Глава 2

Сложеност израчунавања

Важно питање за практичну примену написаних програма је то колико ресурса програм захтева за своје извршавање. Најважнији ресурси су сигурно време потребно за извршавање програма и заузета меморија, мада се могу анализирати и други ресурси (на пример, код мобилних уређаја важан ресурс је утрошена енергија). Дакле, обично се разматрају:

- **временска** сложеност алгоритма;
- **просторна (меморијска)** сложеност алгоритма.

На пример, ако један програм израчунава потребан број за 10 секунди, а други за два и по минута, јасно је да је први програм практично примењивији. Међутим, ако први програм за своје извршавање захтева преко 10 гигабајта меморије, други око 1 гигабајт, а ми имамо рачунар са 4 гигабајта меморије, први програм нам је практично неупотребљив (иако ради много брже од другог). Ипак, с обзиром на то да савремени рачунарски системи имају прилично велику количину меморије, време је чешће ограничавајући фактор и у наставку ћемо се чешће бавити анализом временске ефикасности алгоритама.

При том, прилично је релативно колико брзо програм треба да ради да бисмо га сматрали ефикасним. На пример, ако програм успе да за пола сата реши неки нерешен математички проблем, који људи годинама нису могли да реше, он је свакако користан и можемо га сматрати веома ефикасним. Са друге стране, ако програм уграђен у аутомобил контролише кочнице приликом проклизавања, њему и неколико стотина милисекунди израчунавања може бити превише, јер ће за то време аутомобил неконтролисано слетети са пута.

Понашање програма (па и количина утрошених ресурса), наравно, зависи од његових улазних параметара. Јасно је, на пример, да ће програм брже израчунати просечну оцену двадесетак ученика једног одељења, него просечну оцену неколико десетина хиљада ученика који полажу Државну матуру. Такође се може претпоставити да понашање програма не зависи од конкретних оцена које су ученици добили, већ само од броја ученика. Зато сложеност алгоритма често изражавамо у функцији *величине (димензије) његових улазних параметара*, а не самих вредности параметара. Величина улазне вредности може бити број улазних елемената које треба обрадити, број битова потребних за записивање улаза који треба обрадити, сам улазни број који треба обрадити итд. Увек је потребно експлицитно навести у односу на коју величину улазне вредности се разматра сложеност.

Са друге стране, неки се алгоритми не извршавају исто за све улазе исте величине, па је потребно наћи начин за описивање ефикасности алгоритма на разним могућим улазима исте величине.

- **Анализа најгорег случаја** заснива процену сложености алгоритма на најгорем случају (на случају за који се алгоритам најдуже извршава – у анализи временске сложености, или на случају за који алгоритам користи највише меморије – у анализи просторне сложености). Та процена може да буде варљива, тј. превише песимистична. На пример, ако се програм у 99,9% случајева извршава испод секунде, док се само у 0,1% случајева извршава за око 10 секунди, анализом најгорег случаја закључили бисмо да ће се програм извршавати за око 10 секунди. Са друге стране, анализа најгорег случаја нам даје јаке гаранције да програм који је у најгорем случају довољно ефикасан у свим случајевима може да се изврши са расположивим ресурсима.
- У неким ситуацијама могуће је извршити **анализу просечног случаја** и израчунати просечно време извршавања алгоритма, али да би се то урадило, потребно је прецизно познавати простор допуштених

улазних вредности и вероватноћу да се свака допуштена улазна вредност појави на улазу програма. У случајевима када је битна гаранција ефикасности сваког појединачног извршавања програма процена просечног случаја може бити варљива, превише оптимистична, и може да се деси да у неким ситуацијама програм не може да се изврши са расположивим ресурсима. На пример, анализа просечног случаја би за претходни програм пријавила да се у просеку извршава испод једне секунде, међутим, за неке улазе он се може извршавати и преко десет секунди.

- Анализа најбољег случаја је, наравно, превише оптимистична и никада нема смисла.

Некада се анализа врши тако да се процени укупно време потребно да се изврши одређен број сродних операција. Тај облик анализе назива се **амортизована анализа** и у тим ситуацијама нам није битно време извршавања појединачних операција, већ само збирно време извршавања свих операција.

У наставку ће, ако није другачије речено другачије, бити подразумевана анализа најгорег случаја.

2.1 Мерење времена извршавања

Понашање за конкретне вредности улазних параметара се може експериментално одредити, тестирањем рада програма. На пример, размотримо наредна два алгоритма (дата у псеудокоду) који израчунавају збирове природних бројева од 1 до i, за свако i из интервала 1 до n.

```
zbir = 0
foreach i in [1, n]:
    zbir += i
    print(zbir)
```

и

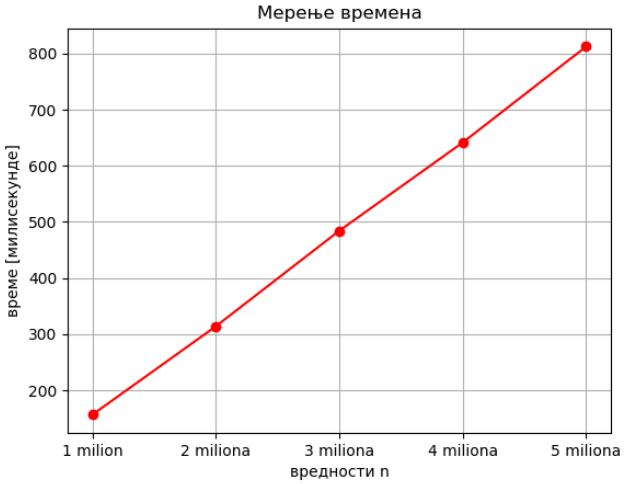
```
function zbir(k):
    zbir = 0
    foreach i in [1, k]:
        zbir += i
    return zbir

foreach k in [1, n]:
    print(zbir(k))
```

Ако се први алгоритам имплементира (на пример, у програмском језику Python) и ако се измери његово време извршавања за различите вредности n, добијају се следећи резултати (као резултат приказан је збир свих бројева од 1 до n).

```
n =  1000000, rezultat:   500000500000, vreme: 0.16 sekundi
n =  2000000, rezultat:  2000001000000, vreme: 0.31 sekundi
n =  3000000, rezultat:  4500001500000, vreme: 0.49 sekundi
n =  4000000, rezultat:  8000002000000, vreme: 0.65 sekundi
n =  5000000, rezultat: 12500002500000, vreme: 0.83 sekundi
```

Већ одавде се види да су времена приближно сразмерна вредностима n. Још прегледнији начин да уочимо ову пропорционалност је приказивање графика времена израчунавања суме у зависности од n, са кога се јасно види да код првог програма време извршавања приближно линеарно зависи од n.

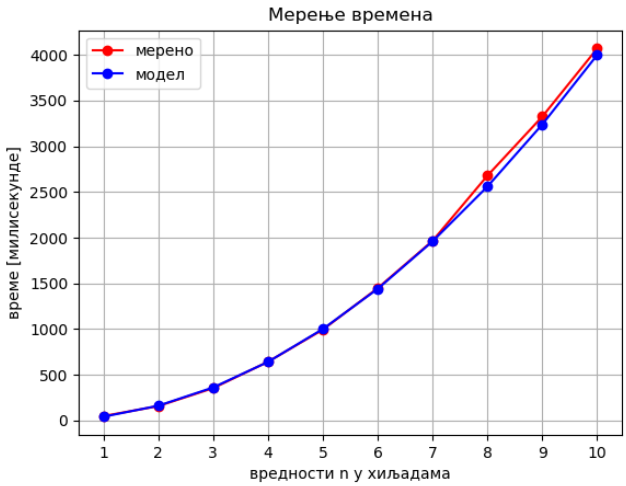


Слика 2.1: Линеарна зависност измереног времена

Ако се други алгоритам имплементира и ако се измери његово време извршавања (на пример, у програмском језику Python), добијају се следећи резултати (поново је приказан само збир свих бројева од 1 до n).

```
n =  1000, rezultat:   500500, vreme:    31.24 ms
n =  2000, rezultat:  2001000, vreme:   156.25 ms
n =  3000, rezultat:  4501500, vreme:   359.35 ms
n =  4000, rezultat:  8002000, vreme:   633.19 ms
n =  5000, rezultat: 12502500, vreme:   993.25 ms
n =  6000, rezultat: 18003000, vreme:  1448.38 ms
n =  7000, rezultat: 24503500, vreme:  1984.94 ms
n =  8000, rezultat: 32004000, vreme:  2547.53 ms
n =  9000, rezultat: 40504500, vreme:  3291.45 ms
n = 10000, rezultat: 50005000, vreme:  4049.11 ms
```

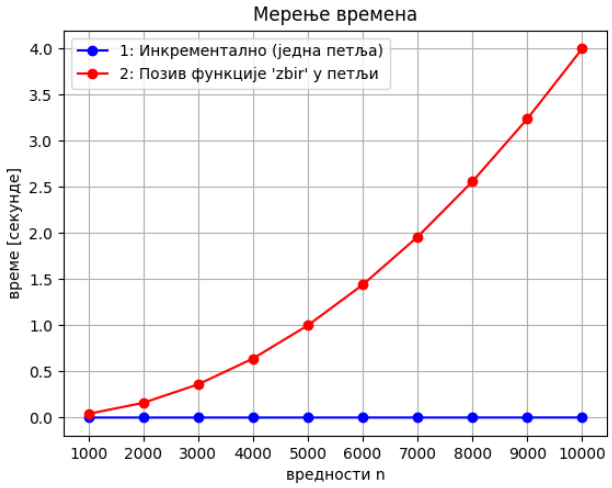
У овом случају зависност није линеарна. Наиме, када се n удвостручи, време се уместо 2 пута, отрпилике увећа 4 пута, што сугерише да је време извршавања сразмерно са квадратом величине улаза, тј. да је у питању квадратна зависност. Рачунањем $\frac{t(n)}{n^2}$ за разне n добијамо приближно сталну вредност 0.00004, што значи да се време извршавања $t(n)$ може апроксимирати као $t(n) \approx 0.00004n^2$. На следећој слици видимо график измерених вредности и график вредности добијене квадратне функције која моделира време извршавања.



Слика 2.2: Квадратна зависност измереног времена

Два дата графика се у значајној мери поклапају, што показује да је $0.00004n^2$ веома добра процена времена извршавања алгоритма за разне вредности n. Наравно, време извршвања увек зависи од програмског језика и конкретног рачунара на ком је мерење извршено, али експеримент говори да можемо очекивати да ће време рада другог алгоритма увек бити прилижно квадратна функција величине улаза. Ово се, наравно, може и формално доказати анализом броја извршених инструкција.

