Алгоритми и структуре података

Материјал са предавања

Миодраг Живковић
е-маил: ezivkovm@matf.bg.ac.rs
УРЛ: www.matf.bg.ac.rs/~ezivkovm

Весна Маринковић
е-маил: vesnap@matf.bg.ac.rs
УРЛ: www.matf.bg.ac.rs/~vesnap
Математички факултет, Београд

Аутори:
проф. др Миодраг Живковић, редовни професор Математичког факултета у Београду
др Весна Маринковић, доцент Математичког факултета у Београду

АЛГОРИТМИ И СТРУКТУРЕ ПОДАТАКА

Садржај

Глава 1

Увод

1.1 Увод

Сврха овог поглавља (видети уводни део књиге [1]) је увод у начин размишљања приликом конструкције и анализе алгоритама. Биће приказан начин специфицирања алгоритама, две основне стратегије за конструкцију алгоритама и неке основне идеје које се користе при анализи алгоритама, односно доказивању коректности алгоритама.

Два алгоритма који ће бити приказани решавају проблем сортирања (уређивања у неопадајућем поретку) датог низа од n бројева. Алгоритми се излажу у облику псеудокода, који, иако се не може директно превести у било који стандардни програмски језик, описује структуру алгоритма довољно јасно, тако да програмер може да га реализује на програмском језику по свом избору. Приказују се алгоритам сортирања уметањем, који користи инкрементални приступ, и сортирање обједињавањем, алгоритам који користи идеју разлагања (подели-па-владај, divide-and-conquer). Иако време потребно за извршавање ова два алгоритма расте са n, брзине раста тих времена нису исте. Одредићемо времена извршавања ових алгоритама и упознати се са корисном нотацијом за њихово изражавање.

1.2 Алгоритми

Неформално, **алгоритам** је прецизно дефинисана процедура израчунавања, која, полазећи од неке вредности или скупа вредности као **улаза**, производи неку вредност, или скуп вредности, као **излаз**.

Алгоритам такође можемо посматрати као средство за решавање прецизно дефинисаног (рачунарског) **проблема**. Поставка проблема прецизира жељени однос излаза са улазом. Алгоритам описује специјалну процедуру израчунавања којом се постиже жељени однос излаза са улазом.

На пример, нека је потребно поређати чланове датог низа бројева тако да чине неопадајући низ. На овај проблем се често наилази; он може да послужи као добра подлога за увођење многих стандардних техника за конструкцију и анализу. Формално се проблем **сортирања** дефинише на следећи начин:

Улаз: Низ бројева $(a_1, a_2, \ldots, a_n)$.

Излаз: Пермутација $(a'_1, a'_2, \ldots, a'_n)$ улазног низа таква да је $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

На пример, за дати низ $(31, 41, 59, 26, 41, 58)$ алгоритам сортирања као излаз даје низ $(26, 31, 41, 41, 58, 59)$. Сваки овакав улазни низ зове се **инстанца** проблема сортирања. Уопште, **инстанцу проблема** чини улаз (који задовољава сва ограничења која проистичу из формулације проблема) неопходан да би се израчунало решење проблема.

Сортирање је фундаментална операција (многи програми је користе као међукорак), па је због тога развијен велики број добрих алгоритама за сортирање. Који алгоритам је најбољи за конкретну примену зависи, поред осталог, од:

- дужине низа који треба сортирати,
- могућих ограничења за чланове низа,
- од тога у којој су мери чланови низа већ сортирани,
- типа уређаја у коме је низ смештен (оперативна меморија, диск, ...).

Каже се да је алгоритам **коректан** ако за се за сваку инстанцу завршава са тачним излазом. Тада кажемо да алгоритам **решава** дати проблем.

Алгоритам може бити специфициран на српском језику, као рачунарски програм, или чак као пројекат уређаја који га извршава. Једини захтев је да спецификација мора да обезбеди прецизан опис процедуре израчунавања.

Сортирање, наравно, није једини проблем за чије решавање су развијени алгоритми. Наводимо примере проблема који се решавају алгоритмима:

- Један од циљева пручавања људског генома је идентификација свих око 100000 гена у људској ДНК, одређивање редоследа око три милијарде базних парова који чине људску ДНК, смештање тих информација у базе података, и развијање алата за анализу података. Сваки од тих корака захтева софистициране алгоритме. Циљ је постизање уштеда у утрошеном времену, и за људе и рачунаре, као и добијање што више података.
- Интернет омогућује људима широм света да се брзо повезују и да проналазе велике количине података. Проблеми које при томе треба решавати су проналажење добрих путева којим ће подаци бити усмерени, као и употреба претраживача за брзо проналажење страница на којима се жељена информација налази.
- Електронска трговина омогућује да се роба и услуге размењују електронски. При томе је од суштинског значаја безбедно чување података као што су бројеви кредитних картица, лозинке и бројеви банковних рачуна. Агоритми за шифровање са јавним кључем и дигитални потписи који се за те потребе користе заснивају се на нумеричким алгоритмима и теорији бројева.

1.2.1 Ефикасност алгоритама

Претпоставимо на тренутак да су рачунари бесконачно брзи и да је меморија бесплатна. Да ли би тада било потребе да се проучавају алгоритми? Одговор је - да, у најмању руку зато што треба доказати да се наш метод решавања завршава (зауставља) и да даје коректан одговор.

Ако би рачунари били бесконачно брзи, било који коректан метод за решавање проблема био би прихватљив. И даље би постојао захтев да је реализација алгоритма добра, односно добро документована, али би се обично међу методима бирао онај чија је реализација најједноставнија.

У стварности, рачунари јесу брзи, али не бесконачно брзи. Меморија може бити јефтина, али није бесплатна. Због тога алгоритми морају бити ефикасни у погледу потребног времена и меморијског простора.

Алгоритми намењени решавању истог проблема често се драстично разликују у погледу ефикасности. У наставку ћемо размотрити два алгоритма за сортирање. Трајање извршавања првог од њих — **сортирања уметањем** је $c_1 n^2$ ако се сортира низ дужине n, где је c_1 константа која не зависи од n. Другим речима, утрошено време пропорционално је са n^2. Трајање извршавања другог алгоритма, **сортирања обједињавањем**, је $c_2 n \log n$, где је $\log n = \log_2 n$, а c_2 је друга константа, која такође не зависи од n. Обично је константа за сортирање уметањем мања од константе за сортирање обједињавањем, односно $c_1 < c_2$. Видећемо да константни фактори много мање утичу на време извршавања него облик зависности од величине улаза n. У изразу код сортирања обједињавањем имамо фактор $\log n$, а код сортирања уметањем фактор n, који је много већи. Иако је сортирање уметањем обично брже од сортирања обједињавањем за низове мале дужине, кад величина улаза n постане довољно велика, предност сортирања обједињавањем у односу $\log n/n$ довољна је да компензује разлику у константним факторима. Колико год c_1 било мање од c_2, увек ће постојати гранична вредност n, после које ће сортирање обједињавањем бити брже.

Претпоставимо да хоћемо да упоредимо бржи рачунар A на коме се извршава сортирање уметањем и спорији рачунар B на коме се извршава сортирање обједињавањем. Оба рачунара треба да сортирају низ од милион бројева. Нека рачунар A извршава милијарду операција у секунди, а рачунар B само десет милиона операција у секунди, односно да је рачунар A 100 пута бржи од рачунара B. Претпоставимо поред тога да је највештији програмер на свету написао програм за сортирање уметањем на машинском језику за рачунар A, тако да добијени програм извршава $2n^2$ инструкција при сортирању низа дужине n (односно $c_1 = 2$). С друге стране, програм за сортирање обједињавање за рачунар B написао је осредњи програмер на високом језику, користећи неефикасни преводилац, тако да добијени програм извршава $50n \log n$ инструкција (односно $c_2 = 50$). За сортирање милион бројева рачунару A потребно је

$$\frac{2 \cdot (10^6)^2 \text{instrukcija}}{10^9 \text{instrukcija/sec}} = 2000\text{sec},$$

док је рачунару B потребно

$$\frac{50 \cdot 10^6 \log 10^6 \text{instrukcija}}{10^7 \text{instrukcija/sec}} \simeq 100\text{sec}.$$

Користећи алгоритам чије време извршавања спорије расте са n, чак и са лошијим преводиоцем, рачунар B је бржи 20 пута од рачунара A. Предност сортирања обједињавањем постаје још значајнија када се сортира десет милиона бројева: тада је за сортирање обједињавањем потребно 2.3 дана, а сортирање обједињавањем завршава се после мање од 20 min. Уопште, са порастом величине низа расте предност сортирања обједињавањем.

1.3 Сортирање уметањем

Сортирање уметањем је ефикасан алгоритам за сортирање малог броја елемената. Алгоритам ради на начин на који већина људи ређа у руци карте за играње. Започињемо са празном левом руком и неколико карата на столу, окренутих лицем на доле. Затим узимамо једну по једну карту са стола и умећемо је на њено место у левој руци, као што је то приказано на слици 1. У сваком тренутку су карте у левој руци поређане по величини (сортиране); при томе су то карте које су биле на врху гомиле карата на столу.

Слика 1.1: Сортирање карата применом сортирања уметањем

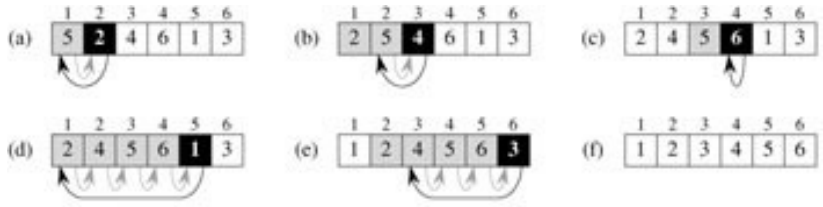
Алгоритам се може описати псеудокодом. Улазни параметри алгоритма су низ A који треба сортирати, и n — дужина низа A. Улазни низ се **сортира у месту**: бројеви се премештају у оквиру низа A, при чему се највише константан број њих записује ван низа у сваком тренутку. У тренутку завршетка алгоритма низ A садржи сортирани излазни низ.

```
SortiranjeUmetanjem(A, n)
1 for j ← 2 to n do
2   t ← A[j]
3   {уметнути A[j] у сортирани низ A[1..j − 1]}
4   i ← j − 1
5   while i > 0 and A[i] > t do
6     A[i + 1] ← A[i]
7     i ← i − 1
8   A[i + 1] ← t
```

1.3.1 Инваријанта петље и доказ коректности сортирања уметањем

На слици 2.2 приказано је како алгоритам ради за низ $A = (5, 2, 4, 6, 1, 3)$. Индекс j је индекс "текуће карте", карте која се тренутно умеће у леву руку. На почетку сваког проласка кроз "спољашњу" **for** петљу, којој одговара индекс j, подниз $A[1..j-1]$ одговара картама које се тренутно налазе у левој руци, а елементи $A[j+1..n]$ одговарају картама које се још налазе на столу. Прецизније, елементи $A[1..j-1]$ су уствари елементи који су се на почетку налазили на позицијама од 1 до $j-1$, али су сада поређани по величини. За ту особину подниза $A[1..j-1]$ кажемо да је **инваријанта петље**.

На почетку сваког проласка кроз **for** петљу у линијама $1-8$ подниз $A[1..j-1]$ састоји се од елемената који су на почетку били у поднизу $A[1..j-1]$, али у неопадајућем поретку.



Слика 1.2: Резултат примене алгоритма *SortiranjeUmetanjem* на низ $A = (5, 2, 4, 6, 1, 3)$. (a)-(e) Итерације **for** петље из линија $1-8$. У свакој итерацији црни квадрат садржи број узет из $A[j]$, који се упоређује са вредностима из сивих квадрата лево од себе у линији 5. Сиве стрелице показују бројеве који су померени за једно место удесно у линији 6, а црна стрелица показује где је број преписан у линији 8. (f) Сортирани низ на крају.

Инваријанта петље нам помаже да схватимо зашто је алгоритам коректан. У вези са инваријантом петље треба да докажемо три ствари:

Иницијализација: Тврђење је тачно пре првог проласка кроз петљу.

Одржавање: Ако је тврђење тачно пре проласка кроз петљу, онда је тачно и пре наредног проласка кроз петљу.

Завршетак: По завршетку петље тачност инваријанте петље има за последицу тврђење из кога следи да је алгоритам коректан.

Кад су прва два тврђења тачна онда је инваријанта петље тачна пре сваког проласка кроз петљу, на основу принципа математичке индукције. Треће тврђење је коначан доказ коректности алгоритма. Погледајмо ове три особине у случају сортирања уметањем.

Иницијализација: На почетку треба показати да је тврђење тачно пре првог проласка кроз петљу, кад је $j = 2$. Тада се подниз $A[1..j-1]$ састоји само од елемента $A[1]$, који је уствари првобитни елеменат

$A[1]$. Поред тога, подниз је (тривијално) уређен неопадајуће. Према томе, инваријанта петље је тачна пре првог проласка кроз петљу.

Одржавање: Сада треба доказати да сваки наредни пролазак кроз петљу задржава тачност инваријанте петље. Заиста, у телу спољашње **for** петље елементи $A[j-1]$, $A[j-2]$, $A[j-3]$, ... премештају се по једно место удесно, све док се не нађе право место за $A[j]$ (линије $4-7$). Тада се вредност $A[j]$ копира на то место (линија 8).

Завршетак: Испитајмо шта се дешава по завршетку петље. Спољашња петља се завршава кад j постане веће од n, тј. кад је $j = n+1$. Замењујући j са $n+1$ у инваријанти петље, добијамо да се подниз $A[1..n]$ састоји од елемената који су у почетку били у $A[1..n]$, али у неопадајућем поретку. Међутим, подниз $A[1..n]$ је уствари цео низ! Према томе, цео низ је сортиран, што значи да је алгоритам коректан.

1.3.2 Псеудокод

За специфицирање алгоритма искористили смо специјални неформални језик, **псеудокод**. Наводимо нека од правила овог језика.

