврши израчунавање. Да би се истовремено израчунавали сви x_i, потребно је обезбедити преклапање израчунавања (у току описаног израчунавања x_0 у сваком кораку је активан само један процесор!). У првом кораку сваки процесор P_i израчунава производ $a_{i-1,i}b_i$ (индекси се рачунају по модулу n), и шаље га процесору P_{i+1}. У кораку j процесор P_i прима вредност $\sum_{k=0}^{j-2} a_{i-j+1,i-k-1}b_{i-k-1}$, додаје јој $a_{i-j,i}b_i$, и шаље је процесору P_{i+1}.

12.19. Довољно је решити проблем за процесоре

$$P[n,i], P[n-1,i], \ldots, P[i+1,i], P[i,i], P[i,i+1], \ldots, P[i,n-1], P[i,n].$$

У овом низу од $2(n-i)+1$ процесора повезана су свака два суседна, а потребно је да размене своје елементе j-ти процесори са леве и десне стране од средњег ($P[i,i]$), $1 \leq j \leq n-i$. Овај посао може се поделити на два симетрична, који се извршавају један за другим: најпре леви процесори шаљу своје елементе десним, а онда (на исти начин) — десни процесори левим. Претпоставља се да сваки процесор има довољну меморију величине $O(1)$ (која не зависи од n). Да би леви процесори послали своје елементе десним, сви процесори у горе наведеном низу у $2(n-i)$ паралелних корака елеменат, добијен од левог суседа у претходном кораку, прослеђују десном суседу. Успут, процесор $P[i,i+j]$ трајно памти ("зауставља") елеменат добијен у кораку $2j$, $1 \leq j \leq n-i$, јер је управо то елеменат који је кренуо (у "каравану") из процесора $P[i+j,i]$.

12.20. За произвољну компоненту повезаности посматрајмо граф G чији су чворови њене тачке, а гране повезују свака два чвора који одговарају суседним (хоризонтално или вертикално) тачкама. Нека је v чвор G са најмањом ознаком, и нека је та ознака k. Индукцијом се доказује да после d корака сваки чвор на растојању од v мањем или једнаком од d, добија ознаку k (у кораку $d+1$ чвор на растојању $d+1$ од v, пошто је суседан неком чвору на растојању d од v, добија мању ознаку k). Према томе, после највише n^2 корака све тачке компоненте повезаности добиће ознаку k. Није тешко конструисати пример компоненте повезаности за коју означавање траје бар cn^2 корака (за неку константу c): то је нпр. "змијаста" трака састављена од непарних врста повезаних наизменично левом, односно десном црном тачком у парним врстама.

Поступак се може зауставити у тренутку кад се после извршења неког корака збир свих ознака не смањи. Међутим, израчунавање суме свих смањења ознака може се извести за $2n = O(n)$ паралелних корака. Једно могуће решење је зауставити поступак после n^2 итерација; тада је процес означавања сигурно завршен. Такође је могуће да се нпр. после сваких n итерација провери да ли се од итерације до итерације променила сума ознака. Тада,

ако су компоненте повезаности мањег дијаметра, извршавање поступка може да траје знатно краће.

12.21. Проблем се може решити директно, али је лакше свести га на множење матрица. Најпре се користи пермутација σ за формирање *пермутационе матрице* S, такве да у колони i једино елеменат у врсти $\sigma(i)$ има вредност 1, а сви остали елементи једнаки су нули. Ова матрица се може лако формирати, односно њени елементи се могу дистрибуирати одговарајућим процесорима за n корака. Пермутовање врста у складу са пермутацијом σ еквивалентно је израчунавању производа SA, за који знамо да се на оваквом рачунару може израчунати за $O(n)$ корака. Пермутовање колона изводи се на исти начин множењем здесна пермутационом матрицом Q.