На основу измерених времена видимо да је први програм неупоредиво бржи у односу на други. Линеарно време извршавања је толико мање од квадратног, да када се оба времена прикажу на истом графику, делује да је линеарно време стално једнако нули.

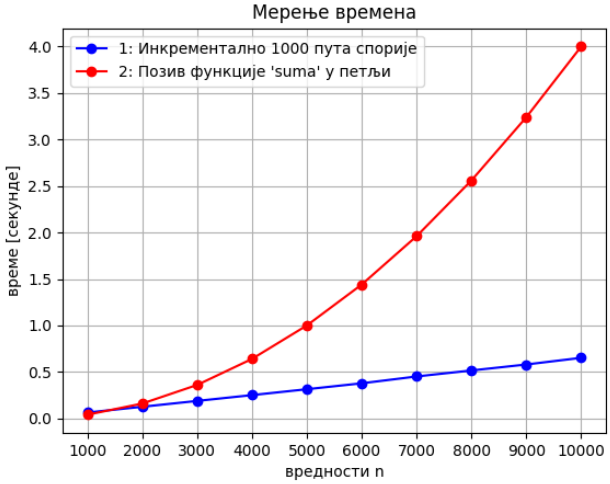


Слика 2.3: Однос линеарног и квадратног времена

2.2 Асимптотска анализа сложености

Мерење времена извршавања се може урадити за сваки програм, али оно није поуздан аргумент само за себе, већ служи пре свега за почетно стицање осећаја о ефикасности појединих алгоритама (да знамо шта треба да докажемо), као и за потврду закључака добијених теоријским разматрањем. Осим тога, често нам је потребно да можемо да унапред дамо неку грубу процену потребних ресурса за произвољне улазне вредности, без покретања програма (па чак и пре писања програма, само на основу алгоритма који ће бити примењен).

Питање које се природно поставља је то на ком ће се рачунару програм извршавати. Наравно, ако је један рачунар два пута бржи (у неком сегменту) од другог, за очекивати је да ће се програм на њему извршавати два пута брже. Ипак, показаће се да су разлике између ефикасних и неефикасних алгоритама толико велике, да је то што је неки рачунар 2, 3 или чак 10 пута бржи од другог заправо небитно и не може да надомести то колико је неефикасан алгоритам лошији од ефикасног. На пример, наредна слика приказује како би изгледао однос времена извршавања ако би се бржи алгоритам извршавао на 1000 пута споријем рачунару. Као што видимо, линеарна функција се мало "одлепила" од нуле, међутим, и даље су њене вредности много мање него код квадратне функције. Другим речима, алгоритам линеарне сложености, чак и када се успори 1000 пута, и даље је много бржи од алгоритма квадратне сложености (и што је n веће, разлика у брзини је све већа).



Слика 2.4: Однос линеарног и квадратног времена на рачунарима различите брзине

Да бисмо проценили зависност времена извршавања од димензије проблема, основни приступ је да покушамо да конструишемо функцију $f(n)$ која одређује зависност броја инструкција које алгоритам треба да изврши у односу на величину улаза n.

Израчунајмо број наредби сабирања које се изврше у првом и у другом програму из претходног примера (програми извршавају и друге наредбе, попут оних потребних да се организују петље, међутим, претпоставићемо да су нам сабирања једино значајна).

Јасно је да се у првом програму врши једно сабирање по петљи, па је укупан број сабирања једнак броју корака извршавања петље, а то је n. Ово је линеарна зависност, што је у складу са измереним временима.

У другом програму анализа је мало компликованија. За било које дато k, функција `zbir` врши око k сабирања. Пошто се функција позива за све вредности k од 1 до n, то се укупно изврши $1 + 2 + \ldots + n = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$ сабирања. И ако бисмо рачунали све инструкције, добили бисмо да је број инструкција нека квадратна функција облика $an^2 + bn + c$, што је опет у складу са измереним временима.

Ако (поједностављено) претпоставимо да се свака инструкција на рачунару извршава за једну наносекунду ($10^{-9}s$), а да број инструкција зависи од величине улаза n на основу функције $f(n)$, тада је време потребно да се алгоритам изврши дат у следећим табелама.

Алгоритми чија је сложеност одозго ограничена полиномијалним функцијама, у принципу се сматрају ефикасним.

$n/f(n)$	$\log n$	$\sqrt{n}$	n	$n \log n$	n^2	n^3
10	0,003 μs	0,003 μs	0,01 μs	0,033 μs	0,1 μs	1 μs
100	0,007 μs	0,010 μs	0,1 μs	0,644 μs	10 μs	1 ms
1,000	0,010 μs	0,032 μs	1,0 μs	9,966 μs	1 ms	1 s
10,000	0,013 μs	0,1 μs	10 μs	130 μs	0,1 s	16,7 min
100,000	0,017 μs	0,316 μs	100 μs	1,67 ms	10 s	11,57 dan
1,000,000	0,020 μs	1 μs	1 ms	19,93 ms	16,7 min	31,7 god
10,000,000	0,023 μs	3,16 μs	10 ms	0,23 s	1,16 dan	3×10^5 god
100,000,000	0,027 μs	10 μs	0,1 s	2,66 s	115,7 dan	
1,000,000,000	0,030 μs	31,62 μs	1 s	29,9 s	31,7 god	

Алгоритми чија је сложеност одоздо ограничена експоненцијалном или факторијелском функцијом се сматрају неефикасним.

$n/f(n)$	2^n	$n!$
10	1 μs	3,63 ms
20	1 ms	77,1 god
30	1 s	$8,4 \times 10^{15}$ god
40	18,3 min	
50	13 dan	
100	4×10^{13} god	

Можемо поставити и питање која димензија улаза се отприлике може обрадити за одређено време. Одговор је дат у наредној табели.

t	n	$n \log n$	n^2	n^3	2^n	$n!$
$1ms$	10^6	$63,000$	$1,000$	100	20	9
$10ms$	$10 \cdot 10^6$	$530,000$	$3,200$	215	23	10
$100ms$	$100 \cdot 10^6$	$4,5 \cdot 10^6$	$10,000$	465	27	11
$1s$	10^9	$40 \cdot 10^6$	$32,000$	$1,000$	30	12
$1min$	$60 \cdot 10^9$	$1,9 \cdot 10^9$	$245,000$	$3,900$	36	14

Из претходних табела јасно је да време извршавања суштински зависи од функције $f(n)$. На пример, ако поредимо алгоритме код којих је $f_1(n) = n$, $f_2(n) = 5n$, $f_3(n) = n^2$ и $f_4(n) = 2n^2 + 3n + 2$, јасно нам је да ће се за $n = 10^6$, време извршавања првог и другог алгоритма мерити милисекундама, док ће се време извршавања трећег и четвртог алгоритма мерити минутима. Код функције f_4, јасно је да је време које потиче од фактора $3n$ (три милисекунде) и 2 (две наносекунде) апсолутно занемариво у односу на време које долази од фактора $2n^2$ (око 33 минута). За прва два алгоритма рећи ћемо да имају *линеарну временску сложеност*, а за друга два да имају *квадратну временску сложеност*.

Чак ни педесет пута бржи рачунар неће помоћи да се трећи или четврти алгоритам изврше брже од првог или другог. Иако смо поједностављено претпоставили да се све инструкције извршавају исто време (једну наносекунду), што није случај у реалности, из претходних табела је јасно да нам тај поједностављени модел даје сасвим добру основу за поређење различитих алгоритама и да прецизнија анализа не би ни по чему значајно променила ситуацију. Сложеност се обично процењује на основу изворног кода програма. Савремени компилатори извршавају различите напредне оптимизације и машински кôд који се извршава може бити прилично другачији од изворног кода програма (на пример, компилатор може скупу операцију множења заменити ефикаснијим битовским операцијама, може наредбу која се више пута извршава у петљи изместити ван петље и слично). Детаљи који се у изворном коду не виде, попут питања да ли се неки податак налази у кеш-меморији или је потребно приступати РАМ-у, такође могу веома значајно да утичу на стварно време извршавања програма. Савремени процесори подржавају проточну обраду и паралелно извршавање инструкција, што такође чини стварно понашање програма другачијим од класичног, секвенцијалног модела који се најчешће подразумева приликом анализе алгоритама. Дакле, стварно време извршавања програма зависи од карактеристика конкретног рачунара на ком се програм извршава, али и од карактеристика програмског преводиоца, па и оперативног система на ком се програм извршава. Међутим, поново наглашавамо да ништа од тих фактора не може променити однос између времена извршавања алгоритама линеарне и алгоритама квадратне сложености, за велике улазе (код малих улаза, сви алгоритми раде веома ефикасно, па нам обрада малих улаза није интересантна).

Дакле, можемо закључити да нам за је за грубу процену времена потребног за извршавање неког алгоритма, чији број инструкција полиномијално зависи од величине улаза n довољно да знамо само који је степен тог полинома. Можемо слободно да занемаримо све мономе мањег степена, а можемо и слободно да занемаримо коефицијенте уз водећи степен, као и коефицијент којим се одређује брзина стварног рачунара у односу на овај фиктивни, за који смо приказали времена. Наиме у реалним ситуацијама сви ти коефицијенти могу да утичу да ће програм бити бржи или спорији највише десетак пута (па нек је и стотинак пута), али не могу да утичу на то да се за велики улаз алгоритам чији је број инструкција квадратни изврши брже од алгоритма чији је број инструкција линеаран (говоримо о односу минута и милисекунди).

Горња граница сложености се обично изражава коришћењем O-нотације.

Дефиниција: Ако постоје позитивна реална константа c и природан број n_0 такви да за функције f и g над

природним бројевима важи $f(n) \leq c \cdot g(n)$ за све природне бројеве n веће од n_0 онда пишемо $f(n) = O(g(n))$ и читамо "f је велико ‚о' од g".

У неким случајевима користимо и ознаку Θ која нам не даје само горњу границу, већ прецизно описује асимптотско понашање.

Дефиниција: Ако постоје позитивне реалне константе c_1 и c_2 и природан број n_0 такви да за функције f и g над природним бројевима важи $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ за све природне бројеве n веће од n_0, онда пишемо $f(n) = \Theta(g(n))$ и читамо „f је велико 'тета' од g".

Дакле, асимптотским ознакама смо занемарили мономе мањег степена и сакрили константе уз највећи степен полинома. Стварно време извршавања зависи и од константи сакривених у асимптотским ознакама, међутим, асимптотско понашање обично прилично добро одређује његов ред величине (да ли су у питању микросекунде, милисекунде, секунде, минути, сати, дани, године).

Наведимо карактеристике основних класа сложености.