1. Увлачење редова одговара блоковској структури. На пример, **for** петља која почиње од линије 1, састоји се од линија $2-8$; тело **while** петље која почиње у линији 5 садржи линије $6-7$, али не и линију 8. Увлачење линија примењује се на уобичајени начин и на **if** $-$ **then** $-$ **else** наредбе. Сврха увлачења редова је повећање јасноће кода.

2. Петље **while** и **for**, као и условне наредбе **if**, **then**, **else** имају значење слично као у Паскалу. Изузетак је да у **for** петљи бројачка променљива после проласка кроз петљу задржава своју вредност. Тако, кад се **for** петља из линије 1 (**for** $j \leftarrow 2$ **to** n **do**) заврши, биће $j = n+1$.

3. Текст у витичастим заградама је коментар.

4. Члан низа означава се додавањем индекса у угластим заградама. На пример, $A[i]$ је i-ти члан низа A. Ознака .. служи за издвајање подниза. Тако $A[1..j]$ означава подниз низа A који се састоји од j елемената $A[1]$, $A[2]$,...,$A[j]$.

1.4 Анализа алгоритама

Анализа алгоритма подразумева предвиђање ресурса потребних за његово извршавање, и то најчешће времена, односно меморијског простора. Уопште, анализирајући неколико алгоритама — кандидата за решавање датог проблема, обично се лако издваја најефикаснији. Резултат анализе може бити да постоји више од једног прихватљивог кандидата, док се више лошијих алгоритама одбацују.

Претпоставља се да се алгоритам извршава као рачунарски програм, инструкција по инструкција, без истовремених операција.

Прецизнија анализа захтевала би дефинисање скупа инструкција на рачунару и њихова трајања. То би било превише напорно и не би дало много информација потребних при анализи и конструкцији алгоритама. С друге стране, не сме се злупотребити модел рачунара — шта ако бисмо претпоставили да рачунар има инструкцију за сортирање? Таква претпоставка је, наравно, нереална. Дакле, претпостављаћемо да рачунар може да извршава инструкције које постоје на обичним рачунарима: аритметичке (сабирање, одузимање, множење, дељење), копирање података, инструкције за контролу тока (условни и безусловни скок, позив потпрограма и повратак из њега).

Анализа чак и једноставих алгоритама у оваквом моделу може да представља изазов. Потребни математички алати су комбинаторика, теорија вероватноће, манипулација са алгебарским изразима и способност препознавања најзначајнијих чланова у изразу. Због тога што се алгоритам може понашати другачије за сваки могући улаз, потребно је имати на располагању средства за представљање њиховог понашања једноставним, лако разумљивим изразима.

1.4.1 Анализа сортирања уметањем

Време потребно за *SortiranjeUmetanjem* зависи од улаза: сортирање хиљаду бројева траје дуже од сортирања три броја. Поред тога, *SortiranjeUmetanjem* може да траје различито ако се сортирају два низа исте дужине, зависно од степена у коме су они већ сортирани. У општем случају, време које троши алгоритам расте са величином улаза, па је уобичајено да се време извршавања програма изражава у зависности од величине улаза. При томе се "време извршавања" и "величина улаза" морају пажљивије дефинисати.

Појам **величине улаза** зависи од проблема који се решава. За многе проблеме, као што је сортирање, најприроднија мера је *број елемената у улазу* — на пример дужина n низа који се сортира. За многе друге проблеме, као што је на пример множење два броја, то је *укупан број битова* потребних да се улаз представи у бинарном облику.

Време извршавања алгоритма за конкретан улаз је број примитивних операција или "корака" који се извршавају. Уобичајено је да се појам корака дефинише тако да што мање зависи од конкретног рачунара. У нашим примерима применићемо следећи приступ. За извршавање сваке линије псеудокода потребно је константно време. Извршавање једне линије захтева другачије време од извршавања друге линије, али ћемо претпоставити да извршавање i-те линије траје c_i, где је c_i константа.

У наставку ће израз за време извршавања алгоритма *SortiranjeUmetanjem* еволуирати из компликоване формуле која користи све константне цене c_i у много једноставнији израз коришћењем једноставније нотације. Та једноставнија нотација ће такође омогућити утврђивање да ли је један алгоритам ефикаснији од другог.

Најпре процедуру *SortiranjeUmetanjem* допуњујемо оценама трајања инструкција (линија). За свако $j = 2, 3, \ldots n$ нека t_j означава број колико пута је извршен тест у **while** петљи у линији 5 за ту вредност j. Коментари се не извршавају, па не троше време.

SortiranjeUmetanjem(A, n)	цена	број понављања
1 **for** $j \leftarrow 2$ **to** n **do**	c_1	n
2 $\quad t \leftarrow A[j]$	c_2	$n-1$
3 $\quad$ {уметнути $A[j]$ у сортирани низ $A[1..j-1]$}	0	$n-1$
4 $\quad i \leftarrow j-1$	c_4	$n-1$
5 $\quad$ **while** $i > 0$ **and** $A[i] > t$ **do**	c_5	$\sum_{j=2}^{n} t_j$
6 $\quad\quad A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $\quad\quad i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $\quad A[i+1] \leftarrow t$	c_8	$n-1$

Време извршавања алгоритма је збир времена извршавања свих инструкција. Ако се нека инструција састоји од c_i корака, а извршава се n пута, онда је њен допринос укупном времену $c_i n$. Да бисмо израчунали $T(n)$, време извршавања алгоритам *SortiranjeUmetanjem*, треба да саберемо производе у колонама *цена* и *број понављања*, па је

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j-1) + \\ &+ c_7\sum_{j=2}^{n}(t_j-1) + c_8(n-1). \end{aligned}$$

Чак и за улазе задате величине, време извршавања алгоритма може да зависи од тога *који* улаз те величине је задат. На пример, за алгоритам *SortiranjeUmetanjem* најбољи случај је кад је низ већ сортиран. За свако $j = 2, 3, \ldots, n$ испоставља се да је $A[i] \leq t$ у линији 5 тачно кад i има своју почетну вредност $j-1$. Према томе, $t_j = 1$ за $j = 2, 3, \ldots, n$, па је време извршавања у најбољем случају

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

Ово време извршавања може се изразити у облику $an+b$ за неке константе a и b, које зависе од цена наредби c_j; време извршавања је дакле **линеарна функција** од n.

Ако је низ на почетку опадајући, добија се најгори случај. Елемент $A[j]$ мора се упоредити са сваким елементом подниза $A[1..j-1]$, па је $t_j = j$ за $j = 2, 3, \ldots, n$. Пошто је $\sum_{j=2}^{n} j = n(n+1)/2 - 1$ и $\sum_{j=2}^{n}(j-1) = n(n-1)/2$, добијамо да је у најгорем случају време извршавања алгоритма *SortiranjeUmetanjem* једнако

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &+ c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad -(c_2 + c_4 + c_5 + c_8). \end{aligned}$$

Ово време извршавања може се изразити у облику $an^2 + bn + c$ за неке константе a, b и c које зависе од цена наредби c_j; време извршавања је дакле **квадратна функција** од n.

1.4.2 Анализа најгорег и просечног случаја

У спроведеној анализи сортирања уметањем разматрали смо и најбољи случај, кад је низ већ сортиран, и најгори случај, кад је низ сортиран обрнутим редоследом, опадајуће. Приликом анализе алгоритама обично се разматра само **време извршавања у најгорем случају**. Постоји неколико аргумената за овакав приступ.

- Време извршавања алгоритма у најгорем случају је горња граница за време извршавања алгоритма за било који улаз исте величине. Знајући то, имамо гаранцију да извршавање алгоритма никада неће трајати више од процењеног времена.
- За неке алгоритме се најгори случај дешава прилично често. На пример, приликом тражења конкретног податка у бази података, најгори случај је кад тог податка нема у бази. У неким применама претраживања база података тражење одсутног податка може да буде чест случај.
- "Просечан случај" је често исто толико лош колико и најгори случај. Претпоставимо да на случајан начин изаберемо n бројева и применимо на њих сортирање уметањем. Колико треба времена да се одреди где је у поднизу $A[1..j-1]$ место за уметање елемента $A[j]$? У просеку је пола елемената $A[1..j-1]$ мање од $A[j]$, а пола је веће од њега. У просеку треба проверити пола чланова низа $A[1..j-1]$, па је $t_j = j/2$. Ако ову анализу просечног случаја спроведемо до краја, добијамо да је просечно време извршавања такође квадратна функија од n, као и у најгорем случају.

1.4.3 Брзина раста

Приликом анализе сортирања уметањем користили смо нека упрошћавања заснована на уопштавању. Најпре, занемаривали смо стварне цене појединих инструкција, коришћењем константи c_i за представљање њихових цена. Затим смо запазили да нам те константе пружају више детаља него што нам је потребно: време извршавања је $an^2 + bn + c$ за неке константе a, b и c које зависе од цена инструкција c_i. Према томе, ми смо игнорисали не само стварне цене инструкција, него и апстрактне цене c_i.

Сада ћемо учинити следеће упрошћење засновано на уопштавању. Ради се о томе да нас занима само **брзина раста** односно **поредак раста** времена извршавања. Због тога ми разматрамо само водећи члан (нпр. an^2), јер чланови нижег реда постају занемарљиви за велике вредности n. Према томе, ми кажемо да, на пример, сортирање уметањем има време извршавања $\Theta(n^2)$ у најгорем случају. Ову ознаку користићемо неформално, иако је лако дефинисати је и прецизно. Обично сматрамо један алгоритам ефикаснијим од другог ако његово време извршавања има мању брзину раста. Због константних фактора и занемарених чланова нижег реда, ова

процена може бити погрешна за мале n. Међутим, за довољно велике улазе се на пример $\Theta(n)$ алгоритам извршава брже од $\Theta(n^2)$ алгоритма.

1.5 Конструкција алгоритама применом разлагања

Постоји више техника за конструкцију алгоритама. Код сортирања уметањем коришћен је **инкрементални** приступ: полазећи од сортираног подниза $A[1..j-1]$, ми умећемо наредни елеменат $A[j]$ на одговарајуће место, и тако долазимо до сортираног подниза $A[1..j]$.

Сада ћемо размотрити други приступ конструкцији алгоритама, заснован на разлагању улазних података (енг. divide-and-conquer). Користећи овај приступ, долази се до алгоритма за сортирање чије је време извршавања много мање него за сортирање уметањем. Алгоритми засновани на овом приступу имају предност да се њихово време извршавања лако одређује у веома општем случају.

Многи корисни алгоритми имају **рекурзивну** структуру: да би решили дати проблем, они позивају себе да би обрадили блиско повезане потпроблеме. Такви алгоритми често користе приступ заснован на **разлагању**: они разбијају проблем на неколико сличних, али мањих потпроблема, решавају их рекурзивно, а онда комбинују ова решења да би формирали решење полазног проблема.

Приступ заснован на разлагању укључује три корака на сваком нивоу рекурзије:

Разлагање проблема на неколико мањих потпроблема.

Рекурзивно **решавање потпроблема**. У случају кад је величина потпроблема довољно мала, потпроблем се решава директно, тј. излази се из рекурзије.

Комбиновање решења потпроблема да би се добило решење полазног проблема.

Алгоритам **сортирање обједињавањем** директно реализује ову технику. Интуитивно, он се извршава на следећи начин.

Разлагање: Низ дужине n који треба сортирати дели се на два подниза дужине по $n/2$.

Решавање потпроблема: Два подниза се рекурзивно сортирају применом истог алгоритма.

Комбиновање: Два добијена сортирана подниза се обједињавају, и тако се добија сортирани полазни низ.

Из рекурзије се излази кад поднизови које треба сортирати имају дужину 1: такви низови су већ сортирани.

Кључна операција у овом алгоритму је обједињавање два сортирана низа у кораку "комбиновање". Да бисмо извршили обједињавање, користимо помоћну процедуру $Objedinjavanje(A, p, q, r)$, где је A низ, а p, q и r су индекси чланова низа A, такви да је $p \leq q < r$. Процедура претпоставља да су поднизови $A[p..q]$ и $A[q+1..r]$ већ сортирани. Процедура их **обједињује** у један сортирани подниз, који замењује текући подниз $A[p..r]$.

Процедура *Objedinjavanje* извршава се за време $\Theta(n)$, где је $n = r-p+1$ број елемената који се обједињавају, и ради на следећи начин. Враћајући се на мотив са картама, претпоставимо да су на столу две гомиле карата лицем окренутим навише. Обе гомиле су сортиране, тако да је најмања карта на врху. Наш циљ је да објединимо две гомиле у једну сортирану излазну гомилу, у којој ће карте бити окренуте лицем наниже. Основни корак је избор мање од две карте на врху гомила, скидање те карте са гомиле (чиме се открива наредна карта на истој гомили) и њено пребацивање лицем наниже на излазну гомилу. Основни корак се понавља све док се не испразни једна од улазних гомила; тада се друга улазна гомила пребацује лицем наниже на излазну гомилу. За извршавање сваког оваквог корака потребно је константно време, јер се упоређују само две карте на врху гомила. Пошто је број основних корака који се извршавају највише n, време извршавања обједињавања је $\Theta(n)$.