12.22. У првом кораку се упоређују елементи у 2^{n-1} парова процесора

$$(0, a_1, a_2, \ldots, a_{n-1}), \quad (1, a_1, a_2, \ldots, a_{n-1}),$$

а већи од њих смешта се у процесор $(0, a_1, a_2, \ldots, a_{n-1})$. После тога се процесори са првом координатом 1 искључују. Даље се наставља на исти начин са 2^{n-1} података и $(n-1)$-димензионалном хиперкоцком. Уопште, у k-том кораку се упоређују елементи у процесорима

$$(0, 0, \ldots, 0, a_{k+1}, \ldots, a_{n-1}), (0, 0, \ldots, 1, a_{k+1}, \ldots, a_{n-1}),$$

а већи се смешта у процесор $(0, 0, \ldots, 0, a_{k+1}, \ldots, a_{n-1})$. После n корака се највећи елеменат налази у процесору $(0, 0, \ldots, 0)$. После k-тог корака раде само процесори код којих је првих k координата једнако нули, $1 \le k \le n$.

12.23. Индукцијом се проблему може приступити тако да се паралелни префикс решава истовремено у обе половине хиперкоцке (које су $(n-1)$-димензионалне хиперкоцке), а онда се највећи префикс ниже половине дистрибуира (са успутним сабирањем) паралелно свим процесорима више половине. Базни случај је тривијалан. Дистрибуција се у d-димензионалној хиперкоцки може извести у d паралелних корака, па време извршавања алгоритма задовољава диференцну једначину $T(d+1) = T(d) + d$, односно $T(d) = O(d^2)$. Алгоритам се може побољшати појачавањем индуктивне хипотезе. Претпоставимо да сваки процесор израчунава не само свој префикс, него и *суму свих елемената хиперкоцке.* Базни случај $d = 1$ је и даље једноставан. Дату $(d+1)$-димензионалну хиперкоцку делимо на две d-димензионалне хиперкоцке и индукцијом решавамо проблем на њима обема. Сада се, међутим, не мора дистрибуирати сума ниже половине хиперкоцке — ту суму знају сви њени елементи, па је одговарајућим елементима (суседима) у вишој половини хиперкоцке могу доставити у једном паралелном кораку. Сви префикси се могу израчунати у само једном допунском паралелном кораку. У наредном паралелном кораку се сума свих елемената израчунава истовремено у свим процесорима више половине хиперкоцке, а онда паралелно шаље у све суседне процесоре у нижој хиперкоцки. Нова диференцна једначина је према томе $T(d+1) = T(d) + 2$, одакле је $T(d) = O(d)$. Придруживање бројева процесорима јасно је из овог описа.

12.24. Потребно је елеменат (i, j), $0 \le i < 2^k$, $0 \le j < 2^m$, пресликати у неки елеменат хиперкоцке тако да му суседи буду елементи $(i \pm 1, j)$ и $(i, j \pm 1)$ (прва координата се рачуна по модулу 2^k, а друга по модулу 2^m). Ово се може постићи помоћу Грејових кодова $a_0, a_1, \ldots, a_{2^k-1}$ (речи са k бита, таквих да се a_i и a_{i+1} разликују на тачно једној битској позицији, видети одељак 1.7), односно $b_0, b_1, \ldots, b_{2^m-1}$ (m-битних речи са аналогном особином). Елеменат (i, j) смешта се у теме хиперкоцке са координатама (a_i, b_j), при чему је a_i први блок од k координата, а b_j други блок од m координата темена. Тиме се постиже да су слике суседних елемената у мрежи увек суседне у хиперкоцки.

12.25. Редослед комуницирања може се конструисати индукцијом. За $k = 1$ решење је тривијално. Ако се зна решење за $n = 2^k$ особа, онда се $2n = 2^{k+1}$ особа деле у две групе $\{1, 2, \ldots, n\}$ и $\{n+1, n+2, \ldots, 2n\}$, у оквиру којих се по индуктивној хипотези за k корака могу разменити сви трачеви. Затим у $(k+1)$-ом кораку размењују трачеве особа i са особом $i + n$, $1 \le i \le n$, после чега свих $2n$ особа знају све трачеве.