- $O(1)$ – *константна сложеност*, алгоритми линијско-разгранате структуре који се извршавају практично моментално, нпр. алгоритми у којима се имплементира нека математичка формула;
- $O(\log n)$ – *логаритамска сложеност*, изузетно ефикасно, нпр. бинарна претрага;
- $O(\sqrt{n})$ – *коренска сложеност*, "логаритам за оне са јефтинијим улазницама" - немамо најбоља места, али ипак можемо да гледамо утакмицу, нпр. испитивање да ли је број прост, факторизација броја на просте чиниоце;
- $O(n)$ – линеарна сложеност, оптимално, када је за решење потребно погледати цео улаз, нпр. минимум/максимум серије елемената;
- $O(n \log n)$ – *квазилинеарна сложеност*, "линеарни алгоритам за оне са јефтинијим улазницама", ефикасни алгоритми засновани на декомпозицији (нпр. сортирање обједињавањем), ефикасном сортирању, коришћењу структура података са логаритамским временом приступа;
- $O(n^2)$ – *квадратна сложеност*, обично (али не обавезно) угнежђене петље, нпр. сортирање селекцијом, сортирање уметањем;
- $O(n^3)$ – *кубна сложеност*, обично (али не обавезно) вишеструко угнежђене петље, нпр. множење матрица;
- $O(2^n)$ – *експоненцијална сложеност*, изузетно неефикасно, нпр. испитивање свих подскупова;
- $O(n!)$ – *факторијелна сложеност*, изузетно неефикасно, нпр. испитивање свих пермутација.

Иако су класе поређане редом, не треба претпостављати да су оне "подједнако размакнуте". На пример, честа је заблуда да су алгоритми сложености $O(n \log n)$ по својој ефикасности негде између $O(n)$ и $O(n^2)$. Истина је заправо да су они прилично слични класи $O(n)$ и да је често тешко емпријски измерити разлику између те две класе, а да су и једни и други неупоредиво бржи од алгоритама квадратне сложености. На пример, ако је $n = 10^6$, веома груба процена (када се занемаре сви константни коефицијенти) броја корака линеарног алгоритма је 10^6, квазилинеарног алгоритма је око $20 \cdot 10^6$, а број корака квадратног алгоритма је 10^{12}. Дакле, на тако великом улазу квазилинеарни алгоритам би био тек пар десетина пута спорији од линеарног (конкретан однос, наравно, зависио би од констаних фактора), док би квадратни алгоритам био милион пута спорији од линеарног и неких педесет хиљада пута спорији од квазилинеарног.

2.3 Математичке основе

Да бисмо могли да анализирамо сложеност потребно је да владамо одређеним математичким апаратом. У наставку ћемо резимирати основне математичке појмове које ћемо користити у анализи сложености алгоритама.

2.3.1 Сумирање

Током анализе алгоритама често имамо потребу да израчунамо одређене коначне суме. Са њима сте се сретали у средњој школи и курсевима дискретне математике. Резимирајмо их кроз неколико најзначајнијих примера.

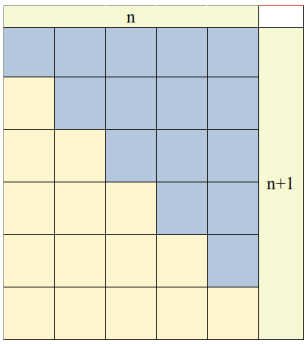
2.3.1.1 Аритметички низ

Гаусу се приписује да је још као дете израчунао да је

$$1 + 2 + \ldots + n = \sum_{k=0}^{n} k = \frac{n(n+1)}{2}.$$

Заиста, у овом збиру се крије $n/2$ парова чији је збир $n+1$ (ово, наравно, важи само када је n паран број, али нам даје одличну интуицију која нам помаже да ову формулу лако запамтимо). Прецизније, ако означимо тај збир са S онда је $2S = S + S = (1 + 2 + \ldots + n) + (n + (n-1) + \ldots + 1) = n \cdot (n+1)$.

Некада слика говори више од речи.



Слика 2.5: Гаусова формула

На основу претходног једноставно се изводи да је збир првих n чланова аритметичког низа чији је први члан a, а разлика између свака два суседна члана једнака r једнака

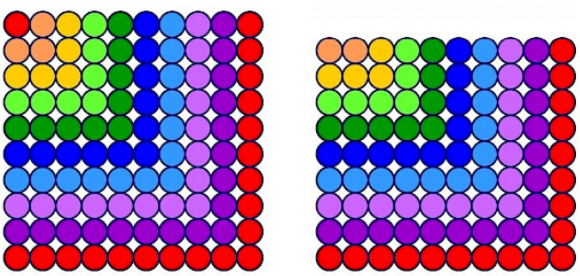
$$a + (a + r) + (a + 2r) + \ldots + (a + (n-1) \cdot r) = \sum_{k=0}^{n-1} (a + k \cdot r) = n \cdot a + r\frac{n(n-1)}{2}.$$

Интуиција нам опет говори да се овде крије $\frac{n}{2}$ парова чији је збир $a_0 + a_{n-1}$, што опет доводи до формуле $\frac{n}{2}(2a + (n-1) \cdot r)$.

Један важан аритметички низ је низ непарних бројева. Важи да је $1+3+5+\ldots+(2k-1) = \frac{k}{2} \cdot (1+(2k-1)) = k^2$.

Израчунавање збира узастопних парних бројева се лако своди на збир узастопних бројева. $2+4+6+\ldots+2k = 2(1 + 2 + 3 + \ldots + k) = k(k+1)$.

Поново слика говори више од речи.



Слика 2.6: Збир непарних бројева од 1 до $2k-1$ и збир парних бројева од 2 до $2k$

2.3.1.2 Геометријски низ и ред

Изведимо формулу за збир првих n чланова геометријског низа коме је први члан a а количник свака два узастопна члана $q \neq 1$. Обележимо тражену суму са S.

$$S = a + a \cdot q^1 + a \cdot q^2 + \ldots + a \cdot q^{n-2} + a \cdot q^{n-1} = \sum_{k=0}^{n-1} a \cdot q^k.$$

Ако леву и десну страну претходне једнакости помножимо са $1 - q$ добијамо једнакост:

$S \cdot (1 - q) = a \cdot (1 - q) + a \cdot q \cdot (1 - q) + a \cdot q^2 \cdot (1 - q) + \ldots + a \cdot q^{n-2} \cdot (1 - q) + a \cdot q^{n-1} \cdot (1 - q)$

Извршимо множења на десној страни једнакости:

$$S \cdot (1 - q) = a - a \cdot q + a \cdot q - a \cdot q^2 + a \cdot q^2 - a \cdot q^3 + \ldots + a \cdot q^{n-2} - a \cdot q^{n-1} + a \cdot q^{n-1} - a \cdot q^n$$

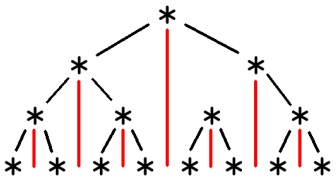
Сређивањем последњег израза добијамо $S \cdot (1 - q) = a - a \cdot q^n$. Према томе, пошто је $q \neq 1$, важи

$$S = a \cdot \frac{1 - q^n}{1 - q}.$$

За $|q| < 1$ геометријски ред конвергира и сума му је $\frac{a}{1-q}$.

Нама ће најчешће бити корисни случајеви $q = 2$ и $q = 1/2$.

На основу претходне формуле, за $a = 1$ и $q = 2$, важи да је $1 + 2 + \ldots + 2^{n-1} = 2^n - 1$. Ова формула има интересантно тумачење. Сума са леве стране представља укупан број чворова на првих n нивоа потпуног бинарног дрвета, док је израз са десне стране за један мањи од броја чворова на наредном нивоу $n+1$. Дакле, на сваком наредном нивоу бинарног дрвета има један чвор више него што је чворова на свим претходним нивоима дрвета.



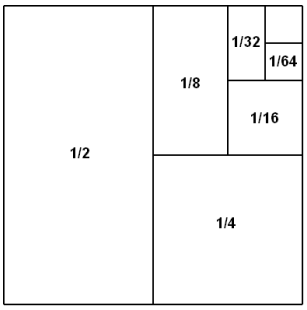
Слика 2.7: Број чворова на најнижем нивоу бинарног дрвета за један је већи од укупног броја чворова на претходним нивоима

За $a = 1$ и $q = 1/2$ добијамо да је

$$1 + 1/2 + \ldots + (1/2)^{n-1} = \frac{1 - (1/2)^n}{1 - 1/2} = 2 - (1/2)^{n-1}.$$

Са порастом n ова вредност се приближава вредности 2 (свакако је њоме ограничена одозго).

Опет слика говори више од речи.



Слика 2.8: Збир геометријског реда за $a = 1/2$, $q = 1/2$

2.3.1.3 Степене суме

Прикажимо како можемо израчунати суму квадрата свих природних бројева од 1 до n. Важи да је

$$(k+1)^3 - k^3 = k^3 + 3k^2 + 3k + 1 - k^3 = 3k^2 + 3k + 1.$$

Зато је

$$\begin{aligned} 2^3 - 1^3 &= 3 \cdot 1^2 + 3 \cdot 1 + 1 \\ 3^3 - 2^3 &= 3 \cdot 2^2 + 3 \cdot 2 + 1 \\ &\dots \\ (n+1)^3 - n^3 &= 3 \cdot n^2 + 3 \cdot n + 1 \end{aligned}$$

Сабирањем претходних једнакости добијамо

$$(n+1)^3 - 1 = 3 \cdot (1^2 + \ldots + n^2) + 3 \cdot (1 + \ldots + n) + (1 + \ldots + 1)$$

тј.

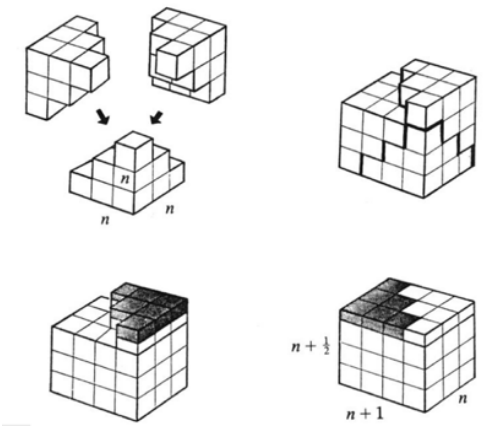
$$3 \cdot \sum_{k=1}^{n} k^2 = (n+1)^3 - 1 - 3\sum_{k=1}^{n} k - \sum_{k=1}^{n} 1.$$

На основу раније изведених формула за збир аритметичког низа, следи да је

$$\sum_{k=1}^{n} k^2 = \frac{1}{3} \cdot \left(n^3 + 3n^2 + 3n - 3\frac{n(n+1)}{2} - n \right) = \frac{n \cdot (2n+1) \cdot (n+1)}{6} = \frac{1}{3}\left(n \cdot (n + \frac{1}{2}) \cdot (n+1) \right).$$

Дакле, сума квадрата природних бројева од 1 до n се асимптотски понаша као $\frac{n^3}{3}$.