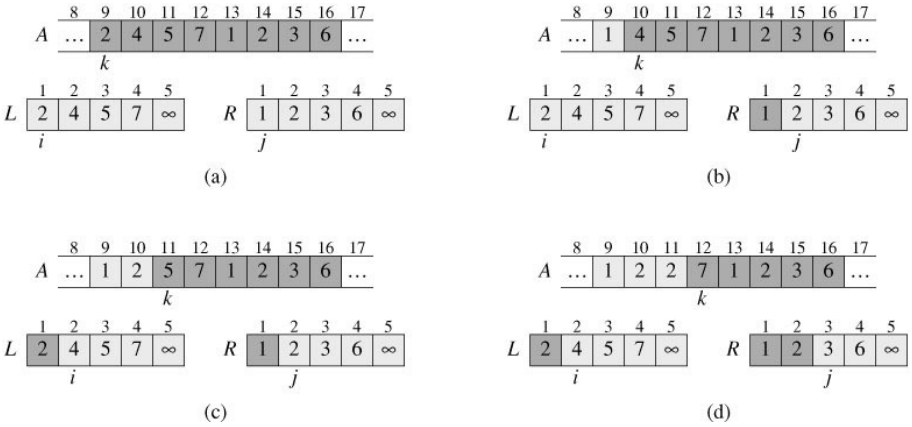
Наредни псеудокод реализује горњу идеју, уз допунски трик који омогућује да се избегне потреба да се у сваком основном кораку проверава да ли је једна од гомила испражњена. Идеја је да се на дно обе гомиле карата стави **граничник**, карта која садржи специјалну вредност, коју користимо да бисмо упростили код. Овде као вредност граничника користимо ∞, тако да кад је та карта на врху гомиле, она никад не може бити мања од друге карте, сем ако се и на врху друге гомиле налази та вредност. Али када се то деси, све остале карте сем граничника су већ пребачене на излазну гомилу. Пошто ми унапред знамо да на излазну гомилу треба пребацити тачно $r - p + 1$ карата, заустављамо се у тренутку кад извршимо толико основних корака.

```
Objedinjavanje(A, p, q, r)
1   n1 ← q − p + 1
2   n2 ← r − q
3   креирати низове L[1..n1 + 1] и D[1..n2 + 1]
4   for i ← 1 to n1 do
5     L[i] ← A[p + i − 1]
6   for j ← 1 to n2 do
7     D[j] ← A[q + j]
8   L[n1 + 1] ← ∞
9   D[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r do
13    if L[i] ≤ D[j] then
14      A[k] ← L[i]
15      i ← i + 1
16    else A[k] ← D[j]
17      j ← j + 1
```

Процедура *Objedinjavanje* ради на следећи начин. У линији 1 израчунава се дужина n_1 подниза $A[p..q]$, а у линији 2 дужина n_2 подниза $A[q + 1..r]$. Затим се креирају низови L и D ("леви" и "десни"), дужина n_1+1, односно n_2+1, у линији 3. Петља **for** у линијама $4-5$ копира подниз

$A[p..q]$ у $L[1..n_1]$, а петља **for** у линијама $6-7$ копира подниз $A[q+1..r]$ у $D[1..n_2]$. Линије $8-9$ уписују граничнике у низове L и D. Линије $10-17$, као што је приказано на сликама 2.3 и 1.4 извршавају $r-p+1$ основних корака, одржавајући следећу инваријанту петље:

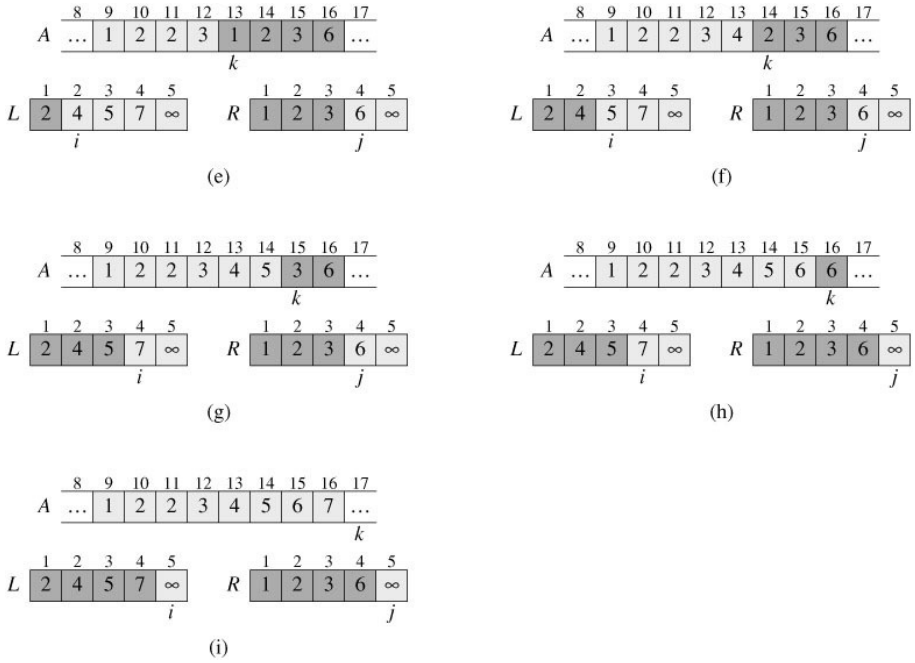
На почетку сваког проласка кроз **for** петљу у линијама $12-17$ подниз $A[p..k-1]$ садржи $k-p$ најмањих елемената низова $L[1..n_1+1]$ и $D[1..n_2+1]$, и то у сортираном поретку. Поред тога, $L[i]$ и $D[j]$ су најмањи елементи у одговарајућим низовима међу елементима који још нису копирани назад у A.



Слика 1.3: Извршавање линија $10-17$ приликом позива $Objedinjavanje(A, 9, 12, 16)$ када подниз $A[9..16]$ садржи низ $(2, 4, 5, 7, 1, 2, 33, 6)$. После копирања и уметања граничника садржај низа L је $(2, 4, 5, 7, \infty)$, а садржај низа D је $(1, 2, 3, 6, \infty)$. Светло обојени елементи низа A садрже своје коначне вредности које треба копирати назад у A. Посматране заједно, светло обојене позиције у сваком тренутку садрже вредности које су на почетку биле у $A[9..16]$, заједно са два граничника. Затамњене позиције у A садрже вредности које ће бити преписане новим садржајем, а затамњене позиције у L и D садрже вредности које су већ биле прекопиране назад у A. $(a)-(h)$ Низови A, L и D и одговарајући индекси k, i и j пре сваког проласка кроз петљу у линијама $12-17$. (j) Низови и индекси приликом завршетка. У том тренутку је подниз $A[9..16]$ сортиран, а два граничника у L и D у једини елементи у овим низовима који још нису копирани назад у A.

Показаћемо сада да је инваријанта петље тачна пре првог проласка кроз **for** петљу у линијама $12-17$, да инваријанта остаје тачна после сваког проласка кроз петљу, и да је последица инваријанте тврђење из кога следи коректност алгоритма кад се петља заврши.

Иницијализација: Пре првог проласка кроз петљу је $k=p$, па је подниз $A[p..k-1]$ празан. Овај празан подниз садржи $k-p=0$ најмањих елемената из L и D, па пошто је $i=j=1$, $L[i]$ и $D[j]$ су најмањи елементи у својим низовима који нису били копирани назад у A. Тврђење је тачно пре првог проласка кроз петљу.



Слика 1.4: Извршавање линија $10-17$ приликом позива $Objedinjavanje(A, 9, 12, 16)$, други део.

Одржавање: Да бисмо се уверили да сваки пролазак кроз петљу одржава инваријанту петље, претпоставимо најпре да је $L[i] \leq D[j]$. Тада је $L[i]$ најмањи елеменат међу онима који још нису копирани назад у A. Пошто $A[p..k-1]$ садржи $k-p$ најмањих елемената, када се у линији 14 $L[i]$ прекопира у $A[k]$, подниз $A[p..k]$ садржаће $k-p+1$ најмањих елемената. Повећавање за 1 вредности k (приликом повратка у **for** петљи) и j (у линији 15) обнавља тачност инваријанте петље за следећи пролазак. Ако је пак $L[i] > D[j]$, онда линије $16-17$ извршавају оно што је неопходно да се одржи инваријанта петље.

Завршетак: На крају је $k = r+1$. На основу инваријанте петље подниз $A[p..k-1]$, што је уствари $A[p..r]$, садржи $k-p = r-p+1$ најмањих елемената из $L[1..n_1+1]$ и $D[1..n_2+1]$, и то у сортираном поретку. Низови L и D садрже заједно $n_1+n_2 = r-p+3$ елемената. Сви сем два од њих копирају се назад у A, а та два највећа елемента су граничници.

Да бисмо се уверили да је трајање процедуре *Objedinjavanje* $\Theta(n)$, где је $n = r-p+1$, приметимо да се линије $1-3$ и $8-11$ извршавају за константно време, да извршавање **for** петљи у линијама $4-7$ траје $\Theta(n_1+n_2) = \Theta(n)$, као и да се кроз **for** петљу у линијама $12-17$ пролази n пута, при чему сваки пролазак троши константно време.

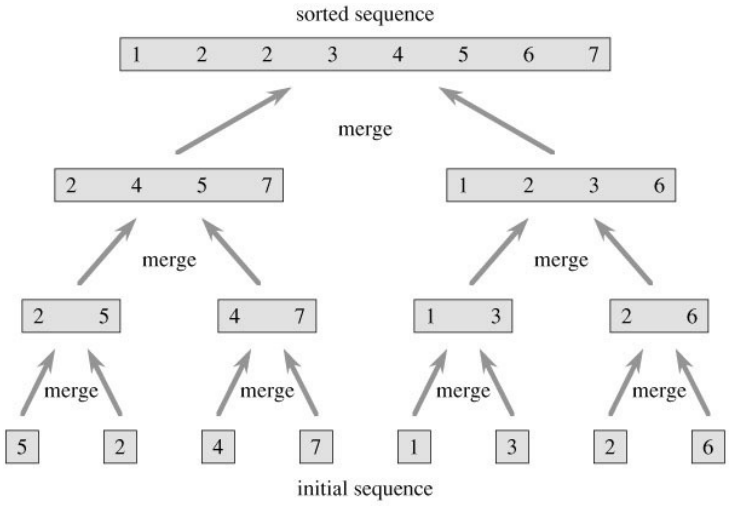
Сада можемо да искористимо процедуру *Objedinjavanje* као потпрограм у алгоритму за сортирање обједињавањем. Процедура

$$SortiranjeObjedinjavanjem(A, p, r)$$

сортира елементе у поднизу $A[p..r]$. Ако је $p \geq r$ онда подниз има највише један елемент, па је према томе већ сортиран. У противном, корак разлагања једноставно израчунава индекс q, који раздваја $A[p..r]$ у два подниза: $A[p..q]$ са $\lceil n/2 \rceil$ елемената, и $A[q+1..r]$ са $\lfloor n/2 \rfloor$ елемената. Овде $\lceil x \rceil$, односно $\lfloor x \rfloor$ означавају редом најмањи цели број већи или једнак од x, односно највећи цели број мањи или једнак од x.

SortiranjeObjedinjavanjem(A, p, r)
1 **if** $p < r$ **then**
2 $q \leftarrow \lfloor (p+r)/2 \rfloor$
3 *SortiranjeObjedinjavanjem*(A, p, q)
4 *SortiranjeObjedinjavanjem*$(A, q+1, r)$
5 *Objedinjavanje*(A, p, q, r)

Да бисмо извршили сортирање целог низа $A = (A[1], A[2], \ldots, A[n])$, позивамо *SortiranjeObjedinjavanjem*$(A, 1, n)$. Слика 2.4 илуструје извршавање процедуре одоздо навише када је n степен двојке. У алгоритму се обједињавају парови 1-чланих поднизова у двочлане поднизове, затим се обједињавају парови поднизова дужине 2 у поднизове дужине 4, и тако даље, све док се на крају два подниза дужине $n/2$ не обједине у низ дужине n.



Слика 1.5: Резултат примене сортирања обједињавањем на низ $A = (5, 2, 4, 7, 1, 3, 2, 6)$. Дужине сортираних поднизова који се обједињавају расту са напредовањем алгоритма одоздо навише.

1.6 Анализа алгоритама заснованих на разлагању

Када алгоритам садржи рекурзивни позив самог себе, његово време извршавања се често може описати **рекурентном релацијом**, која изражава укупно време извршавања на проблему са улазом величине n преко времена

извршавања на мањим улазима. Тада се овакве рекурентне релације могу решити да би се добила граница за ефикасност алгоритма.

Рекурентна релација за време извршавања алгоритма заснованог на разлагању одређена је са три основна корака оваквог алгоритма. Нека, као и раније, $T(n)$ означава време извршавања на улазу величине n. Ако је величина улаза довољно мала, нпр. $n \leq c$ за неку константу c, директно решавање траје константно време, што се може написати као $\Theta(1)$. Претпоставимо да се разлагањем улаза добија a потпроблема, од којих је сваки b пута мањи од полазног. (У алгоритму сортирања обједињавањем је $a = b = 2$, али постоје примери оваквих алгоритама у којима је $a \neq b$). Ако са $D(n)$ означимо време потребно да се улаз разложи, а са $C(n)$ да се комбинују решења потпроблема у решење полазног проблема, добијамо рекурентну релацију

$$T(n) = \begin{cases} \Theta(1), & n \leq c, \\ aT(n/b) + D(n) + C(n), & n > c. \end{cases}$$

1.6.1 Анализа сортирања обједињавањем

Иако псеудокод за сортирање обједињавање исправно ради и ако дужина низа није парна, анализа решавањем рекурентне релације се поједностављује ако се претпостави да је почетна дужина низа степен двојке.

Сада ћемо извести рекурентну релацију за $T(n)$, оцену времена извршавања сортирања обједињавањем у најгорем случају. Сортирање једночланог низа траје константно време. Ако треба сортирати $n > 1$ елемената, време извршавања разлажемо на следећи начин.

Разлагање: Овај корак израчунава средину низа, па се извршава за константно време. Дакле, $D(n) = \Theta(1)$.

Решавање потпроблема: Два подниза дужине $n/2$ се рекурзивно сортирају применом истог алгоритма, што траје $2T(n/2)$.

Комбиновање: Видели смо да извршавање процедуре *Objedinjavanje* на подниз дужине n траје $\Theta(n)$, па је $C(n) = \Theta(n)$.