Српско-енглески и енглеско-српски терминолошки речник

Српско-енглески речник

алгоритам са јавним кључем - public key algorithm
алгоритам заснован на декомпозицији - divide-and-conquer algorithm

алтернирајући пут - alternating path
база чворова - vertex basis
базен - pool
бинарно стабло претраге - binary search tree
бипартитни граф - bipartite graph
блоковска шифра - block cipher
брат (дете истог оца) - sibling
циклус - circuit
чвор - node, vertex
датотека - file
дигитални потпис - digital signature
динамичко програмирање - dynamic programming
директна грана - forward edge
добротворни проблем - philantropist problem
долазна DFS **нумерација -** preorder DFS numbering
доминирајући скуп - dominating set
достижан (чвор) - reachable
дрво - tree
ефикасан алгоритам - efficient algorithm
ефикасност (паралелног алгоритма) - efficiency
електронска кодна књига - electronic codebook
фајл - file
фактор равнотеже - balance factor
FIFO **листа -** FIFO list
глава повезане листе - head of linked list
грана - edge
грана стабла - tree edge
гранање са одсецањем - branch-and-bound

Грејов код - Gray code
Хамилтонов пут - Hamiltonian path
хеш функција - hash function
хеуристика - heuristic
хип - heap
хипсорт - heap sort
индуковани подграф - vertex-induced subgraph
интервални граф - interval graph
излазна обрада - postWORK
излазни степен - outdegree
извор - source
јавни кључ - public key
једнодимензионална претрага опсега - one-dimensional range query

клика - clique
кључ - key
колизија - collision
коло - circuit
компајлер - compiler
компоненте повезаности - connected components
копија рекурзивне процедуре - instance
корак назад - backtrack
коренско стабло - rooted tree
крај гране - head
критична тежина - bottleneck weight
квиксорт - quicksort
ланчана шифра - stream cipher
лавински ефекат - avalanche efect
линеарно попуњавање - linear probbing
линија за простирање - broadcast line
лист - leaf
листа - queue
листа повезаности - adjacency list
листа са приоритетом - priority queue
максимално упаривање - maximal matching
матрица повезаности - adjacency matrix
машина са униформним приступом - random access machine
меморија са равномерним приступом - random access memory
минимално повезујуће стабло - minimum-cost spanning tree
модел са заједничком меморијом - shared memory model
мрежа - network
мрежа рачунара - interconnection network
најкраћи пут - shorthest path
најмањи заједнички надниз - smallest common supersequence

највећи заједнички подниз - largest common subsequence
нападач - attaker
неупарени чвор - unmatched vertex
неусмерени (граф) - undirected
неусмерени облик (графа) - undirected form
независан скуп - independent set
ниво - level
обилазак - traversing
обједињавање-раздвајање - merge-split
одлазна DFS **нумерација -** postorder DFS numbering
одржавати - to maintain
одвојено низање - separate chaining
опадајући први одговарајући - decreasing first fit
оптимално упаривање - maximum matching
отац - parent
отворени текст - cleartext, plaintext
петља - loop
пивот - pivot
подели и смрви - divide and crush
подграф - subgraph
похлепни метод - greedy method
показивач - pointer
покривање скупова - set-cover
покривач грана - vertex cover
покривач грана минималне тежине - minimum-weight vertex cover

понор - sink
попречна грана - cross edges
порука - message
потомак - descendant
повезана листа - linked list
повезани (граф) - connected
повезујућа шума - spanning forest
повезујуће стабло - spanning tree
повећавајући пут - augmenting path
повратна грана - backward edge, back edge
почетак гране - tail
пошиљалац - sender
права ослонца - supporting line
празан показивач - nil
празан слог - dummy record
предак - ancestor
преклапање израчунавања - pipelining
препознавање простих бројева - primality testing