Опет слика говори више од речи.



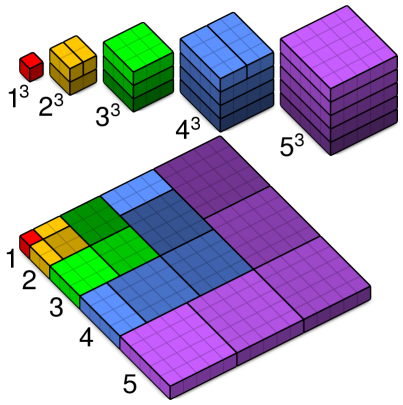
Слика 2.9: Збир квадрата свих природних бројева од 1 до n

Потпуно аналогно, сумирањем израза $(k+1)^4 - k^4$ од 1 до n и применом до сада изведених формула може се показати да је

$$1^3 + 2^3 + \ldots + n^3 = \sum_{k=1}^{n} k^3 = \frac{(n(n+1))^2}{4}.$$

Ово тврђење, познато као Никомахова теорема показује да је збир кубова свих природних бројева од 1 до n једнак квадрату збира свих природних бројева од 1 до n и асимптотски се понаша као $\frac{n^4}{4}$.

Опет слика говори више од речи.

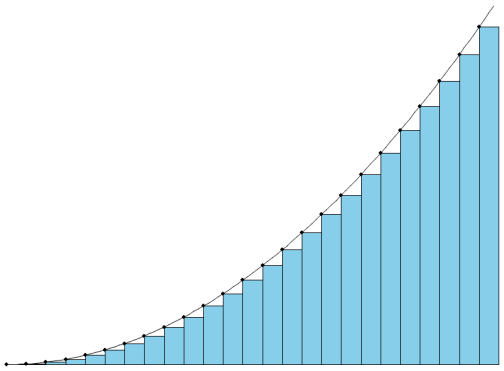


Слика 2.10: Збир кубова свих природних бројева од 1 до n

Пошто ће нас у анализи алгоритама најчешће занимати само асимптотско понашање функција, најважније је запамтити да се сума n p-тих степена свих природних бројева од 1 до n асимптотски понаша као $\frac{n^{p+1}}{p+1}$.

2.3.1.4 Примена диференцијалног и интегралног рачуна у израчунавању и оцени сума

За израчунавање и оцену сума могу се користити и изводи и интеграли. Приметимо да је неодређени интеграл функције x^p функција $\frac{x^{p+1}}{p+1}$, а да се збир p-тих степена свих природних бројева од 1 до n асимптотски понаша управо као $\frac{n^{p+1}}{p+1}$. То није случајност. Размотримо монотоно растућу функцију f (таква је функција $f(x) = x^n$) на домену $x \geq 0$. Сума $\sum_{k=0}^{n-1} f(k)$ се визуелно може представити као површина n правоугаоника (свакоме је ширина 1, а висина k-тог је $f(k)$). На слици је приказана сума првих 25 потпуних квадрата. Са слике је прилично очигледно да је та сума (збир површина правоугаоника) веома блиска површини испод криве $f(x) = x^2$ која се може израчунати (тј. чије се асимптотско понашање може проценити) применом одређених интеграла.



Слика 2.11: Процена суме одређеним интегралом

Илуструјмо ово и мало прецизније. Површина испод криве $f(x)$ за $k \leq x \leq k+1$ једнака је одређеном интегралу $\int_k^{k+1} f(x)\,dx$. Пошто је функција растућа, та површина је већа од површине правоугаоника чија је висина $f(k)$ и ширина 1, а мања од површине правоугаоника чија је висина $f(k+1)$ и ширина 1 тј. важи

$$f(k) \leq \int_k^{k+1} f(x)\,dx \leq f(k+1).$$

Зато је

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

Горњу границу суме можемо директно прочитати из прве неједнакости. Пошто је

$$\sum_{k=0}^{n-1} f(k+1) = \sum_{k=0}^{n-1} f(k) - f(0) + f(n),$$

из друге неједнакости следи и доња граница, па важи:

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

Случај када је $f(x)$ монотоно опадајућа функција за $x \geq 0$ се обрађује аналогно (само је потребно уместо $\leq$ користити $\geq$).

На пример, понашање суме $\sum_{k=0}^{n-1} ka^k = a+2a^2+3a^3+\ldots+(n-1)a^{n-1}$, можемо проценити израчунавањем одређеног интеграла $\int_0^n xa^x\,dx$.

Он се једноставно може израчунати парцијалном интеграцијом за $u = x$ (па је $du = dx$) и $dv = a^x\,dx$, одакле је $v = \frac{a^x}{\ln a}$. Зато је

$$\int_0^n xa^x\,dx = \int_0^n u\,dv = (uv)\big|_0^n - \int_0^n v\,du = \frac{na^n}{\ln a} - \frac{\int_0^n a^x\,dx}{\ln a} = \frac{na^n}{\ln a} - \frac{(a^n-1)}{\ln^2 a},$$

па се ова функција асимптотски понаша као na^n.

Ову суму је могуће израчунати и егзактно, применом диференцирања. Наиме, важи да је

$$\sum_{k=0}^{n-1} x^k = 1 + x + \ldots + x^{n-1} = \frac{x^n - 1}{x - 1}.$$

Диференцирањем обе стране по x добијамо

$$1 + 2x + 3x^2 + \ldots + (n-1)x^{n-2} = \frac{nx^{n-1}(x-1) - (x^n - 1)}{(x-1)^2}.$$

Множењем са x добијамо

$$\sum_{k=0}^{n-1} kx^k = x + 2x^2 + 3x^3 + \ldots + (n-1)x^{n-1} = x\frac{nx^{n-1}(x-1) - (x^n - 1)}{(x-1)^2}.$$

На пример, за $x = 2$ добијамо да је $\sum_{k=0}^{n-1} k2^k = (n-2)\cdot 2^n + 2$.

Напоменимо и да се та сума веома једноставно може израчунати и сасвим елементарним техникама. Нека је $S_x(n) = \sum_{k=0}^{n-1} kx^k$. Тада множењем ове једнакости са x, одузимањем добијеног резултата од полазне једнакости и применом формуле за збир геометријског низа добијамо

$$\begin{aligned}
S_x(n) &= x^1 + 2x^2 + 3x^3 + \ldots + (n-2)x^{n-2} + (n-1)x^{n-1} \\
x \cdot S_x(n) &= x^2 + 2x^3 + 3x^4 + \ldots + (n-2)x^{n-1} + (n-1)x^n \\
S_x(n) - xS_x(n) &= x^1 + x^2 + x^3 + \ldots + x^{n-1} + (n-1)x^n \\
(1-x)S_x(n) &= x(x^0 + \ldots + x^{n-2}) + (n-1)x^n = x\frac{1-x^{n-1}}{1-x} + (n-1)x^n
\end{aligned}$$

Одатле следи да је

$$S_x(n) = \frac{x - x^n}{(1-x)^2} + (n-1)x^n$$

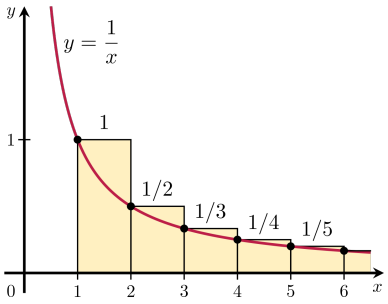
Иако није добијен идентичан израз ономе који је добијен применом диференцирања, лако се показује да су добијени изрази једнаки. Заменом вредности $x = 2$ поново добијамо да је $S_2(n) = \sum_{k=0}^{n-1} k2^k = (n-2) \cdot 2^n + 2$.

2.3.1.5 Хармонијски ред

Размотримо збир $H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n}$. Њега не можемо егзактно израчунати, али можемо га прилично добро приближно проценити.

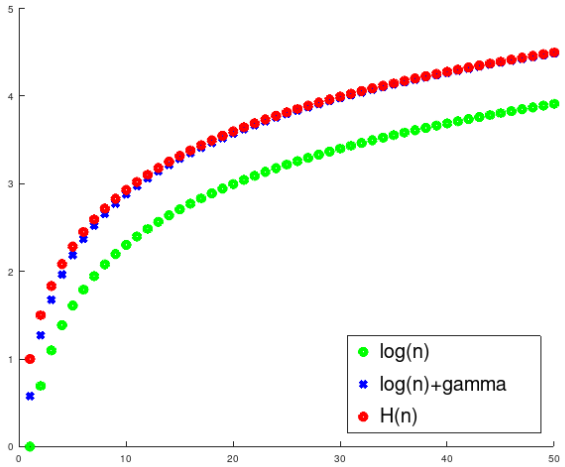
Прво покажимо да $H(n)$ дивергира и да тежи бесконачности како n тежи бесконачности. Важи да је $H(n) > 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \ldots$ Наиме, важи да је $\frac{1}{3} > \frac{1}{4}$, затим да је $\frac{1}{5} > \frac{1}{8}$, $\frac{1}{6} > \frac{1}{8}$ и $\frac{1}{7} > \frac{1}{8}$ итд. Међутим, важи да је $\frac{1}{4} + \frac{1}{4} = \frac{1}{2}$, да је $\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} = \frac{1}{2}$ итд. Зато је десни збир једнак $1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \ldots$, а овај збир јасно дивергира тј. тежи бесконачности са повећањем броја сабирака.

Слично би се могло доказати и коришћењем процене суме одређеним интегралом. Број $H(n)$ се може проценити као $\int_1^{n+1} \frac{1}{x} \mathrm{d}x$, који је једнак $(\ln x)\big|_1^{n+1}$ тј. $\ln(n+1)$.



Слика 2.12: Процена збира хармонијског реда $H(n)$ одређеним интегралом

Ако се нацрта график функције $H(n)$ може се уочити да она веома споро тежи бесконачности и може се приметити веома сличан облик графику функције $\ln n$. Заиста, доказује се да разлика између функција $H(n)$ и $\ln n$ веома брзо конвергира и тежи константи γ која је позната под именом Ојлер-Маскеронијева константа и чија је вредност приближно једнака $\gamma = 0.57722$. Ништа се не би променило и да се посматра однос $H(n)$ са функцијом $\ln(n+1)$ (јер са повећањем n разлика између $\ln(n+1)$ и $\ln n$ веома брзо тежи нули), осим што би процена била мало прецизнија за мале вредности n.