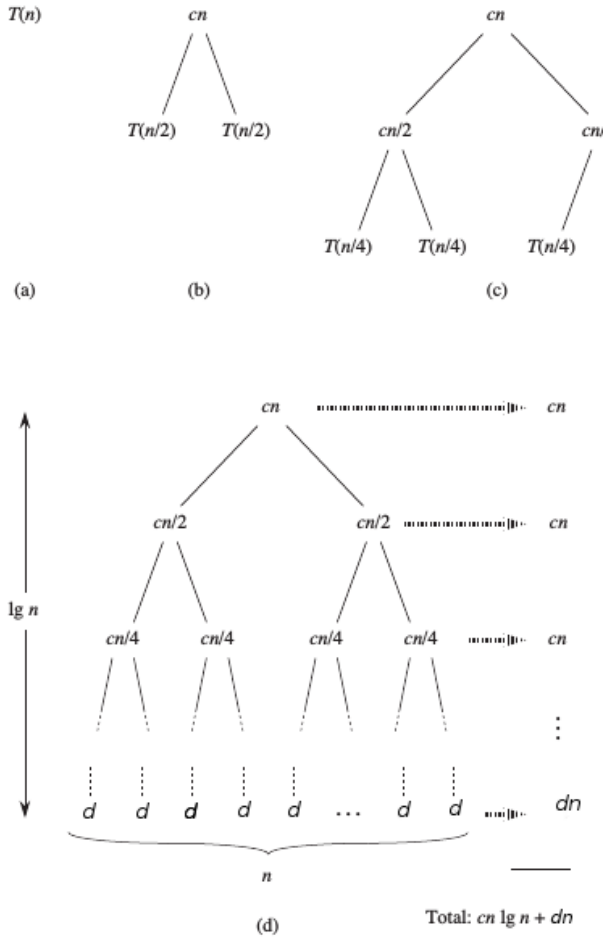
Сабирајући функције $D(n)$ и $C(n)$ приликом анализе сортирања обједињавањем, ми сабирамо једну функцију која је $\Theta(n)$ са једном функцијом која је $\Theta(1)$. Збир је линеарна функција од n, односно $\Theta(n)$. Према томе, тражена рекурентна релација је

$$T(n) = \begin{cases} d, & n = 1, \\ 2T(n/2) + cn, & n > 1. \end{cases}$$

где је c утрошено време по елементу низа у корацима разлагања и комбиновања.

Слика 2.5 показује како се може решити ова рекурентна релација. Због једноставности, претпоставимо да је n степен двојке. Део (a) слике приказује $T(n)$, израз који је у делу (b) развијен у еквивалентно стабло које представља рекурентну релацију. Члан cn је у корену (трајање највишег нивоа рекурзије), а два подстабла су трајања два мања рекурентна позива $T(n/2)$. Део (c) показује резултат примене истог поступка после развоја $T(n/2)$.

Трајање сваког чвора на другом нивоу рекурзије је $cn/2$ Настављамо са развијањем сваког чвора стабла његовим разбијањем на саставне делове одређене рекурентном релацијом, све док величине поднизова не дођу до 1. Део (d) показује коначно стабло.



Слика 1.6: Конструкција стабла рекурзије за рекурентну релацију $T(n) = 2T(n/2) + cn$. Део (a) приказује $T(n)$ који се прогресивно разлаже у деловима $(b) - (d)$ да би се добило стабло рекурзије. Потпуно развијено стабло рекурзије у делу (d) има $\log n + 1$ нивоа (односно има висину $\log n$, као што је назначено), а збир времена у свим чворовима на истом нивоу је cn. Укупна сума је, према томе, $cn \log n + dn$, што је $\Theta(n \log n)$.

После тога ми сабирамо цене на сваком нивоу стабла. Највиши ниво има цену cn; следећи ниво има цену $c(n/2) + c(n/2) = cn$ следећи ниво има цену $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$, итд. Уопште, ниво i испод корена има 2^i чворова са ценом $c(n/2^i)$, па је укупна цена свих чворова на i-том нивоу испод врха једнака $2^i c(n/2^i) = cn$. На најнижем нивоу има n чворова са ценом d, па је њихова укупна цена dn.

Укупан број нивоа у "стаблу рекурзије" на слици 2.4 је $\log n + 1$, што се

лако показује индукцијом. Да бисмо добили решење рекурентне релације, треба да саберемо цене на свим нивоима овог стабла. Укупно има $\log n$ нивоа са ценом по cn, и један ниво са ценом dn па је укупна цена $cn \log n + dn$. Занемарујући члан нижег реда и константу c, добијамо резултат $T(n) = \Theta(n \log n)$.

1.7 Задаци за вежбу

1. Навести неке проблеме из реалног света који захтевају сортирање.
2. Поред брзине, које би се додатне мере ефикасности алгоритма могле користити?
3. Претпоставимо да поредимо имплементацију сортирања уметањем и сортирања обједињавањем на истом рачунару. За улаз величине n сортирање уметањем ради у $8n^2$ корака, а сортирање обједињавањем у $64n \log n$ корака. За које n ће сортирање уметањем радити брже од сортирања обједињавањем?
4. За сваку функцију $f(n)$ и време t одредити максимално n за које се проблем може решити у времену t, под претпоставком да је време извршавања алгоритма једнако $f(n)$ наносекунди.

	1 секунд	1 минут	1 сат	1 дан	1 месец	1 год	1 век
$\log n$							
$\sqrt{n}$							
n							
$n \log n$							
n^2							
n^3							
2^n							
$n!$							

5. Демонстрирати примену сортирања уметањем на низ $\{31, 41, 59, 26, 41, 58\}$
6. Изменити псеудокод алгоритма за сортирање уметањем тако да се добије нерастући поредак уместо неопадајућег.
7. Демонстрирати примену сортирања обједињавањем на низ $\{3, 41, 52, 26, 38, 57, 9, 49\}$
8. Преправити алгоритам за сортирање обједињавањем тако да не користи граничник, већ да стаје када један од низова дође до краја и да се онда преостали елементи ископирају у резултујући низ A.
9. Математичком индукцијом доказати да је за $n = 2^k$ решење рекурентне једначине:

$$T(n) = \begin{cases} 2, & n = 2 \\ 2T(n/2) + n, & n = 2^k, k > 1. \end{cases}$$

једнако $T(n) = n \log n$.

10. Написати код алгоритма за сортирање избором (енгл. selection sort). Израчунати време извршавања овог алгоритма и доказати његову коректност методом инваријанте петље.

SortiranjeIzborom(A, n)
улаз: A - низ бројева, n - димензија низа
излаз: низ A сортиран неопадајуће
1 **for** $i \leftarrow 1$ **to** $n-1$ **do**
2 {на место $A[i]$ довести најмањи елемент низа $A[i..n]$}
3 $indeks \leftarrow i$
4 **for** $j \leftarrow i+1$ **to** n **do**
5 **if** $A[j] < A[indeks]$ **then**
6 $indeks \leftarrow j$
7 {$indeks$ садржи индекс најмањег елемента низа $A[i..n]$}
8 замени$(A[i], A[indeks])$

Време извршавања алоритма израчунаћемо као збир времена извршавања свих инструкција. За линију i претпоставићемо да је њено време извршавања c_i и израчунаћемо колико се пута она извршава у најгорем случају.

SortiranjeIzborom(A, n)	цена	бр пон
улаз: A - низ бројева, n - димензија низа		
излаз: низ A сортиран неопадајуће		
1 **for** $i \leftarrow 1$ **to** $n-1$ **do**	c_1	n
2 {на место $A[i]$ довести најмањи елемент низа $A[i..n]$}	0	$n-1$
3 $indeks \leftarrow i$	c_3	$n-1$
4 **for** $j \leftarrow i+1$ **to** n **do**	c_4	$\sum_{i=1}^{n-1}(n-i+1)$
5 **if** $A[j] < A[indeks]$ **then**	c_5	$\sum_{i=1}^{n-1}(n-i)$
6 $indeks \leftarrow j$	c_6	$\sum_{i=1}^{n-1} t_i$
7 {$indeks$ садржи индекс најмањег елемента низа $A[i..n]$}	0	$n-1$
8 замени$(A[i], A[indeks])$	c_8	$n-1$

Ако је t_i број извршавања линије 4 за фиксирано i, онда је у најгорем случају $t_i = n - i$: то је случај када се увек врши измена вредности индекса.

Време извршавања у најгорем случају је једнако

$T(n) = nc_1 + (n-1)c_3 + \sum_{i=1}^{n-1}(n-i)(c_4+c_5+c_6) + (n-1)c_4 + (n-1)c_8 = O(n^2)$

тј. време је квадратна функција величине улаза n.

За вежбу: израчунати вредност горње суме.

Доказ коректности алгоритма:

За инваријану петље можемо узети следеће својство: на почетку сваког проласка кроз **for** петљу на позицијама $A[1..i-1]$ налазе се најмањих $i-1$ елемената низа A и то у сортираном поретку.

Иницијализација: пре првог проласка кроз петљу, важи да је $i = 1$, те је подниз празан. Овај празан подниз садржи најмањих $i - 1 = 0$ елемената низа, те тврђење важи пре првог проласка кроз петљу.

Одржавање: треба показати да сваки наредни пролазак кроз петљу одржава тачност инваријанте петље. Према претпоставци на позицијама $A[1..i-2]$ налазе се најмањих $i-2$ елемената низа у сортираном поретку. Након тога, у наредној итерацији петље се међу елементима на позицијама $A[i-1..n]$ тражи индекс минималног елемента и он се смешта на позицију $A[i-1]$. Пошто се на позицијама $A[1..i-2]$ налазе најмањих $i-2$ елемената, низа а на позицији i се налази минимални међу преосталим елементима низа, можемо закључити да се након ове итерације петље на позицијама $A[1..i-1]$ налазе најмањих $i-1$ елемената низа. Да ли су они поређани у растућем поретку? Да би ово важило потребно је доказати да важи да се на позицији $i-1$ налази број који није мањи ни од једног од бројева из низа $A[1..i-2]$. То је тачно јер се на позицијама $A[1..i-2]$ налазе најмањих $i-2$ елемената низа па је елемент на позицији $i-1$ сигурно није мањи од њих.

Завршетак: По завршетку петље важи да је $i = n + 1$. На основу тачности инваријанте петље закључује се да се на позицијама $A[1..n]$ налазе најмањих n елемената низа A сортирани у неопадајућем поретку, а то је цео сортиран низ A.

11. Како у времену $O(n \log n)$ установити да ли у низу A димензије n има поновљених елемената.

Глава 2

Структуре података

Структуре података су основни елементи од којих се граде алгоритми. Да бисмо владали техникама контрукције алгоритама, неопходно је добро познавање структура података, у смислу техника рада са њима, као и познавање сложености операција са њима. У програмима се обично тачно задаје тип сваког податка (да ли је у питању цео или реалан број и слично); међутим при конструкцији алгоритма конкретан тип података обично није битан. На пример, ако алгоритам ради са листом, од интереса су операције уметања елемента у листу, односно избацивања елемента из листе и некад је корисно да се не прецизира тип података. У том случају говоримо о **апстрактном типу података**. Најважнија карактеристика апстрактног типа података је списак операција које се над њим могу извршавати. Концентришући пажњу на природу структура података са аспекта списка потребних операција, а занемарујући конкретну реализацију, добија се општији алгоритам и конструкција алгоритма постаје модуларнија (програм се састоји од међусобно мало зависних целина).

2.1 Примитивне структуре података

Под **елементом** подразумевамо податак коме тип није прецизиран. Тако, на пример, ако се ради о проблему сортирања, 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.

Низ или **вектор** је низ елемената истог типа на узастопним локацијама у меморији. Величина низа је број елемената у њему. Величина низа се мора унапред задати, односно фиксирана је, јер се унапред зна величина додељене меморије. На пример, низ од 100 елемената од којих се сваки елемент записује у 8 бајтова заузима у меморији 800 бајтова. Ако је први елемент на локацији x, а величина елемента је a бајтова, онда се k-ти елемент низа налази на локацији $x + (k-1)a$ и ова вредност се може директно израчунати. Основна особина ове структуре података јесте једнако време приступа свим елементима. Главни недостатак низа је тај што се

његова величина не може мењати динамички, у току извршавања алгоритма. Такође, некада је потребно уметнути или избацити елемент из средине, што је код низа неефикасно.

Структура (**слог** или **рекорд**) је такође низ елемената који, међутим, не морају бити истог типа – једино је битно да се фиксира комбинација типова елемената. Величина структуре такође мора бити унапред дефинисана. И елементима структуре се може приступати у константном времену: у ове сврхе користи се посебан низ са почетним адресама елемената структуре, дужине једнаке броју елемената структуре.

2.1.1 Динамички скупови

Као и у математици, и у рачунарству се скупови сматрају основним појмом. Док се математички скупови не мењају, скупови који се користе у алгоритмима могу да имају променљиву величину – могу да се повећавају, да се смањују или да се мењају током времена. Овакве скупове зваћемо **динамичким скуповима**. Наиме, у алгоритмима је често потребно да се број елемената у структури података динамички мења. Када се унапред не зна број елемената низа могуће је резервисати довољно велики простор и тако решити проблем. Ипак, неефикасно је резервисати меморијски простор према најгорем случају. Поред тога, некада је потребно уметнути или обрисати елемент из средине – што је неефикасно код низа. Размотрићемо сада основне технике за представљање динамичких скупова и њихову манипулацију на рачунару.

У алгоритмима се извршавају различите операције са скуповима. Типичне операције са скуповима су додавање елемента, брисање елемента из скупа и испитивање присуства елемента у скупу. Овакав динамички скуп зваћемо **речник**. Понекад се очекује да динамички скуп обезбеђује операције за додавање новог елемента и за издвајање најмањег елемента из скупа. Скуп са овакве две операције зове се **ред са приоритетом**. Наравно, реализација динамичког скупа зависи од операција које треба да буду подржане.

У типичној имплементацији динамичког скупа, сваки елемент је представљен објектом чијим се пољима може приступити уколико имамо показивач на објекат. Неке врсте динамичких скупова претпостављају да је једно од поља **кључ**. Ако су сви кључеви међусобно различити, можемо на динамички скуп гледати као на скуп кључева. Објекат може да садржи и **сателитске податке**, који се налазе у другим пољима објекта, али се не користе у имплементацији скупа. Такође, објекат може да има и поља која користе операције над скуповима: ова поља могу да садрже податке или показиваче на друге објекте у скупу.

Неки динамички скупови претпостављају да се кључеви узимају из потпуно уређеног скупа, као што је нпр. скуп реалних бројева. Ово нам омогућава да дефинишемо минимални елемент скупа, или да говоримо о наредном већем елементу од датог елемента у скупу.

2.1.2 Операције са динамичким скуповима

Операције над динамичким скуповима се могу груписати у две категорије:

- **очитавања** – операције које враћају неку информацију о скупу,
- **промене** – операције које мењају скуп.