пресек - cut
претходник - predecessor
претрага - backtracking
претрага са приоритетом - priority search
претрага у дубину - depth-first-search, DFS
претрага у ширину - breadth-first-search, BFS
преводилац - compiler
приближан алгоритам - approximation algorithm
прималац - receiver
пример улаза за проблем - instance
принцип имитирања паралелизма - parallelism folding principle
проблем паковања - bin packing problem
проблем придруживања - assignment problem
проблем ранца - knapsack problem
проблем трачева - gossip problem
проблем трговачког путника - traveling salesman problem
прости циклус - simple circuit
прости пут - simple path
простирање грешака - error extension
просторна сложеност - space complexity
први одговарајући - first fit
пут - path
ранг - rank
рачунарска графика - computational geometry
ред чекања - queue
рекорд - record
ретки граф - sparse graph
резидуални граф - residual graph
решив проблем - tractable problem
савршено упаривање - perfect matching
симетричан алгоритам - public key algorithm
син - child
систолично рачунање - systolic computation
скинути са стека - pop
скуп повратних грана - feedback-edge set
следбеника - successor
слек - slack
слободно стабло - free tree
слог - record
сложеност израчунавања - computational complexity
случајна шифра - one-time pad
сортирање директним вишеструким разврставањем - straight radix sort
сортирање избором - selection sort

сортирање обједињавањем - mergesort
сортирање обратним вишеструким разврставањем - radix exchange sort
сортирање парно-непарним обједињавањем - odd-even mergesort

сортирање парно-непарним транспозицијама - odd-even transposition sort
сортирање раздвајањем - quicksort
сортирање разврставањем - bucket sort
сортирање у месту - in-place sorting
сортирање уметањем - insertion sort
сортирање вишеструким разврставањем - radix sort
стабло - tree
стабло најкраћих путева - shortest path tree
стабло одлучивања - decision tree
стек - stack
степен - degree
стринг - string
сви најкраћи путеви - all-pairs shortest-paths
шифра са једнократним кључем - one-time pad
шифра замене - substitution cipher
шифрат - ciphertext
шифровање - encryption
шума - forest
тајни кључ - secret key
техника покретне линије - line-sweep techique
тежина - weight
тежински граф - weighted graph
ток - flow
тополошко сортирање - topological sort
транзитивно затворење - transitive closure
убрзање - speedup
уланчавање блокова шифрата - cipher block chaining
улазна обрада - preWORK
улазни степен - indegree
унутрашњи чвор - internal node
упаривање - matching
уписати на стек - push
уравнотежено стабло - balanced tree
уређено бинарно стабло - binary search tree
услов хипа - heap property
усмерени (граф) - directed
увијање поклона, алгоритам - gift wrapping
узорак - pattern

вектор - array
висина - height
враћање шифрата у шифровање - cipher feeback
временска сложеност - time complexity
врећа - pool
задовољивост - satisfiability
замена - swap
"завади па владај" алгоритам - divide-and-conquer algorithm

Енглеско-српски речник

adjacency list - листа повезаности
adjacency matrix - матрица повезаности
all-pairs shortest-paths - сви најкраћи путеви
alternating path - алтернирајући пут
ancestor - предак
approximation algorithm - приближан алгоритам
array - вектор
assignment problem - проблем придруживања
attaker - нападач
augmenting path - повећавајући пут
avalanche efect - лавински ефекат
back edge - повратна гране
backtrack - корак назад
backtracking - претрага
backward edge - повратна грана
balance factor - фактор равнотеже
balanced tree - уравнотежено стабло
BFS - претрага у ширину
bin packing problem - проблем паковања
binary search tree - бинарно стабло претраге, уређено бинарно стабло
bipartite graph - бипартитни граф
block cipher - блоковска шифра
bottleneck weight - критична тежина
branch-and-bound - гранање са одсецањем
breadth-first-search - претрага у ширину
broadcast line - линија за простирање
bucket sort - сортирање разврставањем
child - син
cipher block chaining - уланчавање блокова шифрата
cipher feeback - враћање шифрата у шифровање
ciphertext - шифрат
circuit - циклус, коло
cleartext - отворени текст