Слика 2.13: Збир хармонијског реда $H(n)$ и однос са логаритамском функцијом $\ln n$

2.4 Сложеност неких честих облика петљи

Прикажимо сада кроз неколико примера анализу сложености итеративно имплементираних алгоритама. Иако нећемо посматрати решења конкретних задатака, потрудићемо се да у примерима покријемо облике петљи који се јављају у великом броју конкретних алгоритама и решења конкретних задатака.

У примерима петљи који следе, претпоставља се да кôд у телу петљи који није приказан не утиче на бројачке променљиве и не мења границе петљи.

```
for (int i = 0; i < n; i++)
   // kod slozenosti O(1)
```

Сложеност претходне петље је $O(n)$.

```
for (int i = m; i < n; i++)
   // kod slozenosti O(1)
```

Сложеност претходне петље је $O(n - m)$.

```
for (int i = 0; i < n; i += 2)
   // kod slozenosti O(1)
```

Сложеност претходне петље је $O(n)$. Пошто се петља извршава за парне вредности бројачке променљиве, тело петље се извршава око $\frac{n}{2}$ пута и константни фактор је $\frac{1}{2}$, али је сложеност и даље линеарна.

```
for (int i = 0, j = n-1; i < j; i++, j--)
   // kod slozenosti O(1)
```

Сложеност претходне петље је $O(n)$. Показивачи се сусрећу приближно на средини опсега, а тело петље се извршава око $\frac{n}{2}$ пута.

```
for (int i = 0; i < m; i++)
   for (int j = 0; j < n; j++)
      // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(mn)$. Заиста, спољашња петља се извршава m, а у њеном телу се унутрашња петља извршава n пута, па се тело унутрашње петље изврши тачно mn пута.

```
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
      // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(n^2)$. Заиста, спољашња петља се извршава n, а у њеном телу се унутрашња петља извршава n пута, па се тело унутрашње петље изврши тачно n^2 пута.

```
for (int i = 0; i < n; i++)
   for (int j = i+1; j < n; j++)
      // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(n^2)$. Број извршавања тела унутрашње петље је $(n-1)+(n-2)+\ldots+2+1$, што је једнако $\frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$. Константни фактор је $\frac{1}{2}$, али је сложеност квадратна. До истог резултата можемо доћи ако схватимо да у сваком кораку унутрашње петље пар бројача одређује једну комбинацију бројева од 0 до $n-1$. Зато број извршавања тела одговара броју двочланих комбинација скупа од n елемената, што је једнако $\binom{n}{2} = \frac{n(n-1)}{2}$.

```
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
      for (int k = 0; k < n; j++)
         // kod slozenosti O(1)
```

Сложеност претходних петљи је прилично очигледно $O(n^3)$.

```
for (int i = 0; i < n; i++)
   for (int j = i+1; j < n; j++)
      for (int k = j+1; k < n; j++)
         // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(n^3)$. Најлакши начин да се ово закључи је да се примети да сваком извршавању тела одговара једна трочлана комбинација елемената скупа $0, \ldots, n-1$. Пошто трочланих комбинација има $\binom{n}{3} = \frac{n(n-1)(n-2)}{3 \cdot 2 \cdot 1}$, сложеност је кубна, а константни фактор је $\frac{1}{6}$.

```
for (int i = 0; i < m; i++)
    // kod slozenosti O(1)

for (int i = 0; i < n; i++)
    // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(m + n)$. Наиме, тело прве петља се изврши m пута, а затим тело друге петље n пута, па се тела обе петље укупно изврше $m + n$ пута.

```
for (int i = 0; i < n; i++)
    // kod slozenosti O(1)

for (int i = 0; i < n; i++)
    // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(n)$. Наиме, тело прве петље се изврши n пута, а затим тело друге n пута, па се тела обе петље укупно изврше $2n$ пута, но то је $O(n)$.

```
for (int i = 0; i < n; i++)
   for (int j = i+1; j < n; j++)
       // kod slozenosti O(1)

for (int i = 0; i < n; i++)
    // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(n^2)$. Наиме, тело угнежђених петљи се изврши $\frac{n(n-1)}{2}$ пута, а затим тело друге петље n пута, што је заправо занемариво мало у односу на број извршавања тела угнежђених петљи. Дакле, први део кода апсолутно доминира временом извршавања и сложеност је $O(n^2)$.

Могло би се помислити да број угнежђених петљи одговара степену полинома, али то није увек случај.

```
for (int i = 1; i*i <= n; i++)
   // kod slozenosti O(1)
```

Иако садржи једну петљу, сложеност претходног кода није $O(n)$, већ $O(\sqrt{n})$. Наиме, петља се извршава све док је $i^2 \leq n$ тј. док је $i \leq \sqrt{n}$.

```
for (int i = 1; i < n; i *= 2)
   // kod slozenosti O(1)
```

Иако претходни код садржи петљу, његова сложеност је $O(\log n)$, јер се вредност променљиве `i` дуплира у сваком кораку, све док не престигне граничну вредност n.

```
for (int i = 0; i < 10; i++)
   // kod slozenosti O(1)
```

Сложеност претходног кода је $O(1)$. Иако је присутна петља, број њених извршавања је увек 10 и не зависи ни од једног параметара, па га можемо сматрати малом константом.

```
for (int i = 1; i <= n; i++)
    for (int j = 1; j < i; j *= 2)
          // kod slozenosti O(1)
```

Сложеност претходног кода је $O(n \log n)$. Сложеност унутрашње петље, за свако конкретно `i` је $O(\log i)$, па је укупна сложеност отприлике једнака $\log 1 + \log 2 + \ldots + \log n$, а за ово се може показати да је $O(n \log n)$ (јасно је да је израз мањи или једнак $n \log n$ јер је сваки сабирак мањи или једнак $\log n$, међутим, може се показати и да је збир већи или једнак од $\frac{n}{2} \log \frac{n}{2}$ (што је такође $\Theta(n \log n)$), занемаривањем првих $\frac{n}{2}$ сабирака, након чега остају само сабирци који су већи или једнаки $\log \frac{n}{2}$).

```
for (int i = n; i >= 1; i /= 2)
    for (int j = 1; j < i; j++)
          // kod slozenosti O(1)
```

Сложеност претходног кода је $O(n)$. Наиме, број извршавања унутрашње петље је $n + \frac{n}{2} + \frac{n}{4} + \ldots$, за шта се лако може показати да је одозго ограничено са $2n$.

```
for (int i = 0; i < n; i++) {
   for (int j = i+1; j < n && P(a[j]); j++)
          // kod slozenosti O(1)
   // kod slozenosti O(1)
   i = j;
}
```

Овакав код се може срести у алгоритмима у којима се анализирају све серије узастопних елемената низа који задовољавају неко својство `P` (на пример, коришћењем петљи овог облика можемо пронаћи најдужу серију узастопних парних елемената).

Иако постоје угнежђене петље, сложеност претходног кода је $O(n)$. Број извршавања унутрашње петље зависи од стања низа `a`, па не знамо унапред ни колико пута ће та петља бити покренута нити колико ће се пута њено тело извршити при сваком покретању. Међутим, да бисмо одредили укупну сложеност целог кода, то нам није ни потребно. Можемо извршити амортизовану анализу и израчунати укупан број извршавања тела унутрашње петље. Кључни детаљ је то што унутрашња петља креће од текуће вредности спољашње бројачке промењиве, док спољашња бројачка промењива након унутрашње петље наставља тамо где се унутрашња петља завршила. Зато при сваком пролазу кроз тело унутрашње петље промењива `j` има строго вредност већу него при сваком претходном пролазу, што се може десити највише n пута.

```
int j = 0;
for (int i = 0; i < n; i++) {
    while (j < n && P[j]) {
       // kod slozenosti O(1)
       j++;
    }
   // kod slozenosti O(1)
}
```

Иако постоје угнежђене петље, сложеност претходне петље је $O(n)$. Кључни детаљ је то што унутрашња петља нема иницијализацију промењиве `j` на нулу и бројач `j` у унутрашњој петљи се све време само увећава (исто као и бројач `i` у спољашњој петљи). Укупан број корака је стога ограничен са $2n$.

```
int l = 0, d = n-1;
while (l < d) {
  do l++; while (l < d && P(a[l]));
  do d--; while (l < d && Q(a[d]));
  if (l < d)
     // kod slozenosti O(1)
}
```

Сличан кôд се, на пример, може срести у алгоритму партиционисања низа. И претходни алгоритам је сложености $O(n)$ иако и он садржи угнежђене петље. То поново можемо утврдити амортизованом анализом (јер не знамо појединачни број извршавања унутрашњих петљи, али можемо лако проценити укупан корака који се у њима направи). Наиме, промењива `l` се само увећава кренувши од почетка, а `d` се само смањује кренувши од краја низа, док се не сусретну, што ће се десити у највише n корака.

Дакле, иако нам у већини случајева угнежђеност петљи описује сложеност алгоритма, треба бити обазрив и код анализирати пажљивије, да се сложеност не би преценила.

2.5 Скривена сложеност

Често процену сложености грубо вршимо тако што анализирамо структуру петљи у програму, занемарујући остале операције (обично за све сем петљи сматрамо да је $O(1)$). То може бити прилично варљиво, јер се у коду могу позивати функције (било кориснички дефинисане, било библиотечке) које нису константне сложености. Још горе, и неки оператори могу бити неконстантне сложености (обично линеарне).