Следи листа типичних операција над динамичким скуповима. Свака конкретна примена обично захтева имплементацију само неких од наредних операција:

Nadji(S,k) за дати скуп S и кључ k враћа се показивач x на елемент скупа S за који важи да је $x.kljuc = k$ или **NULL** ако такав елемент не припада скупу S

Umetni(S,x) операција измене скупа којом се скуп S проширује елементом на који показује показивач x

Izbaci(S,x) операција измене скупа којом се из скупа S избацује елемент на који показује показивач x

Minimum(S) за дати потпуно уређени скуп S враћа се показивач на елемент скупа S са најмањом вредношћу кључа

Maximum(S) за дати потпуно уређени скуп S враћа се показивач на елемент скупа S са највећом вредношћу кључа

Sledbenik(S,x) за дати потпуно уређени скуп S и показивач на елемент x тог скупа враћа се показивач на први већи елемент у скупу S или **NULL** уколико је x максимални елемент

Prethodnik(S,x) за дати потпуно уређени скуп S и показивач на елемент x тог скупа враћа се показивач на први мањи елемент у скупу S или **NULL** уколико је x минимални елемент

У неким ситуацијама можемо проширити операције Sledbenik и Prethodnik тако да раде и над скуповима са поновљеним кључевима. За скуп од n кључева, уобичајена претпоставка јесте да позив функције Minimum за којом следи $n-1$ позива функције Sledbenik даје списак елемената скупа S уређен растуће.

Уобичајено је да се време извршавања скуповних операција изражава преко броја елемената скупа.

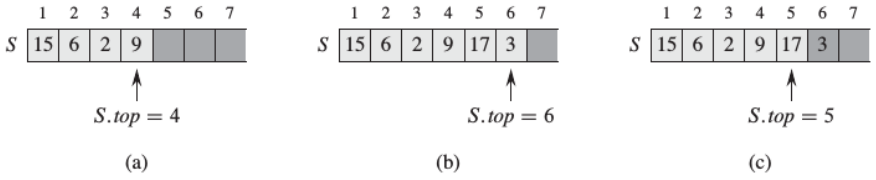
2.2 Стек и ред

Стек и ред су динамички скупови у којима је предодређен (предефинисан) елемент који се брише (избацује) операцијом **Izbaci**. Код **стека**, елемент који се брише из скупа је онај који је последњи додат: стек имплементира **last-in, first-out** или политику **LIFO**. Слично, код **реда** елемент који се брише је увек онај који је у скупу најдуже: ред имплементира **first-in, first-out** или политику **FIFO**. Показаћемо како се ове две структуре података могу имплементирати коришћењем низа.

2.2.1 Стек

Операција уметања у стек се уобичајено назива **Push**, а операција избацивања која нема као аргумент елемент који треба избацити **Pop**. Ови називи су настали у аналогији са физичким стековима, као што су, на пример, тацне у кафићу наређане једна на другу или гомила карата. Редослед у ком се тацне узимају са гомиле је обрнут редоследу у ком су стављане на гомилу, с обзиром на то да је доступна само тацна са врха. Слично је и са гомилом карата.

Као што се може видети на слици 2.1, стек од n елемената можемо имплементирати коришћењем низа $S[1..n]$. Уз низ чува се и поље vrh (енгл. *top*) које индексира најскорије уметнути елемент. Стек се састоји од елемената $S[1..vrh]$, при чему је $S[1]$ елемент на дну стека, а $S[vrh]$ елемент са врха стека.



Слика 2.1: Имплементација стека преко низа. Елементи стека су обојени светло сивом бојом. (a) Стек има 4 елемента. Елемент на врху је 9. (b) Стек након позива Push(S,17) и Push(S,3). (c) Стек након позива Pop(S) враћа елемент 3, који је последњи уметнут. Иако се елемент 3 и даље налази у низу, он више није на стеку; на врху стека се сада налази елемент 17.

Када је $vrh = 0$, стек не садржи елементе и кажемо да је **празан**. Да ли је стек празан можемо утврдити коришћењем операције **Stek_prazan**. Ако покушамо да избацимо елемент из празног стека, долази до грешке која се зове **поткорачење**, што проузрокује грешку. Ако vrh премаши вредност n, долази до **прекорачења** (у имплементацији која је дата у псеудокоду, није укључена провера прекорачења стека).

Сваку од ових операција са стеком можемо имплементирати у само неколико линија кода:

```
Stek_prazan(S, vrh)
1 if vrh = 0 then
2    return TRUE
3 else return FALSE
```

```
Push(S, vrh, x)
1 vrh ← vrh + 1
2 S[vrh] ← x
```

```
Pop(S, vrh)
1 if Stek_prazan(S) then
2    error"potkoracenje"
3 else
4    vrh ← vrh − 1
5    return S[vrh + 1]
```

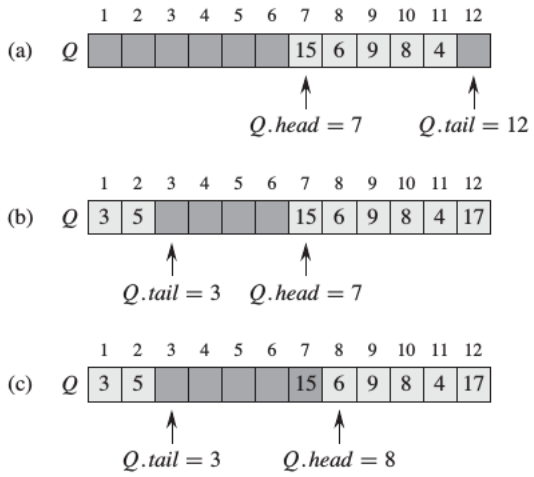
2.2.2 Ред

Операцију уметања у ред зовемо **Enqueue**, а операцију избацивања из реда **Dequeue**. Као што је случај са операцијом Pop код стека, и операција Dequeue као аргумент нема елемент који се избацује. FIFO својство реда узрокује да се ред понаша као ред муштерија које чекају у реду на каси. Ред садржи **почетак**, означаваћемо га са *poc* (енгл. *head*) и **крај**, у ознаци *kraj* (енгл. *tail*). Када се елемент умеће у ред, он заузима своје место на крају реда, као што и нова муштерија стаје на крај реда. Елемент који се избацује из реда је увек онај са почетка реда, као што је случај и са муштеријом на почетку реда која је чекала најдуже.

На слици 2.2 приказана је могућа имплементација реда од највише $n-1$ елемената коришћењем низа $Q[1..n]$. Ред има поље *poc* које показује на почетак реда. Поље *kraj* индексира наредну позицију на коју ће нови елемент бити уметнут у ред. Елементи реда се налазе на позицијама $poc, poc+1, \ldots, kraj-1$, при чему позиције посматрамо циклично, тј. имамо "премотавање преко краја низа" и позиција 1 се налази одмах након позиције n. Када је $poc = kraj$, ред је празан. На почетку важи да је $poc = kraj = 1$. Ако покушамо да избацимо елемент из празног реда, долази до грешке поткорачења реда. Када је $poc = kraj + 1$ (када је $kraj < n$) односно ако је $kraj = n$ и $poc = 1$, ред је пун и ако покушамо да у том моменту уметнемо нови елемент, долази до грешке прекорачење реда.

У псеудокоду процедура Enqueue и Dequeue које следе, изостављена је провера да ли је дошло до грешке због прекорачења или поткорачења. У псеудокоду се такође подразумева да је дужина низа Q једнака n.

```
Enqueue(Q, poc, kraj, x)
1 Q[kraj] ← x
2 if kraj = n then
3    kraj ← 1
4 else kraj ← kraj + 1
```



Слика 2.2: Имплементација реда преко низа $Q[1..12]$. Елементи реда обојени су светло сивом бојом. (а) Ред садржи 5 елемента, на позицијама $Q[7..11]$. (b) Конфигурација реда након позива Enqueue(Q,17), Enqueue(Q,3) и Enqueue(Q,5). (c) Конфигурација реда након позива Dequeue(Q) враћа вредност кључа 15 који се пре тога налазио на почетку реда. Нови почетак реда је елемент са кључем 6.

```
Dequeue(Q, poc, kraj)
1 x ← Q[poc]
2 if poc = n then
3     poc ← 1
4 else
5     poc ← poc + 1
6 return x
```

2.3 Повезане листе

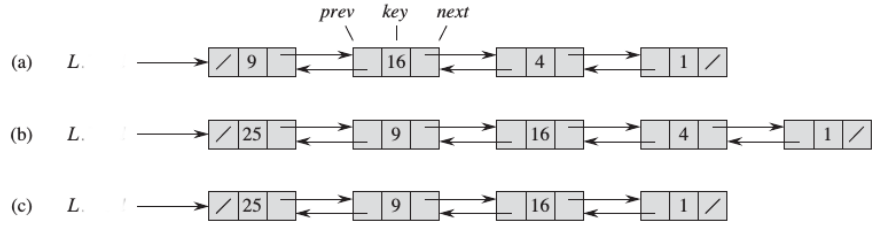
Повезана листа је структура података у којој су објекти уређени у линеарном поретку. За разлику од низа у ком је линеарни поредак одређен низом индекса, поредак у повезаним листама је одређен показивачем у сваком од објеката. Повезане листе омогућују једноставну и флексибилну репрезентацију динамичког скупа и подржавају (не нужно ефикасно) све поменуте операције над динамичким скуповима.

Листа може да има неколико различитих форми. Може бити једноструко или двоструко повезана, може бити уређена (сортирана) или не, и може бити циркуларна или не. Ако је листа **једноструко повезана** онда сваки елемент листе има показивач на наредни елемент листе, док код **двоструко повезане** листе сваки елемент листе има два показивача – на наредни елемент листе и на претходни елемент листе. Ако је листа **уређена**, линеарни поредак у листи одговара линеарном поретку кључева који се чувају

као елементи листе – у том случају је елемент са минималном вредношћу кључа глава листе, а елемент са максималним кључем реп. Ако је листа **неуређена**, елементи се могу јавити у произвољном поретку. У **цирку-ларној** (**кружној**) листи, показивач на претходни елемент главе листе показује на реп листе, а показивач на наредни елемент репа листе на главу листе. Циркуларну листу можемо посматрати као прстен елемената. У наставку текста подразумеваћемо да је листа са којом радимо неуређена и двоструко повезана.

2.3.1 Двоструко повезане листе

Као што је приказано на слици 2.3, сваки елемент **двоструко повезане листе** L је објекат са пољем *kljuc* и два додатна показивачка поља: *nar* (енгл. *next*) који показује на наредни елемент листе и *pret* (енгл. *prev*) који показује на претходни елемент листе. Објекти, такође, могу да садрже и неке сателитске податке. За дати објекат x у листи, $x.nar$ показује на следбеника елемента x у листи, а $x.pret$ на његовог претходника. Ако је $x.pret =$ **NULL**, елемент x нема претходника и стога је први елемент листе или **глава** листе. Ако је $x.nar =$ **NULL**, елемент x нема следбеника и стога је он последњи елемент листе или њен **реп**. Променљива L показује на почетак (први елемент) листе. Уколико је $L =$ **NULL** листа је празна.



Слика 2.3: (a) Двоструко повезана листа која представља динамички скуп $\{1, 4, 9, 16\}$. Сваки елемент листе је објекат који садржи поља за кључ и показиваче (приказане стрелицама) на наредни и претходни елемент. Поље *nar* репа листе и поље *pret* главе листе су **NULL**, што је приказано косом цртом. L показује на почетак листе. (b) Након извршења операције List_insert(L,x), где је $x.kljuc = 25$, повезана листа има нови елемент са вредношћу кључа 25 као нову главу листе. Нови објекат показује на стару главу са вредношћу кључа 9. (c) Резултат операције List_delete(L,x), где x показује на објекат са вредношћу кључа 4.

Проналажење елемента у повезаној листи Процедура List_search(L,k) тражи први елемент листе L са вредношћу кључа k простом линеарном претрагом и враћа показивач на тај елемент. Ако се у листи не налази објекат чија је вредност кључа k, процедура враћа вредност **NULL**. За повезану листу дату на слици 2.3(a), позив List_search(L,4) враћа показивач на трећи елемент листе, а позив List_search(L,7) враћа вредност **NULL**.

```
List_search(L, k)
1 x ← L
2 while x ≠ NULL and x.kljuc ≠ k do
3     x ← x.nar
4 return x
```

За листу од n елемената важи да је време извршавања ове операције у најгорем случају $O(n)$, с обзиром на то да може да се деси да треба претражити целу листу.

Уметање у повезану листу За дати елемент x чија је вредност кључа већ постављена, операција List_insert умеће x на почетак листе, као што је приказано на слици 2.3(b).

```
List_insert(L, x)
1 x.nar ← L
2 if L ≠ NULL then
3     L.pret ← x
4 L ← x
5 x.pret ← NULL
```

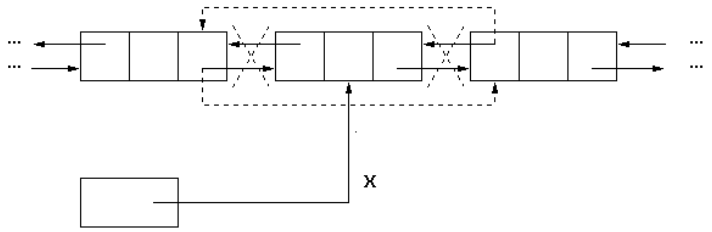
Време извршавања ове операције за листу од n елемената је у најгорем случају $O(1)$.

Слика 2.4: Уметање елемента у двоструко повезану листу

Брисање из повезане листе Процедура List_delete избацује елемент x из повезане листе L. Да би се ова операција извршила неопходно је да је дат показивач на елемент x; елемент x се из листе избацује ажурирањем вредности показивача. Ако желимо да из листе избацимо елемент са датом вредношћу кључа, неопходно је да прво позовемо процедуру List_search да бисмо добили показивач на жељени елемент.

```
List_delete(L, x)
1 if x.pret ≠ NULL then {није први елемент листе}
2     x.pret.nar ← x.nar
```

3 **else** {први елемент листе}
4 $L \leftarrow x.nar$
5 **if** $x.nar \neq$ **NULL** {није последњи елемент листе}
6 $x.nar.pret \leftarrow x.pret$



Слика 2.5: Брисање елемента из двоструко повезане листе

На слици 2.3(c) приказано је како се елемент избацује из повезане листе.