clique - клика
collision - колизија
compiler - компајлер, преводилац
computational complexity - сложеност израчунавања
computational geometry - рачунарска графика
connected - повезани (граф)
connected components - компоненте повезаности
cross edges - попречна грана
cut - пресек
decision tree - стабло одлучивања
decreasing first fit - опадајући први одговарајући
degree - степен
depth-first-search - претрага у дубину
descendant - потомак
DFS - претрага у дубину
digital signature - дигитални потпис
directed - усмерени (граф)
divide and crush - подели и смрви
divide-and-conquer algorithm - алгоритам заснован на декомпозицији, "завади па владај" алгоритам
dobrotvorni problem - пхилантропист проблем
dominating set - доминирајући скуп
dummy record - празан слог
dynamic programming - динамичко програмирање
edge - грана
efficiency - ефикасност (паралелног алгоритма)
efficient algorithm - ефикасан алгоритам
electronic codebook - електронска кодна књига
encryption - шифровање
error extension - простирање грешака
feedback-edge set - скуп повратних грана
FIFO **list** - FIFO листа
file - датотека, фајл
first fit - први одговарајући
flow - ток
forest - шума
forward edge - директна грана
free tree - слободно стабло
gift wrapping - увијање поклона, алгоритам
gossip problem - проблем трачева
Gray code - Грејов код
greedy method - похлепни метод
Hamiltonian path - Хамилтонов пут
hash function - хеш функција

head - крај гране
head of linked list - глава повезане листе
heap - хип
heap property - услов хипа
heap sort - хипсорт
height - висина
heuristic - хеуристика
in-place sorting - сортирање у месту
indegree - улазни степен
independent set - независан скуп
insertion sort - сортирање уметањем
instance - пример улаза за проблем, копија рекурзивне процедуре
interconnection network - мрежа рачунара
internal node - унутрашњи чвор
interval graph - интервални граф
key - кључ
knapsack problem - проблем ранца
largest common subsequence - највећи заједнички подниз
leaf - лист
level - ниво
line-sweep techique - техника покретне линије
linear probbing - линеарно попуњавање
linked list - повезана листа
loop - петља
matching - упаривање
maximal matching - максимално упаривање
maximum matching - оптимално упаривање
merge-split - обједињавање-раздвајање
mergesort - сортирање обједињавањем
message - поруку
minimum-cost spanning tree - минимално повезујуће стабло
minimum-weight vertex cover - покривач грана минималне тежине
network - мрежа
nil - празан показивач
node - чвор
odd-even mergesort - сортирање парно-непарним обједињавањем
odd-even transposition sort - сортирање парно-непарним транспозицијама
one-dimensional range query - једнодимензионална претрага опсега
one-time pad - шифра са једнократним кључем, случајна шифра
outdegree - излазни степен

parallelism folding principle - принцип имитирања паралелизма
parent - отац
path - пут
pattern - узорак
perfect matching - савршено упаривање
pipelining - преклапање израчунавања
pivot - пивот
plaintext - отворени текст
pointer - показивач
pool - базен, врећа
pop - скинути са стека
postorder DFS **numbering** - одлазна DFS нумерација
postWORK - излазна обрада
predecessor - претходник
preorder DFS **numbering** - долазна DFS нумерација
preWORK - улазна обрада
primality testing - препознавање простих бројева
priority queue - листа са приоритетом
priority search - претрага са приоритетом
public key - јавни кључ
public key algorithm - симетричан алгоритам, алгоритам са јавним кључем
push - уписати на стек
queue - ред чекања, листа
quicksort - сортирање раздвајањем, квиксорт
radix exchange sort - сортирање обратним вишеструким разврставањем
radix sort - сортирање вишеструким разврставањем
random access machine - машина са униформним приступом
random access memory - меморија са равномерним приступом
rank - ранг
reachable - достижан (чвор)
receiver - прималац
record - рекорд, слог
residual graph - резидуални граф
rooted tree - коренско стабло
satisfiability - задовољивост
secret key - тајни кључ
selection sort - сортирање избором
sender - пошиљалац
separate chaining - одвојено низање
set-cover - покривање скупова
shared memory model - модел са заједничком меморијом