```
string rez = "";
for (char c : s)
```

```
rez = c + rez;
```

Иако има само једну петљу, претходни фрагмент може бити сложености $O(n^2)$, где је n дужина ниске `s`. Наиме, додавање карактера на почетак ниске у језику C++ може бити линеарне сложености $O(n)$, где је n дужина ниске (јер захтева померање наредних карактера надесно).

2.6 Рекурентне једначине

Сложеност рекурзивних функција се често може описати рекурентним једначинама. Решење рекурентне једначине је функција $T(n)$ и за решење ћемо рећи да је у затвореном облику ако је изражено као елементарна функција по n (и не укључује са десне стране поновно реферисање на функцију T). Често ћемо се задовољити да уместо потпуно прецизног решења знамо само његово асимптотско понашање. Подсетимо се неколико најчешћих рекурентних једначина.

У првој групи се проблем своди на проблем димензије која је тачно за један мања од димензије полазног проблема.

- *Једначина*: $T(n) = T(n-1) + O(1), T(0) = O(1)$. *Пример*: Тражење минимума низа. *Решење*: $O(n)$.
- *Једначина*: $T(n) = T(n-1) + O(\log n), T(0) = O(1)$. *Пример*: Формирање балансираног бинарног дрвета. *Решење*: $O(n \log n)$.
- *Једначина*: $T(n) = T(n-1) + O(n), T(0) = O(1)$. *Пример*: Сортирање селекцијом. *Решење*: $O(n^2)$.

У другој групи се проблем своди на два (или више) проблема чија је димензија за један или два мања од димензије полазног проблема. То обично доводи до експоненцијалне сложености.

- *Једначина*: $T(n) = 2T(n-1) + O(1), T(0) = O(1)$. *Пример*: Ханојске куле. *Решење*: $O(2^n)$
- *Једначина*: $T(n) = T(n-1) + T(n-2) + O(1), T(0) = O(1)$. *Пример*: Фибоначијеви бројеви. *Решење*: $O(2^n)$

У наредној групи се проблем своди на један (или више) потпроблема који су значајно мање димензије од полазног (обично су бар дупло мањи). Ово доводи до полиномијалне сложености, па често и до веома ефикасних решења.

- *Једначина*: $T(n) = T(n/2) + O(1), T(0) = O(1)$. *Пример*: Бинарна претрага сортираног низа. *Решење*: $O(\log n)$.
- *Једначина*: $T(n) = T(n/2) + O(n), T(0) = O(1)$. *Пример*: Проналажење медијане (средишњег елемента) низа. *Решење*: $O(n)$.
- *Једначина*: $T(n) = 2T(n/2) + O(1), T(0) = O(1)$. *Пример*: Обилазак потпуног бинарног дрвета. *Решење*: $O(n)$.
- *Једначина*: $T(n) = 2T(n/2) + O(n), T(0) = O(1)$. *Пример*: Сортирање обједињавањем. *Решење*: $O(n \log n)$.

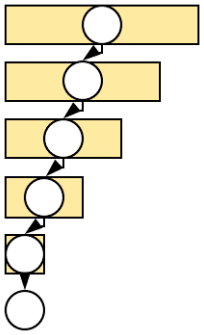
Ако су границе у самим једначинама егзактне, скоро у свим претходно набројаним једначинама дато решење није само горње ограничење, већ је асимптотски егзактно. На пример, решење једначине $T(n) = 2T(n/2) + \Theta(n), T(1) = \Theta(1)$ има решење $T(n) = \Theta(n \log n)$. Изузетак је пример Фибоначијевог низа где понашање јесте експоненцијално, али основа није 2, већ златни пресек $(1 + \sqrt{5})/2$.

Потпуно формално и прецизно извођење и доказивање асимптотског понашања решења ових једначина нам неће бити у директном фокусу. Много важније је стећи неку интуицију зашто су решења баш таква каква јесу (уз одређену дозу резерве, јер овакве грубе апроксимације некада могу довести до грешке). Један начин да се то уради је да се крене са "одмотавањем" рекурзије и да се види до чега се долази.

2.6.1 Једначина $T(n) = T(n-1) + O(1)$, $T(0) = O(1)$

На пример, код једначине $T(n) = T(n-1) + O(1)$ и $T(0) = O(1)$, након одмотавања добијамо да важи

$$\begin{aligned} T(n) &= T(n-1) + O(1) \\ &= T(n-2) + O(1) + O(1) \\ &= T(n-3) + O(1) + O(1) + O(1) \\ &= \dots \\ &= T(0) + n \cdot O(1) \\ &= O(1) + n \cdot O(1) \\ &= O(n). \end{aligned}$$



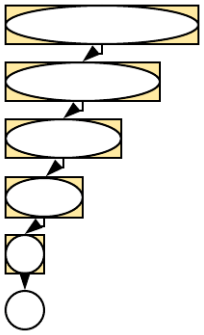
Слика 2.14: Дрво позива у случају $T(n) = T(n-1) + O(1)$, $T(0) = O(1)$ за $n = 4$. Правоугаоник означава димензију улаза, а елипса количину посла који се обавља у том чвору.

2.6.2 Једначина $T(n) = T(n-1) + O(n)$, $T(0) = O(1)$

Код једначине $T(n) = T(n-1) + O(n)$, $T(0) = O(1)$ слично добијамо n сабирака који су сви $O(n)$ тако да је укупна сума $O(n^2)$. Поставља се питање да ли је ова граница егзактна тј. да ли је могуће да је сложеност мања од изведеног горњег ограничења. Претпоставимо да је $T(n) = T(n-1) + cn$ и да је $T(0) = O(1)$. Тада је

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= T(n-2) + (n-1) + cn \\ &= \dots \\ &= T(0) + c(1 + \dots + n) \\ &= O(1) + cn(n+1)/2, \end{aligned}$$

тако да је $T(n) = \Theta(n^2)$.



Слика 2.15: Дрво позива у случају $T(n) = T(n-1) + O(n)$, $T(0) = O(1)$ за $n = 4$

2.6.3 Једначина $T(n) = T(n-1) + O(\log n)$, $T(0) = O(1)$

Покажимо још и шта се дешава са једначином $T(n) = T(n-1) + c\log n$, $T(0) = O(1)$. Одмотавањем добијамо

$$\begin{aligned} T(n) &= T(n-1) + c\log n \\ &= T(n-2) + c\log(n-1) + c\log(n) \\ &= \ldots \\ &= O(1) + c(\log 1 + \ldots + \log n). \end{aligned}$$

Пошто је логаритам растућа функција, сваки од n чланова овог збира ограничен је одозго вредношћу $\log n$. Зато је збир $O(n\log n)$. Докажимо да ово ограничење није превише грубо. Важи да је

$$\log 1 + \log 2 + \ldots + \log n \geq \log(n/2) + \log(n/2+1) + \ldots + \log n,$$

јер је првих $n/2$ чланова који су из суме изостављени сигурно ненегативно. Пошто је логаритам растућа функција (за основу већу од 1), сви сабирци у овом збиру су већи или једнаки $\log(n/2)$, па је

$$\log(n/2) + \log(n/2+1) + \ldots + \log n \geq \log(n/2) + \log(n/2) + \ldots + \log(n/2).$$

Збир на десној страни има $n/2$ истих сабирака и једнак је $(n/2)\cdot\log(n/2)$. Стога је почетни збир логаритама ограничен и одоздо и одозго функцијама које су $\Theta(n\log n)$, па је и сам $\Theta(n\log n)$.

Још један начин да се ово покаже који се често среће у литератури је следећи. Збир логаритама, једнак је логаритму производа, па заправо овде рачунамо вредност $\log 1 \cdot \ldots \cdot n = \log n!$. По Стирлинговој формули $n!$ се понаша као $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$. Зато се $\log n!$ понаша као $n\log n - n + O(\log n)$, па је укупан збир $\Theta(n\log n)$.

Можемо уочити неке правилности. Код свих једначина облика $T(n) = T(n-1) + f(n)$, $T(0) = c$, након одмотавања добијамо да је $T(n) = c + f(1) + f(2) + \ldots + f(n)$, тако да се одређивање асимптотског понашања своди на сумирање.

- Збир n сабирака реда $1 + 2 + \ldots + n$ има вредност $n(n+1)/2$, која је реда $\Theta(n^2)$. То је само дупло мање од вредности збира $n + \ldots + n$, који се састоји од n сабирака и има вредност n^2.
- Збир n сабирака $\log 1 + \ldots + \log n$ има вредност која се понаша као $n\log n - n$, што је асимптотски исто као вредност збира $\log n + \ldots + \log n$ који има n сабирака и вредност $n\log n$.
- Слично, збир $1^2 + \ldots + n^2$ има вредност $n(n+1)(2n+1)/6$, што је $\Theta(n^3)$ и само је око три пута мање од збира $n^2 + \ldots + n^2$ који има n сабирака и вредност n^3.

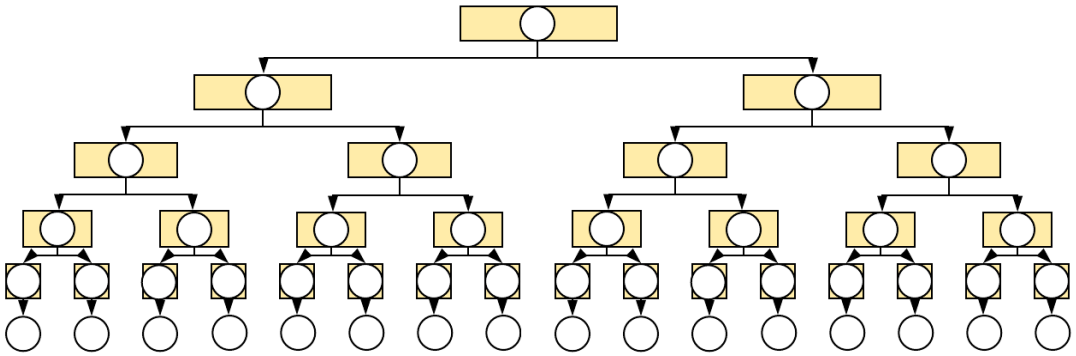
Иако овакве генерализације прете да буду непрецизне, са малом дозом резерве се може проценити да алгоритми у којима се n пута примењује нека операција сложености $\Theta(f(k))$ имају сложеност $\Theta(n \cdot f(n))$, чак и када се операција у сваком кораку примењује над подацима који су се за $O(1)$ повећали у односу на претходни корак и само у крајњој инстанци имамо $\Theta(n)$ података.

2.6.4 Једначина $T(n) = 2T(n-1) + O(1)$, $T(0) = O(1)$

Одмотавањем једначине $T(n) = 2T(n-1) + O(1)$, $T(0) = O(1)$ добијамо

$$\begin{aligned} T(n) &= 2T(n-1) + O(1) \\ &= 2(2T(n-2) + O(1)) + O(1) = 4T(n-2) + 2O(1) + O(1) \\ &= 4(2T(n-3) + O(1)) + 2O(1) + O(1) \\ &= 8T(n-3) + 4O(1) + 2O(1) + O(1) \\ &= \dots \\ &= 2^n T(0) + (2^{n-1} + \dots + 2 + 1) \cdot O(1) \\ &= 2^n \cdot O(1) + (2^n - 1) \cdot O(1) = O(2^n). \end{aligned}$$