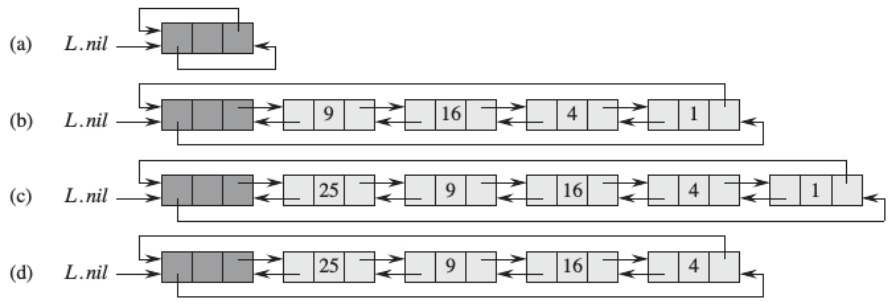
Процедура List_delete се извршава у времену $O(1)$. Уколико желимо да обришемо елемент са датом вредношћу кључа, време извршавања је у најгорем случају $O(n)$ јер прво мора да се позове процедура List_search да би се пронашао тражени елемент листе.

Кôд процедуре List_delete био би много једноставнији уколико бисмо игнорисали граничне услове за почетак и крај листе:

$List\_delete'(L, x)$
1 $x.pret.nar \leftarrow x.nar$
2 $x.nar.pret \leftarrow x.pret$

2.3.2 Двоструко повезане листе са граничником

Граничник (**сентинел**) је "вештачки" објекат који нам омогућава да поједноставимо граничне услове. На пример, претпоставимо да уз листу L прослеђујемо и објекат $L.nil$ који замењује **NULL** , али има све атрибуте осталих објеката у листи. Увек када имамо референцу на **NULL** у коду са листама, замењујемо је референцом на граничник $L.nil$. Као што је приказано на слици 2.6 ова измена трансформише стандардну двоструко повезану листу у **циркуларну двоструко повезану листу са граничником**, при чему се граничник $L.nil$ налази између главе и репа. Показивач $L.nil.nar$ указује на главу листе, а $L.nil.pret$ на реп листе. Слично, и показивач nar репа листе и показивач $pret$ главе листе указују на $L.nil$. С обзиром на то да $L.nil.nar$ указује на главу листе, можемо елиминисати



Слика 2.6: Циркуларна, двоструко повезана листа са граничником. (a) Празна листа (b) Повезана листа у којој глава има вредност кључа 9, а реп 1. (c) Листа након додавања објекта са кључем 25, нови објекат постаје глава листе (d) Листа након брисања објекта са кључем 1. Нови реп је објекат са кључем 4.

показивач који указује на главу листе, заменом референци на њега референцом на $L.nil.nar$.

На слици 2.6(a) приказано је да празна листа садржи само граничник и $L.nil.nar$ и $L.nil.pret$ указују на $L.nil$. Кôд процедуре List-search остаје као и пре, али са измењеним референцама на **NULL** и L.

$List\_search'(L, k)$
1 $x \leftarrow L.nil.nar$
2 **while** $x \neq L.nil$ **and** $x.kljuc \neq k$ **do**
3 $\quad x \leftarrow x.nar$
4 **return** x

Наредна процедура се користи за уметање елемента на почетак двоструко повезане листу са граничником:

$List\_insert'(L, x)$
1 $x.nar \leftarrow L.nil.nar$
2 $L.nil.nar.pret \leftarrow x$
3 $L.nil.nar \leftarrow x$
4 $x.pret \leftarrow L.nil$

На слици 2.6 примерима је илустровано дејство процедура List-insert' и List-Delete'.

2.3.3 Циркуларне повезане листе

Постоје случајеви када је проблем који решавамо циркуларан и када би било природније искористити циркуларне структуре података за њихово представљање. Генерално, предност једноструко повезаних циркуларних листа је у томе што се цела листа може обићи из сваког елемента листе,

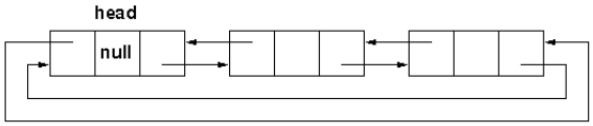
док код нециркуларних листи из елемента у средини листе није могуће приступити елементима који му претходе. Приликом имплементације операција са двоструко повезаним циркуларним листама постоји мањи број специјалних случајева (јер сви елементи имају претходника и следбеника). Ипак, зависно од имплементације, уметање на почетак листе би захтевало проналажење последњег елемента листе, а то може бити скупо. Такође, налажење краја листе је теже и самим тим контрола петље (јер не постоји **NULL** маркер за ознаку почетка и краја листе).

Код нециркуларне листе обично имамо показивач на први елемент листе. Међутим, код циркуларне листе (посебно једноструко повезане) овакав начин имплементације није најбоље решење. Разлог је следећи: уколико имамо показивач на почетак листе и хоћемо да уметнемо/обришемо елемент са почетка листе, морали бисмо да прођемо кроз целу листу да бисмо нашли последњи елемент листе и да бисмо на коректан начин повезали нову листу.

Једна од ствари коју бисмо могли да урадимо јесте да додамо и показивач на последњи елемент листе. Међутим, на тај начин је приликом извршавања сваке операције над листом потребно водити рачуна о постављању коректне вредности још једног атрибута листе.

Друга идеја јесте да заборавимо у потпуности на показивач на први елемент листе и да само чувамо показивач на последњи елемент листе, јер ако знамо показивач на последњи елемент врло лако ћемо одредити и показивач на први ($poc = kraj.nar$).

Трећа могућност јесте да постоји **водећи елемент** (слично као граничник код нециркуларне листе) који се поставља као први елемент циркуларне листе. Овај елемент се од осталих може разликовати било тако што ће имати неку специјалну вредност (нпр. ако елементи листе имају вредности из скупа позитивних бројева, онда он може имати вредност -1) или тако што се сваком елементу дода један флег који има вредност 1 ако је у питању водећи елемент, а 0 иначе.

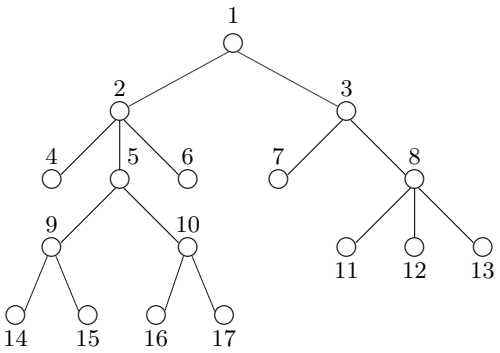


Слика 2.7: Пример двоструко повезане листе са водећим елементом која садржи два елемента.

2.4 Стабла

Низови и повезане листе одражавају само редослед елемената. Често је потребно представити неку сложенију структуру односа међу елементима. **Стабло** је хијерархијска структура, али може да послужи као ефикасна структура за неке операције над линеарним структурама. У оквиру овог поглавља бавићемо се само **хијерархијским** или **коренским стаблом**. Коренско стабло чини скуп елемената које ћемо даље звати **чворовима** и скуп **грана** које повезују чворове на специфичан начин. Један издвојени

чвор је **корен** стабла и он представља врх хијерархије (видети пример на слици 2.8). Корен је повезан са другим чворовима и за њих ћемо рећи да чине ниво 1 хијерархије, чворови везани за њих (сем корена) чине ниво 2 хијерархије итд. Све везе су дакле између чворова и њихових (јединствених) **родитеља** (зваћемо га још и **претходник** или **отац** чвора); једино корен нема оца. Основна карактеристика стабла је да нема циклуса, због чега између свака два чвора у стаблу постоји јединствени пут. Чвор стабла је повезан са оцем и неколико својих **следбеника** (**синова**). Максимални број синова чвора у графу назива се **степен** стабла. Најчешће се за синове сваког чвора дефинише редослед, па се синови могу идентификовати својим редним бројем. Стабло степена два назива се **бинарно стабло**. Сваки чвор бинарног стабла може да има највише два сина: левог и десног. Чвор који нема деце називамо **лист**, а чвор који није лист **унутрашњи чвор**. **Висина стабла** је највећи ниво његове хијерархије, тј. то је максимално растојање између чвора и корена (тј. број грана на јединственом путу од корена до чвора). Сваки чвор има **кључ** који узима вредност из неког потпуно уређеног скупа (нпр. из скупа целих или реалних бројева). Претпоставља се да су кључеви јединствени. Сваки чвор може да има поља са подацима чији садржај зависи од примене стабла.



Слика 2.8: Пример коренског стабла.

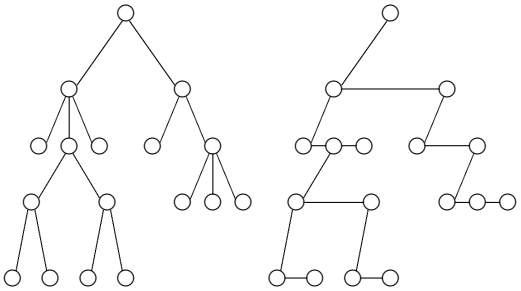
У даљем тексту видећемо како можемо представити стабло, а затим ће бити речи о бинарном стаблу претраге и хипу, као примерима структура података које су засноване на бинарном стаблу.

2.4.1 Представљање стабла

У зависности од тога да ли се користе показивачи или не, представа стабла може бити **експлицитна** и **имплицитна**. У експлицитној представи гране се представљају помоћу показивача, тј. чвор са k синова се представља структуром која садржи низ од k показивача (понекад и додатно показивач ка оцу). Згодно је да сви чворови буду истог типа тако да сваки чвор садржи m показивача, при чему је m максимални број деце неког чвора, тј. степен стабла.

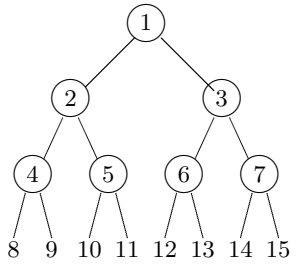
Стабло степена већег од 2 могуће је представити коришћењем само 2 показивача по чвору: први показује на првог сина, а други на наредног брата. Другим речима, синови чвора представљени су повезаном листом

на коју показује први показивач у чвору. За повезивање елемената листе користе се други показивачи у чворовима листе. Пример једне овакве представе дат је на слици 2.9.



Слика 2.9: Приказивање небинарног стабла са по највише два показивача по чвору.

Код имплицитног представљања стабла, не користе се показивачи. Наиме, сви кључеви чворова се смештају у низ и везе међу њима проистичу из њиховог места у низу. Најчешће се ради на следећи начин: корен се смешта на позицију $A[1]$, леви и десни син на $A[2]$ и $A[3]$, итд. Редослед којим се чворови смештају у низ илустрован је на слици 2.10.



Слика 2.10: Имплицитно представљање бинарног стабла. Број уз чвор означава његову позицију у низу.

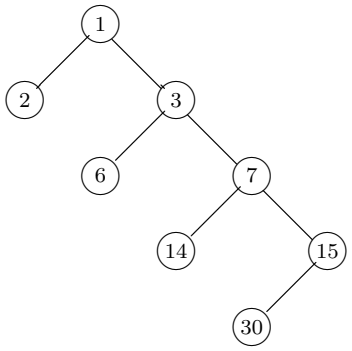
У низу се чворови приказују у оном редоследу којим се стабло обилази слева удесно, ниво по ниво. Притом се мора резервисати простор и за чворове стабла који не постоје. Може се показати да ако је чвор смештен на позицију $A[i]$, онда се његови леви и десни син налазе на позицијама $A[2i]$ и $A[2i+1]$. Покажимо математичком индукцијом да ово важи:

база индукције: $i = 1$, на позицији $A[1]$ налази се корен и његов леви и десни син се налазе на позицијама $A[2]$ и $A[3]$ те тврђење важи за $i = 1$

индуктивни корак: претпоставимо да тврђење важи за $i = k$ и покажимо да важи за $i = k+1$. Из индуктивне хипотезе следи да се синови чвора $A[k]$ налазе на позицијама $A[2k]$ и $A[2k+1]$. Јасно је да се синови чвора $A[k+1]$ налазе одмах након синова чвора $A[k]$ у имплицитној представи стабла, односно на позицијама $A[2k+2]$ и $A[2k+3]$, тј. на

позицијама $A[2(k+1)]$ и $A[2(k+1)+1]$, те смо доказали да тврђење важи за $i = k+1$.

Овај начин представе је згодан због своје једноставности и компактности. Међутим ако је стабло неуравнотежено (тј. неки листови су много даље од корена од других), мора се резервисати простор за велики број непостојећих чворова. На пример, за стабло на слици 2.11, које са састоји од 8 чворова потребно је резервисати низ дужине 30.



Слика 2.11: Пример неуравнотеженог стабла, за које је имплицитно представљање неефикасно.

2.5 Хип

Хип је бинарно стабло које задовољава **услов хипа**: кључ сваког чвора је већи или једнак од кључева његових синова. Из транзитивности релације $\geq$ следи да је кључ сваког чвора већи или једнак од кључева свих његових потомака. Често се у литератури ова реализација хипа назива и макс-хип, док се бинарно стабло које задовољава услов да је кључ сваког чвора мањи или једнак од кључева његових синова назива мин-хип.

Хип је структура података корисна за реализацију **реда са приоритетом**. То је апстрактна структура података за коју су дефинисане две операције:

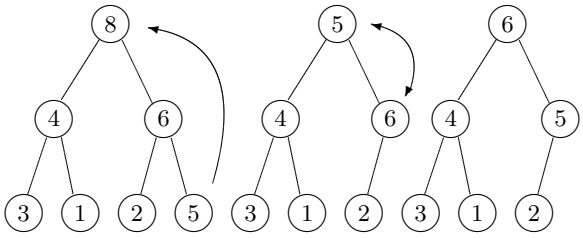
Уметни(x) - уметни кључ x у структуру

Избаци() - избаци највећи кључ из структуре

Хип се може реализовати имплицитно или експлицитно представљеним 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 низ $A[1..k]$, при чему је k горња граница за број елемената хипа. Ако је n текући број елемената хипа, онда се за смештање елемената хипа користе локације у низу A са индексима од 1 до n.