shortest path tree - стабло најкраћих путева
shorthest path - најкраћи пут
sibling - брат (дете истог оца)
simple circuit - прости циклус
simple path - прости пут
sink - понор
slack - слек
smallest common supersequence - најмањи заједнички надниз
source - извор
space complexity - просторна сложеност
spanning forest - повезујућа шума
spanning tree - повезујуће стабло
sparse graph - ретки граф
speedup - убрзање
stack - стек
straight radix sort - сортирање директним вишеструким разврставањем
stream cipher - ланчана шифра
string - стринг
subgraph - подграф
substitution cipher - шифра замене
successor - следбеника
supporting line - права ослонца
swap - замена
systolic computation - систоличко рачунање
tail - почетак гране
time complexity - временска сложеност
to maintain - одржавати
topological sort - тополошко сортирање
tractable problem - решив проблем
transitive closure - транзитивно затворење
traveling salesman problem - проблем трговачког путника
traversing - обилазак
tree - дрво, стабло
tree edge - грана стабла
undirected - неусмерени (граф)
undirected form - неусмерени облик (графа)
unmatched vertex - неупарени чвор
vertex - чвор
vertex basis - база чворова
vertex cover - покривач грана
vertex-induced subgraph - индуковани подграф
weight - тежина

weighted graph - тежински граф

Литература

[1] U. Manber, *Introduction to algorithms, A creative approach*, Addison–Wesley, Reading, 1989.

[2] M. R. Garey, D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.

[3] D. E. Knuth, *The Art of Computer Programming*, 1–3, Addison-Wesley, Reading, 1973.

[4] J. Van Leeuwen, *Handbook of Theoretical Computer Computer Science. Vol A, Algorithms and Complexity*, Elsevier, Amsterdam, 1990.

[5] A. V. Aho, J. D. Ullman, *Computers Science, C Edition*, Computer Science Press, W. H. Freeman, New York, 1995.

[6] B. Schneier, *Applied Cryptography, Protocols, Algorithms and Source Code in C*, John Wiley & Sons, New York, 1994.

[7] R. L. Graham, D. E. Knuth, O. Patashnik, *Concrete Mathematics*, Addison-Wesley, Reading, 1994.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms* The MIT Press, Cambridge, 1991.

[9] N. Christofides, *Graph Theory, An Algorithmic Approach*, Academic Press, New York, 1975.

[10] E. M. Reingold, J. Nievergelt, N. Deo, *Combinatorial Algorithms, Theory and Practice*, Prentice-Hall, Englewood Cliffs, 1977.

[11] A. V. Aho, J. E. Hopcroft, J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, 1976.

[12] S. E. Goodman, S. T. Hedetniemi, *Introduction to the Design and Analysis of Algorithms*, McGraw-Hill, New York, 1977.

[13] E. Horowitz, S. Sahni, *Fundamentals of Computer ALgorithms*, Computer Science Press, Fall River Lane, 1978.

[14] S. G. Akl, *The Design and Analysis of Paralel Algorithms*, Prentice Hall, Englewood Cliffs, 1989.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, New York, 1991.

[16] Ђ. Пауниħ, *Структуре података и алгоритми*, Универзитет у Новом Саду, Нови Сад, 1997.