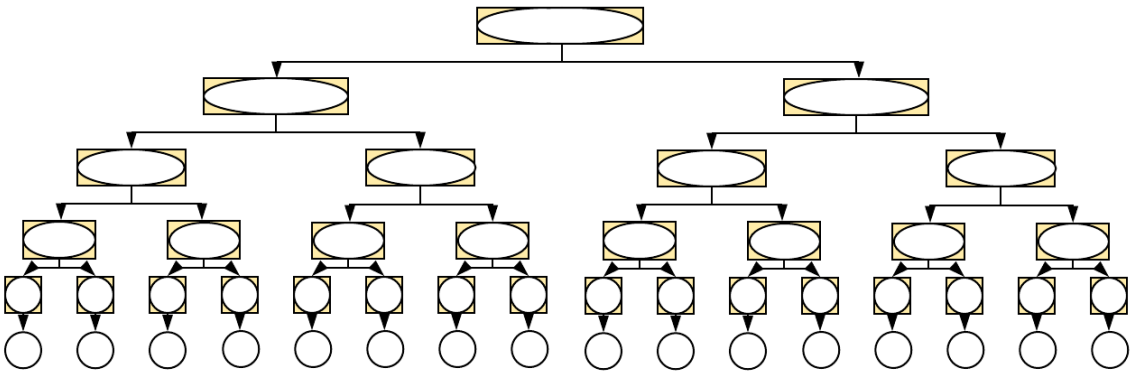
Дакле, иако се у сваком рекурзивном позиву ради мало посла, рекурзивних позива има експоненцијално много, што доводи до изразито неефикасног алгоритма.



Слика 2.16: Дрво позива у случају $T(n) = 2T(n-1) + O(1)$, $T(0) = O(1)$ за $n = 5$

2.6.5 Једначина $T(n) = 2T(n-1) + O(n)$, $T(0) = O(1)$

Функције које задовољавају једначину $T(n) = 2T(n-1) + O(n)$, $T(0) = O(1)$ такође показују експоненцијалну сложеност.



Слика 2.17: Дрво позива у случају $T(n) = 2T(n-1) + O(n)$, $T(0) = O(1)$ за $n = 5$

Одмотавањем једначине $T(n) = 2T(n-1) + c \cdot n$, $T(0) = O(1)$ добијамо

$$\begin{aligned} T(n) &= 2T(n-1) + c \cdot n \\ &= 2(2T(n-2) + c \cdot (n-1)) + c \cdot n = 4T(n-2) + c \cdot (2(n-1) + n) \\ &= 4(2T(n-3) + c \cdot (n-2)) + c \cdot (2(n-1) + n) \\ &= 8T(n-3) + c \cdot (4(n-2) + 2(n-1) + n) \\ &= \ldots \\ &= 2^n T(0) + c(2^{n-1} \cdot 1 + 2^{n-2} \cdot 2 + \ldots + 2(n-1) + n) \cdot O(1) \\ &= 2^n \cdot O(1) + \sum_{k=0}^{n} 2^k (n-k). \end{aligned}$$

Коришћењем раније изведених формула можемо једноставно израчунати и суму

$$\sum_{k=0}^{n} 2^k (n-k) = n + 2(n-1) + 2^2(n-2) + \ldots + 2^{n-1} \cdot 1.$$

Важи да је

$$\begin{aligned} \sum_{k=0}^{n} 2^k (n-k) &= n \sum_{k=0}^{n} 2^k - \sum_{k=0}^{n} k 2^k \\ &= n(2^{n+1} - 1) - ((n-1)2^{n+1} + 2) \\ &= 2^{n+1} - n - 2 \end{aligned}$$

Зато је $T(n) = 2^n \cdot O(1) + 2^{n+1} - n - 2$. Дакле, и у овом случају функција показује експоненцијално понашање $O(2^n)$.

2.6.6 Мастер теорема

Једначине засноване на декомпозицији проблема на неколико мањих потпроблема које су облика $T(n) = aT(n/b) + O(n^k)$, $T(0) = O(1)$ се решавају на основу **мастер теореме**.

Теорема: Решење рекурентне релације $T(n) = aT(n/b) + cn^k$, где су a и b целобројне константе такве да важи $a \geq 1$ и $b \geq 1$, и c и k су позитивне реалне константе је

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & \text{ако је } \log_b a > k \\ \Theta(n^k \log n), & \text{ако је } \log_b a = k \\ \Theta(n^k), & \text{ако је } \log_b a < k \end{cases}$$

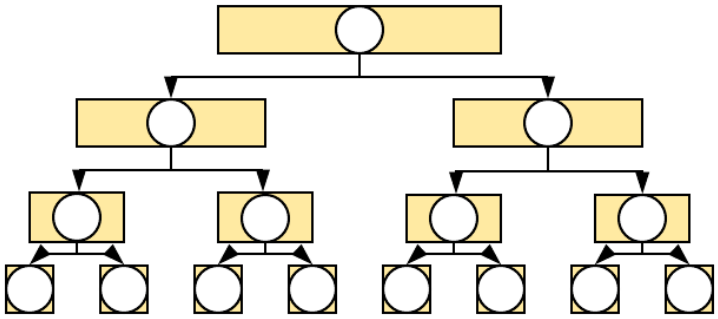
Нећемо давати доказ ове теореме, али покушајмо опет да дамо неко интуитивно јасно објашњење.

2.6.7 Једначина $T(n) = 2 \cdot T(n/2) + O(1)$, $T(1) = O(1)$

У првом случају се добија дрво рекурзивних позива чији број чворова доминира послом који се ради у сваком чвору. Размотримо, на пример, једначину $T(n) = 2 \cdot T(n/2) + O(1)$, $T(1) = O(1)$. Дрво ће садржати $O(n)$ чворова, а у сваком чвору ће се вршити посао који захтева $O(1)$ операција. Одмотавањем рекурентне једначине, добијамо

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + O(1) \\ &= 4 \cdot T(n/4) + 2 \cdot O(1) + O(1) \\ &= 8 \cdot T(n/8) + 4 \cdot O(1) + 2 \cdot O(1) + O(1) \\ &= 2^k \cdot T(n/2^k) + (2^{k-1} + \ldots + 2 + 1) \cdot O(1). \end{aligned}$$

Ако је $n = 2^k$ добијамо да је $n/2^k = 1$, па пошто је на основу формуле за збир геометријског низа $2^{k-1} + \ldots + 2 + 1 = 2^k - 1$, сложеност је $\Theta(n)$. И када n није степен двојке, добија се исто асимптотско понашање (што се може доказати ограничавањем одозго и одоздо степенима двојке).



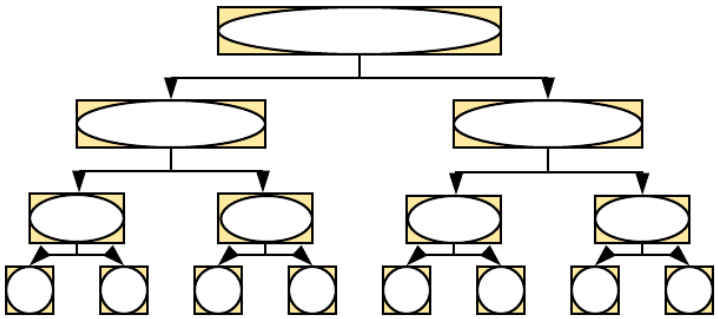
Слика 2.18: Дрво позива у случају $T(n) = 2T(n/2) + O(1), T(1) = O(1)$ за $n = 8$

2.6.8 Једначина $T(n) = 2 \cdot T(n/2) + c \cdot n$, $T(1) = O(1)$

У другом случају су број чворова и посао који се ради на неки начин уравнотежени. Размотримо, на пример, једначину $T(n) = 2 \cdot T(n/2) + c \cdot n, T(1) = O(1)$ и поново покушајмо да је одмотамо.

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + c \cdot n \\ &= 2 \cdot (2 \cdot T(n/4) + c \cdot n/2) + c \cdot n \\ &= 4T(n/4) + c \cdot n + c \cdot n \\ &= 4(2T(n/8) + c \cdot n/4) + 2 \cdot c \cdot n \\ &= 8T(n/8) + 3 \cdot c \cdot n \\ &= \ldots \\ &= 2^k \cdot T(n/2^k) + k \cdot c \cdot n. \end{aligned}$$

Ако је $n = 2^k$ после $k = \log_2 n$ корака $n/2^k$ ће достићи вредност 1 тако да ће збир бити реда величине $n \cdot O(1) + \log_2 n \cdot c \cdot n = \Theta(n \log n)$. Исто важи и када n није степен двојке.



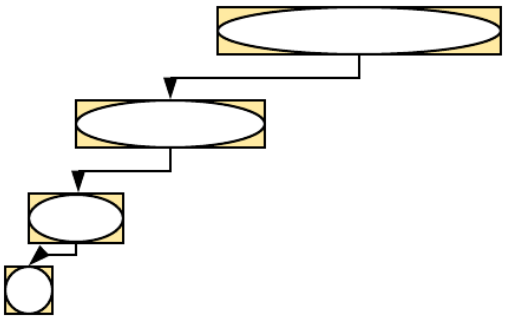
Слика 2.19: Дрво позива у случају $T(n) = 2T(n/2) + O(n), T(1) = O(1)$ за $n = 8$

2.6.9 Једначина $T(n) = T(n/2) + cn$, $T(1) = O(1)$

У трећем случају посао који се ради у чворовима доминира бројем чворова. Размотримо једначину $T(n) = T(n/2) + cn, T(1) = O(1)$. Њеним одмотавањем добијамо да је

$$\begin{aligned} T(n) &= T(n/2) + cn \\ &= T(n/4) + cn/2 + cn \\ &= T(n/8) + cn/4 + cn/2 + cn \\ &= \ldots \\ &= T(n/2^k) + cn(1/2^{k-1} + \ldots + 1/2 + 1). \end{aligned}$$

Поново, ако је $n = 2^k$, тада је први члан једнак $O(1)$ и пошто је на основу формуле за збир геометријског низа $1/2^{k-1} + \ldots + 1/2 + 1 = (1 - (1/2)^k)/(1 - (1/2)) = 2 - 2/n$ збир је једнак $O(1) + cn(2 - 2/n) = \Theta(n)$.



Слика 2.20: Дрво позива у случају $T(n) = T(n/2) + O(n)$, $T(1) = O(1)$ за $n = 8$

2.6.10 Остали типови једначина

Прокоментаришимо да се у неким проблемима добијају једначине које нису баш у сваком рекурзивном позиву идентичне овим наведеним. На пример, приликом анализе алгоритма QuickSort, ако је пивот тачно на средини низа, важи да је $T(n) = 2T(n/2) + O(n)$ и $T(1) = O(1)$. Када би се то стално догађало, решење би било $T(n) = O(n \log n)$, међутим, вероватноћа да се то догоди је страшно мала, јер у већини случајева пивот не дели низ на два дела потпуно исте димензије и зато треба бити обазрив. Ако би се десило да пивот стално завршавао на једном крају низа, једначина би била $T(n) = T(n-1) + O(n)$, $T(1) = O(1)$, што би довело до сложености $O(n^2)$, што и јесте сложеност најгорег случаја. Анализом коју ћемо приказати касније се може утврдити да је просечна сложеност $O(n \log n)$ тј. да иако пивот није стално на средини, да је у довољном процену случајева негде близу ње (рецимо између 25% и 75% дужине низа). Слична анализа важи и за проблем проналажења медијане.

Међутим, постоје и алгоритми код којих ствари стоје другачије. Приликом обиласка бинарног дрвета, балансираност нема утицаја. Наиме, ако је дрво потпуно, тада је једначина $T(n) = 2T(n/2) + O(1), T(1) = O(1)$, чије је решење $O(n)$. Међутим, чак и када је дрво издегенерисано у листу, једначина је $T(n) = T(n-1) + O(1), T(1) = O(1)$, чије је решење опет $O(n)$. Какав год да је однос броја чворова у левом и десном поддрвету решење ће бити $O(n)$. То се може описати једначином $T(n) = T(k) + T(n-k-1) + O(1), T(1) = O(1)$, за $0 \leq k \leq n-1$, чије ће решење бити $O(n)$, без обзира на то какво се k појављује у разним рекурзивним позивима.

2.7 Анализа просечне сложености

2.7.1 Анализа просечне сложености алгоритма QuickSort

Упечатљиво својство алгоритма QuickSort је ефикасност у пракси насупрот квадратној сложености најгорег случаја. Ово запажање сугерише да су најгори случајеви ретки и да је просечна сложеност овог алгоритма осетно повољнија.

Претпоставимо да је једнака вероватноћа да ће произвољни елемент бити изабран за пивот и да је једнака вероватноћа да пивот након партиционисања заврши на било којој позицији од 0 до $n-1$. Ако бројимо само

упоређивања (број замена је мањи или једнак броју упоређивања), сложеност партиционисања је $n-1$. Тада просечна сложеност задовољава наредну рекурентну једначину.

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

Први сабирак се креће од $T(0)$ до $T(n-1)$, а други од $T(n-1)$ до $T(0)$, тако да се свако $T(i)$ јавља тачно два пута. Зато за $n \geq 1$ важи

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