2.5.1 Уклањање са хипа елемента са највећим кључем

Кључ највећег елемента је $A[1]$, па га је лако пронаћи. После тога треба трансформисати преостали садржај низа тако да представља исправан хип. То се може постићи тако да се најпре $A[n]$ прекопира у корен ($A[1] \leftarrow A[n]$) и n смањи за један. Означимо са $x \leftarrow A[1]$ нови кључ корена, а са $y \leftarrow \max\{A[2], A[3]\}$ 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 низу 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 x 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. Ова операција описана је кодом *SkiniMaxSaHipa*. На слици 2.12 приказан је пример уклањања највећег елемента из хипа.



Слика 2.12: Уклањање највећег елемента са хипа.

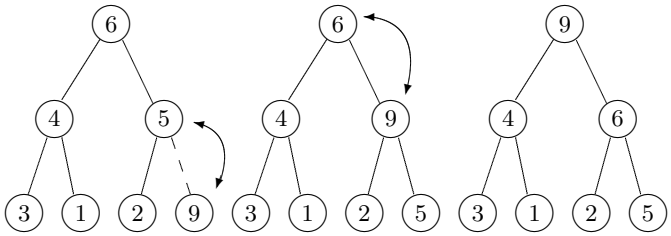
SkiniMaxSaHipa(A, n)
улаз: A - низ величине n за смештање хипа
излаз: Vrh_hipa - највећи елемент хипа,
A - нови хип, n - нова величина хипа

```
1  if n = 0 then
2     error "potkoracenje"
3  else
4     Vrh_hipa ← A[1]
5     A[1] ← A[n]
6     n ← n − 1
7     Otac ← 1
8     if n > 1 then
9        Sin ← 2
10    while Sin ≤ n do
11       if Sin + 1 ≤ n and A[Sin] < A[Sin + 1] then
12          Sin ← Sin + 1
13       if A[Sin] > A[Otac]
```

```
14        zameni(A[Otac], A[Sin])
15        Otac ← Sin
16        Sin ← 2 · Sin
17      else
18        Sin ← n + 1 {да би се искочило из петље}
19  return Vrh_hipa
```

2.5.2 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 $i = \lfloor n/2 \rfloor$, 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 се 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 2.13.



Слика 2.13: Уметање елемента у хип.

Број упоређивања приликом извршења обе операције са хипом је 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 пронаћи задати кључ.

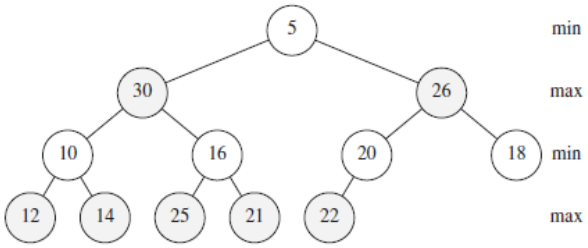
```
UpisUHip(A, n, x)
улаз: A - низ од n кључева хипа, x - број
излаз: A - нови хип, n - нова величина хипа
1  n ← n + 1 { претпоставка је да ново n није веће од величине A }
2  A[n] ← x
3  Sin ← n
4  Otac ← n div 2
5  while Otac ≥ 1 do
6    if A[Otac] < A[Sin] then
7      zameni(A[Otac], A[Sin])
8      Sin ← Otac
9      Otac ← Otac div 2
10   else
```

11 $Otac \leftarrow 0$ {за искакање из петље}

2.5.3 Мин-макс хип

Некада је потребно обезбедити приступ и најмањем и највећем елементу у логаритамском времену. Апстрактну структуру података која подржава приступ овим операцијама називамо **ред са приоритетом са два краја**. За реализацију ове апстрактне структуре података може се користити структура података коју називамо **мин-макс хип** (eng. min-max heap). Мин-макс хип је бинарно стабло које комбинује предности и мин-хипа и макс-хипа. Уметање елемента у мин-макс хип и брисање минималног и максималног елемента могу се извести у времену $O(\log n)$. Као и код класичног хипа, и ова врста хипа се најчешће реализује коришћењем низа, односно имплицитно.

У мин-макс хипу сваки чвор задовољава наредни услов: кључ сваког чвора на парном нивоу стабла је мањи од кључева свих његових потомака, а кључ сваког чвора на непарном нивоу је већи од кључева свих његових потомака у стаблу (видети пример на слици 2.14). Подразумевамо да је корен на нивоу 0. Последица је да се у корену налази најмањи елемент стабла, док је један од два елемента на првом нивоу стабла највећи елемент у овој структури. Често уместо парног нивоа кажемо минимални ниво, а уместо непарног кажемо максимални ниво (у складу са условом који важи на овим нивоима).



Слика 2.14: Пример мин-макс хипа

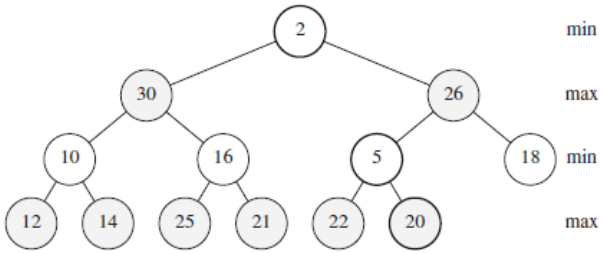
Како се изводе основне операције код мин-макс хипа? Уметање се, као и код основног хипа, врши тако што се n повећа за један и нови елемент се ставља на место $A[n]$. Овим се може нарушити неки од жељених односа у хипу те је хип потребно преуредити. Додавањем новог елеметна могуће је нарушити само односе чворова који се налазе на путу од корена стабла до оца новододатог елемента. Разликујемо два случаја у зависности од тога да ли се елемент додаје на парни или непарни ниво:

- уколико се нови елемент $A[n]$ налази на минималном нивоу, онда се његова вредност упоређује са вредношћу његовог оца $A[n/2]$
  - ако је $A[n/2] < A[n]$ онда се та два елемента замењују, а затим се почев од чвора $A[n/2]$ пролази уназад ка корену кроз максималне нивое и врши преуређење хипа уколико је нарушен услов хипа;

  - ако је $A[n/2] > A[n]$ онда се почев од чвора $A[n]$ пролази уназад ка корену кроз минималне нивое и хип се преуређује, уколико је потребно

- уколико се нови елемент $A[n]$ налази на максималном нивоу, онда се његова вредност упоређује са вредношћу његовог ова $A[n/2]$

  - ако је $A[n/2] > A[n]$ онда се та два елемента замењују, а затим се почев од чвора $A[n/2]$ пролази уназад ка корену кроз минималне нивое и хип се преуређује уколико је потребно;
  - ако је $A[n/2] < A[n]$ онда се почев од чвора $A[n]$ пролази уназад ка корену кроз максималне нивое и хип се преуређује, уколико је потребно.



Слика 2.15: Мин-макс хип са слике 2.14 након додавања елемента 2

Преуређивање само максималних, односно само минималних нивоа ради се исто као и у стандардном хипу, само што за сваки чвор гледамо његовог деду (оца његовог оца), уместо оца.

Брисање минималног, односно максималног елемента изводи се слично као и код стандардног хипа. Жељени елемент се избацује из хипа, а на његово место се доводи последњи елемент у хипу, а затим је потребно извршити нека преуређења хипа која зависе од тога да ли се избацује минимални или максимални елемент хипа.

- Уколико бришемо минимални елемент онда се проналази најмањи међу његовим синовима и унуцима (неопходно је разматрати и синове разматраног чвора јер у подстаблу не морају постојати сви унуци и може се десити да неки син има вредност која је мања од свих унука тог чвора): ако је минимални међу њима унук и ако је он мањи од посматраног елемента, размењује им се место и проверава да ли се тим нарушио однос оца тог унука и унука (ако јесте размењују им се места) и наставља се надоле по минималним нивоима; ако је најмањи чвор син и он је већи од текућег чвора, онда им се само размењују места.

- Брисање максималног елемента се изводи на аналоган начин.

2.6 Бинарно стабло претраге (уређено бинарно стабло)

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, енг. Binary Search Tree, тј. бинарном стаблу које омогућује претраживање), односно **уређеном бинарном стаблу**, кључ сваког чвора већи је од кључева свих чворова левог подстабла, а мањи од кључева свих чворова десног подстабла (видети слику 2.17). 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
- **Ukloni**(x) – ако у структури података постоји елеменат са кључем x, уклони га.

Као што смо већ поменули, апстрактна 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 структура са бар три поља: *kljuc*, *levi* и *desni*. Друго, односно треће поље су показивачи ка другим чворовима, или **NULL** (ако чвор нема левог, односно десног сина). БСП је компликованија структура података од хипа: код хипа се приступа само листовима, док се у БСП сваки чвор може избацити.

2.6.1 Налажење кључа у БСП

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 $x = r$ онда је тражење завршено. Ако је пак $x < r$ (односно $x > r$) онда се тражење рекурзивно наставља у левом (односно десном) подстаблу. Овим се у ствари полови интервал индекса кључева.

Nadji_u_BSP($koren, x$)
улаз: *koren* - показивач на корен БСП, x - број
излаз: *cvor* - показивач на чвор који садржи број x
или **NULL** уколико таквог чвора нема

```
1 if koren = NULL or koren.kljuc = x then
2    cvor ← koren
3 else
4    if x < koren.kljuc then Nadji_u_BSP(koren.levi, x)
5    else Nadji_u_BSP(koren.desni, x)
```

2.6.2 Уметање у БСП

Уметање у БСП је такође једноставна операција. Кључ x који треба уметнути најпре се тражи у БСП; ако се пронађе, уметање је завршено; претпоставка 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.

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 је *sin* = **NULL**

```
1 if koren = NULL then
2    креирај нови чвор на кога показује sin
3    koren ← sin
4    koren.kljuc ← x
5    koren.levi ← NULL
6    koren.desni ← NULL
7 else
8    cvor ← koren { текући чвор у стаблу}
9    sin ← koren { поставља се на вредност различиту од NULL }
10   while cvor ≠ NULL and sin ≠ NULL do
11      if cvor.kljuc = x then sin ← NULL
12      else { силазак низ стабло за један ниво}
13         otac ← cvor
14         if x < cvor.kljuc then cvor ← cvor.levi
15         else cvor ← cvor.desni
16   if sin ≠ NULL then { нови чвор је син чвора otac }
17      креирај нови чвор на кога показује sin
18      sin.kljuc ← x
19      sin.levi ← NULL
20      sin.desni ← NULL
21      if x < otac.kljuc then otac.levi ← sin
22      else otac.desni ← sin
```

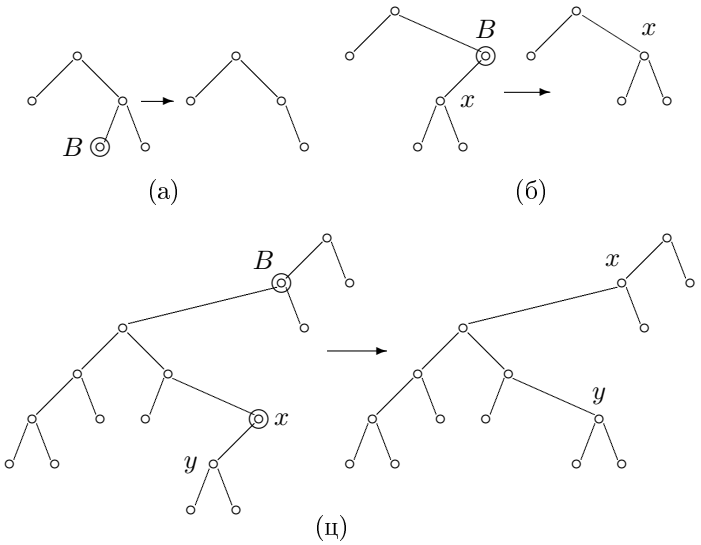
2.6.3 Брисање из БСП

Брисање чвора из БСП коме је кључ једнак задатом броју такође почиње тражењем кључа у стаблу и нешто је компликованија операција од претходних. Најједноставније је обрисати лист: показивач на њега у његовом

оцу поставља се на **NULL** . 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. 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 (видети илустрацију на слици 2.16); полазећи од корена тог подстабла 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**.

Ukloni_iz_BSP($koren, x$)
улаз: *koren* - показивач на корен БСП, x - број
излаз: показивач на корен промењеног стабла

```
1   {прва фаза: тражење чвора са кључем x}
2   cvor ← koren
3   while cvor ≠ NULL and cvor.kljuc ≠ x do
4     otac ← cvor
5     if x < cvor.kljuc then cvor ← cvor.levi
6     else cvor ← cvor.desni
7   if cvor = NULL then
8     print("x није у БСП")
9     halt
10  {брисање нађеног чвора cvor или његовог претходника}
11  if cvor ≠ koren then
12    if cvor.levi = NULL then
13      if x ≤ otac.kljuc then
14        otac.levi ← cvor.desni
15      else otac.desni ← cvor.desni
16    else if cvor.desni = NULL then
17      if x ≤ otac.kljuc then
18        otac.levi ← cvor.levi
19      else otac.desni ← cvor.levi
20    else { случај са два сина }
21      cvor1 ← cvor.levi
22      otac1 ← cvor
23      while cvor1.desni ≠ NULL do
24        otac1 ← cvor1
25        cvor1 ← cvor.desni
26      {а сада иде брисање, cvor1 је претходник чвора cvor}
27      cvor.kljuc ← cvor1.kljuc
28      if otac1 = cvor then
29        otac1.levi ← cvor1.levi {cvor1 је леви син чвора otac1}
30      else
31        otac1.desni ← cvor1.levi {cvor1 је десни син чвора otac1}
```



Слика 2.16: Брисање елемента бинарног стабла претраге.

Операција следбеник дефинисана је и ако дати чвор x нема десног сина. У том случају x има највећи кључ у подстаблу са тачно одређеним кореном y; тада је отац чвора y следбеник чвора x. Наравно, ако је чвор y корен стабла, следбеник не постоји (видети примере на слици 2.17). Према томе, операција следбеник може се описати следећим кôдом (аналогним кôдом може се описати операција претходник).