Ово је такозвана *једначина са потпуном историјом*, јер се вредност $T(n)$ израчунава преко свих претходних вредности $T(i)$. Један начин да се историја елиминише је да се посматрају разлике између суседних чланова низа чиме се добија једначина која описује везу између два суседна члана. У овом случају се сваки од сабирака $T(i)$ дели са n, те пре одузимања сваки од узастопних чланова треба помножити са одговарајућим фактором. Тако се добија

$$\begin{aligned} nT(n) - (n-1)T(n-1) = &\left(n(n-1) + 2\sum_{i=0}^{n-1}T(i)\right) - \\ &\left((n-1)(n-2) + 2\sum_{i=0}^{n-2}T(i)\right) \\ = &\, 2(n-1) + 2T(n-1) \end{aligned}$$

Зато је

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

Иако је ова једначина линеарна рекурентна једначина која повезује само два узастопна члана низа, она није са константним коефицијентима и потребно је мало инвентивности да бисмо је решили. Дељење са $n+1$ нам даје погоднији облик.

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

Наиме, сада се види да су два члана која укључују непознату функцију $T(n)$ истог облика, па једначину можемо једноставно одмотати и коришћењем чињенице да је $T(0) = 0$ добити

$$\frac{T(n)}{n+1} = \frac{2(n-1)}{n(n+1)} + \frac{2(n-2)}{(n-1)n} + \ldots + \frac{2\cdot(1-1)}{1(1+1)} = 2\sum_{i=1}^{n}\frac{i-1}{i(i+1)}$$

Централно питање постаје како израчунати збир

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

Томе помаже раздвајање сабирака на парцијалне разломке. Из једначине

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

следи да је $Ai + A + Bi = i - 1$, па је $A = -1$, $B = 2$ и важи

$$\frac{i-1}{i(i+1)} = \frac{2}{i+1} - \frac{1}{i}$$

Зато је

$$\sum_{i=1}^{n} \frac{i-1}{i(i+1)} = \frac{2}{1+1} - \frac{1}{1} + \frac{2}{2+1} - \frac{1}{2} + \frac{2}{3+1} - \frac{1}{3} + \ldots + \frac{2}{n+1} - \frac{1}{n}$$

Можемо груписати разломке са истим имениоцем и добити

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

Зато је

$$T(n) = 2(n+1) \cdot \left(1 + \frac{1}{2} + \ldots + \frac{1}{n}\right) - 4n.$$

Знамо да се хармонијски збир $1 + 1/2 + \ldots + 1/n$ асимптотски понаша као $\log n + \gamma$, где је γ Ојлер-Маскеронијева константа $\gamma \approx 0,57722$ и зато је

$$T(n) = \Theta\left(2(n+1)(\log n + \gamma) - 4n\right) = \Theta(n \log n).$$

2.8 Амортизована анализа сложености

У неким ситуацијама се извесне операције понављају пуно пута током извршавања програма. У многим ситуацијама можемо да допустимо да појединачно извршавање неке операције траје и мало дуже, ако смо сигурни да више извршавања те операције у збиру неће трајати предуго (ако постоји довољно извршавања те операције која ће трајати кратко). Анализа укупне дужине трајања већег броја операција назива се *амортизована анализа сложености*. Амортизована цена извршавања n операција подразумева количник њихове укупне дужине извршавања и броја n. Илуструјмо је на једном примеру.

2.8.1 Динамички низ

Једна од најчешће употребљаваних структура података је динамички низ. Размотримо колико је потребно времена да се у њега смести елемената. Када у низу нема довољно простора да се смести наредни елемент, низ се динамички реалоцира. У најгорем случају ово подразумева копирање старог садржаја низа на нову локацију, што је операција сложености $O(m)$, где је m број елемената уписаних у низ. Поставља се питање како приликом реалокација одређивати број елемената проширеног низа.

2.8.1.1 Аритметичка стратегија

Једна стратегија може бити аритметичка и она подразумева да се кренувши од празног низа приликом сваке реалокације величина низа повећа за неки број k (ништа се суштински не би променило у анализи и да је иницијални капацитет уместо 0 неки број m_0). Избројмо колико је пута потребно извршити упис елемента у низ (та операција је обично најспорија), при чему ту рачунамо уписе нових елемената и уписе настале током померања постојећих елемената током реалокације. У првом кораку примене аритметичке стратегије алоцирамо k елемената и затим у k наредних корака вршимо упис по једног елемента. Онда вршимо реалокацију на величину $2k$ и притом преписујемо првих k елемената низа. Након тога уписујемо наредних k елемената, а онда приликом реалокације преписујемо $2k$ елемената. Сличан поступак се наставља све док се не упише елемент n. Дакле, број операција је онда једнак $k + k + k + 2k + k + 3k + \ldots$. Да би се сместило n елемената, реалокацију је потребно вршити око n/k пута, па ће укупан број операција бити отприлике једнак

$$\frac{n}{k}k + k \cdot (1 + 2 + \ldots + \frac{n}{k}) = n + k\frac{\frac{n}{k}(\frac{n}{k}+1)}{2} = \frac{n^2}{2k} + \frac{3n}{2}$$

Дакле, укупан број уписа асимптотски је једнак $\frac{n^2}{2k}$ тј. $O(n^2)$ и стога је амортизована цена једне операције асимптотски једнака $\frac{n}{2k}$, што је $O(n)$, додуше, за доста малу вредност константе уз n (што је веће k, то је реалокација мање, па је цена операције мања, али се цена плаћа кроз веће заузеће меморије и мању попуњеност алоцираног простора).

2.8.1.2 Геометријска стратегија

Друга стратегија може бити геометријска и она подразумева да се сваки пут величина низа повећа q пута за неки фактор $q > 1$. Претпоставимо да је почетна величина низа m_0. Дакле, након почетне алокације можемо да упишемо m_0 елемената. Након тога се врши прва реалокација у којој се величина низа повећава на qm_0 елемената и при чему се преписује m_0 елемената. Након тога се врши упис наредних $qm_0 - m_0$ елемената. У наредној реалокацији величина низа се повећава на q^2m_0 и притом се преписује qm_0 елемената. Након тога се уписује преосталих $q^2m_0 - qm_0$ елеманата. Поступак се даље наставља по истом принципу. Дакле, укупан број уписа у низ једнак је

$$m_0 + m_0 + (qm_0 - m_0) + qm_0 + (q^2m_0 - qm_0) + \ldots = m_0 + qm_0 + q^2m_0 + \ldots$$

После r реалокација укупан број уписа једнак је

$$m_0(1 + \ldots + q^r) = m_0\frac{q^{r+1} - 1}{q - 1}.$$

Ако претпоставимо да је цео низ попуњен после r реалокација, тј. да је $n = m_0q^r$, онда је укупан број операција потребних за попуњавање низа једнак

$$m_0\frac{q^{\frac{n}{m_0}} - 1}{q - 1} = \frac{qn - m_0}{q - 1}.$$

Асимптотски је, дакле, укупна цена извођења свих операција једнака $O(n)$, а амортизована цена извођења једне операције додавања у овакав низ је $O(1)$. Константан фактор једнак је $\frac{q}{q-1}$ и он је све мањи како q расте. Заиста, са повећањем q врши се све мање реалокација, али се цена плаћа већим ангажовањем меморије тј. мањом попуњеношћу низа.

Амортизована анализа сложености нам показује да са становишта времена извршавања геометријска стратегија реалокације даје значајно боље резултате него аритметичка.

2.9 Савети за побољшање сложености

Кључни савет за побољшање сложености је то да рачунар ради само оно што је неопходно да би се добио коначан резултат. Када се та идеја мало детаљније разради, добијамо следећи низ савета који нас често доводе до алгоритама мање сложености:

- Немој терати рачунар да врши дуготрајна израчунавања која се могу извршити и "пешке", применом математике.
- Немој терати рачунар да више пута израчунава једно те исто – упамти потребне резултате израчунавања у меморији, да их не би рачунао више пута.
- Немој терати рачунар да израчунава ствари које нису потребне за добијање коначног решења проблема.
- Немој терати рачунар да испитује случајеве за које унапред можеш закључити да не могу бити тражено решење проблема.
- Ако је то могуће, припреми податке тако да се касније могу ефикасније обрадити.
- Користи ефикасније структуре података.
- …

У наставку овог поглавља приказаћемо низ задатака које ћемо решити различитим алгоритмима, анализираћемо њихову асимптотску сложеност најгорег случаја и приказаћемо како се на бази приказаних савета могу изградити значајно ефикаснији алгоритми.