Sledbenik(x)
улаз: *koren* - показивач на корен стабла,
x - показивач на чвор стабла чијег следбеника тражимо
излаз: y - показивач на следбеника чвора x
или **NULL** уколико таквог чвора нема

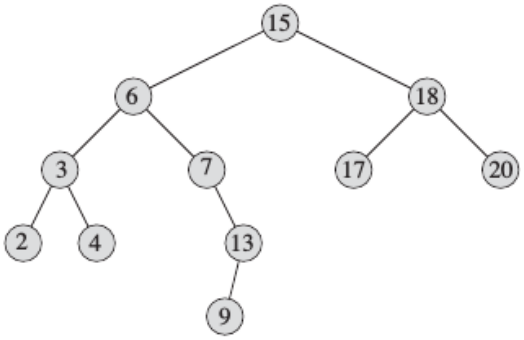
```
1   if x.desni ≠ NULL then return Minimum(x.desni)
2   otac ← koren
3   sled ← NULL
4   while otac ≠ NULL
5     if x.kljuc < otac.kljuc
6       sled ← otac
7       otac ← otac.levi
8     elseif x.kljuc > otac.kljuc
9       otac ← otac.desni
10    else
11      break
12  return sled
```

Minimum(x)
улаз: x - корен бинарног стабла претраге
излаз: x - показивач на минимални елемент датог бинарног стабла претраге

```
1 while x.levi ≠ NULL
```

2 $x \leftarrow x.levi$
3 **return** x

Операција $Maksimum(x)$ за проналажење максималног елемента бинарног стабла претраге са датим кореном може се описати аналогним кôдом.



Слика 2.17: Следбеник чвора са кључем 15 је чвор са кључем 17 јер је он минимални кључ у десном подстаблу стабла са кореном 15. Чвор са кључем 13 нема десно подстабло па је његов кључ највећи од кључева у подстаблу са кореном у чвору y са кључем 6, а следбеник је отац чвора y, са кључем 15. Чвор са кључем 20 нема следбеника у стаблу.

Структура бинарног стабла претраге омогућава проналажење следбеника (претходника) чвора без икаквог поређења кључева уколико је уз чвор дат показивач *otac*. Наредна процедура враћа следбеника чвора x у БСП ако постоји, односно **NULL** ако је x чвор са највећом вредношћу у стаблу.

$Sledbenik'(x)$
улаз: x - показивач на чвор стабла чијег следбеника тражимо
излаз: y - показивач на чвор који садржи следбеника
или **NULL** уколико таквог чвора нема
1 **if** $x.desni \neq$ **NULL** **then return** $Minimum(x.desni)$
2 $y \leftarrow x.otac$
3 **while** $y \neq$ **NULL** **and** $x = y.desni$
4 $x \leftarrow y$
5 $y \leftarrow y.otac$
6 **return** y

2.6.4 Сложеност операција над БСП

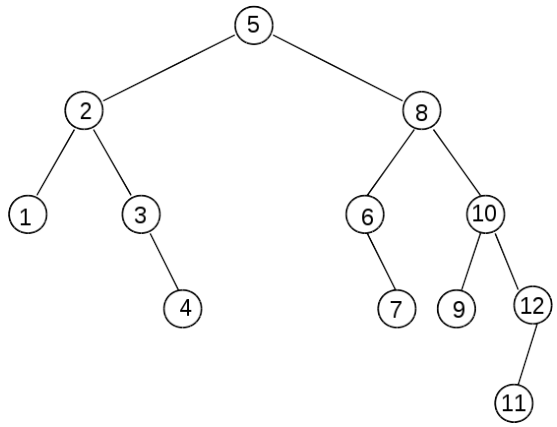
За све три разматране операције време извршења зависи од облика стабла и положаја чвора на коме се врши интервенција — у најгорем случају се тражење завршава у листу стабла. Приликом брисања кључа у чвору са оба сина, наставља се са спуштањем наниже, али је и у том случају

укупан пут који се прелази мањи од висине 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, 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 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 је коришћење АВЛ стабала, која су подврста бинарних стабала претраге.

2.7 АВЛ стабла

АВЛ стабла (која су име добила по ауторима, Адельсон–Вельский и Ландис, 1962.) су структура података која гарантује да сложеност ни једне од операција тражења, уметања и брисања у **најгорем случају** није већа од $O(\log n)$, где је n 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. Прецизније, АВЛ стабло се дефинише као бинарно стабло претраге код кога је за сваки чвор апсолутна вредност разлике висина левог и десног подстабла мања или једнака од један (видети пример на слици 2.18).



Слика 2.18: Пример АВЛ стабла.

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

Теорема 1. *За АВЛ стабло са n унутрашњих чворова висина h задовољава услов $h < 1.4405 \log_2(n+2) - 0.327$.*

Доказ. Нека је T_h АВЛ стабло висине $h \geq 0$ са најмањим могућим бројем чворова. Стабло T_0 садржи само корен, а стабло T_1 корен и једног сина, рецимо десног. За $h \geq 2$ стабло T_h може се формирати на следећи начин: 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 2.19. Означимо са n_h минимални број унутрашњих чворова АВЛ стабла висине h, тј. број унутрашњих чворова стабла T_h; $n_0 = 0$, $n_1 = 1$, $n_2 = 2$, $n_3 = 4, \ldots$. 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 као и Фибоначијеви бројеви F_h ($F_{h+2} = F_{h+1} + F_h$ за $h \geq 1$; $F_1 = F_2 = 1$). Узимајући у обзир и прве чланове низа, индукцијом се непосредно показује да важи $n_h + 1 = F_{h+2}$. Према томе, за број n унутрашњих чворова АВЛ стабла висине h важи неједнакост

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

Полазећи од израза за општи члан Фибоначијевог низа (видети пример у делу о решавању линеарних диференцних једначина)

$$F_h = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^h$$

и везе

$$\frac{1-\sqrt{5}}{2} = -\frac{1}{\frac{1+\sqrt{5}}{2}}$$

добијамо

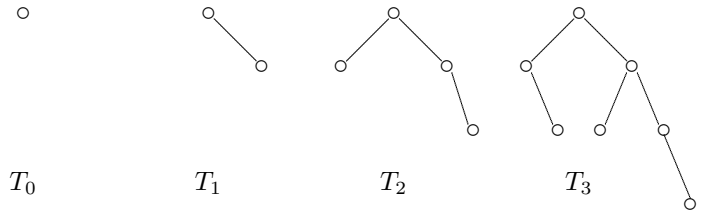
$$n \geq \frac{1}{\sqrt{5}}\left(\alpha^{h+2} - \left(-\frac{1}{\alpha}\right)^{h+2}\right) - 1 > \frac{1}{\sqrt{5}}\alpha^{h+2} - 2,$$

где је са $\alpha = (\sqrt{5}+1)/2$ означен позитиван корен карактеристичне једначине $x^2 - x - 1 = 0$ линеарне диференцне једначине $F_n = F_{n-1} + F_{n-2}$. 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 $\log_2 5/(2\log_2 \alpha) \simeq 0.3277$.

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



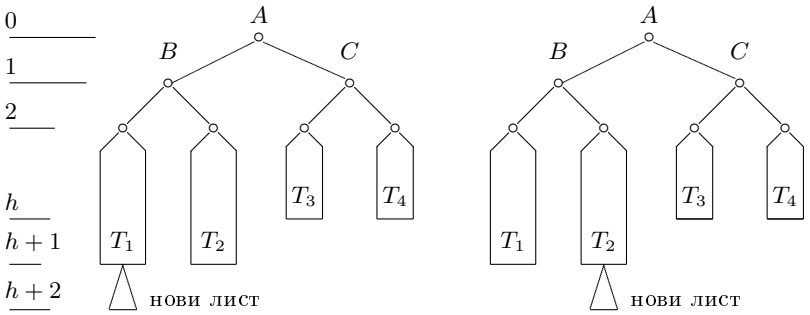
Слика 2.19: Фибоначијева стабла T_0, T_1, T_2 и T_3.

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 чвор са кључем једнаким задатом броју. Чворовима на путу који се том приликом прелазе одговарају разлике висина левог и десног подстабла (**фактори равнотеже**) из скупа $\{0, \pm 1\}$. Посебно је интересантан последњи чвор на том путу који има фактор равнотеже различит од нуле, тзв. **критични чвор**. Испоставља се да је приликом уметања броја у АВЛ стабло *довољно уравнотежити подстабло са кореном у критичном чвору* - у то ћемо се уверити када видимо детаље поступка уравнотежавања.

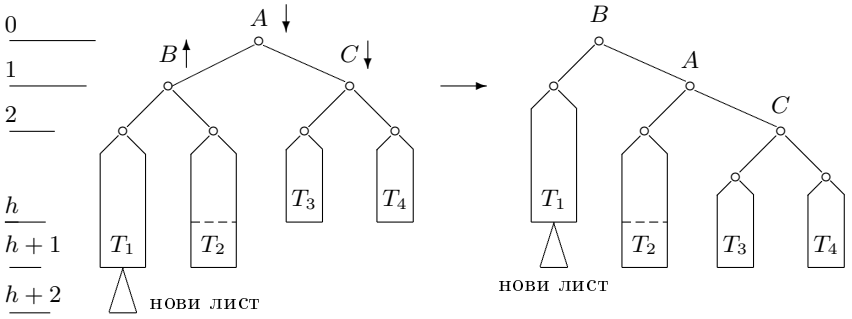
Претпоставимо да је фактор равнотеже у критичном чвору једнак 1. Нови чвор може да заврши у:

- десном подстаблу – у том случају подстабло коме је корен критични чвор остаје АВЛ
- левом подстаблу – у том случају стабло престаје да буде АВЛ и потребно је интервенисати. Према томе да ли је нови чвор додат левом или десном подстаблу левог подстабла, разликујемо два случаја, видети слику 2.20.
  - У првом случају се на стабло примењује **ротација**: корен левог подстабла B (видети слику 2.21) подиже се и постаје корен подстабла (коме је корен био критичан чвор), а остатак стабла преуређује се тако да стабло и даље остане БСП. 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. Пошто је A критичан чвор, фактор равнотеже чвора B је 0, па стабла T_1 и T_2 имају исту висину. Стабла T_3 и T_4 не морају имати исту висину, јер чвор C није на путу од критичног чвора до места уметања (слика 2.21).
  - Други случај је компликованији; тада се стабло може уравнотежити **двоструком ротацијом**, видети слику 2.22. Нови чвор подстабла уместо критичног чвора постаје десни син левог сина критичног чвора.

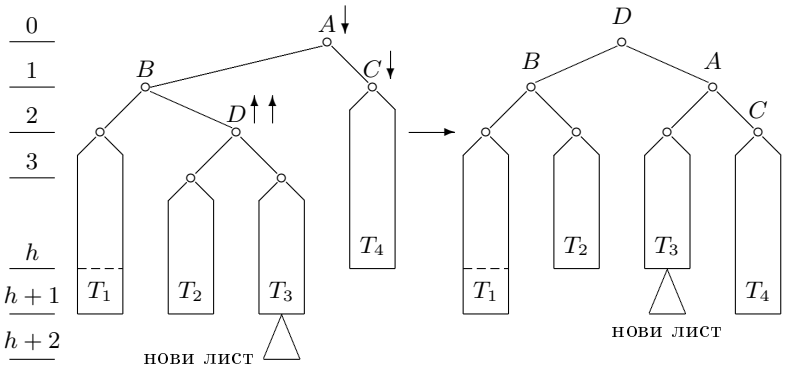
Запажамо да у оба случаја висина подстабла коме је корен критични чвор после уравнотежавања остаје непромењена. Уравнотежавање подстабла коме је корен критичан чвор због тога не утиче на остатак стабла.



Слика 2.20: Уметања која ремете АВЛ својство стабла.



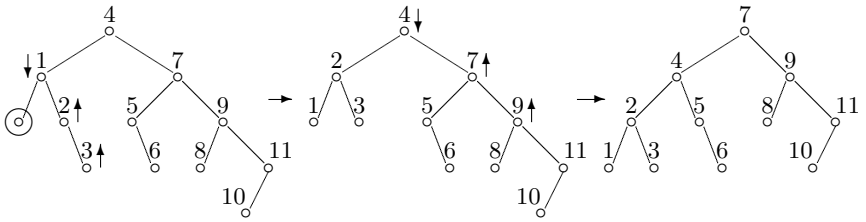
Слика 2.21: Уравнотежавање АВЛ–стабла после уметања — ротација.



Слика 2.22: Уравнотежавање АВЛ–стабла после уметања — двострука ротација.

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. 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 прође 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 "лоше" изабраног чвора, потребно је извршити $h-2$, односно $O(\log n)$ ротација (видети слику 2.23 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 време извршавања брисања такође ограничено одозго са $O(\log n)$. На овом месту прескочићемо детаље.



Слика 2.23: 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, на пример $2-3$ стабла, B-стабла и црвено-црна стабла.

2.8 Скуповне операције

Коначни скупови су уобичајена и често коришћена структура података. Стога је потребно осмислити једноставан и ефикасан начин за њихово представљање и за њихову манипулацију, односно за тражење, уметање и брисање елемента скупа, као и бинарне операције са скуповима (унија, пресек, разлика, симетрична разлика). У наставку текста претпоставља се да су скупови коначни и да су вредности елемената скупа из скупа ненегативних целих бројева.

Скупове је могуће представити на више различитих начина. Размотрићемо два начина имплементације скупа, помоћу сортираног низа и помоћу битског низа (енг. bit array).

Нека су дата два скупа: скуп S_1 који садржи m елемената и скуп S_2 који садржи n елемената и нека је сваки од њих представљен сортираним низом елемената скупа. У том случају операција испитивања присуства елемента у скупу од n елемената своди се на бинарну претрагу, па се ова операција може извести за време $O(\log n)$. За уметање новог елемента у скуп се такође може искористити бинарна претрага за одређивање његове позиције, али како је у најгорем случају за додавање најмањег елемента у скуп потребно померити за једно место удесно све елементе, то је операција уметања сложености $O(n)$. Елемент се брише из скупа тако што се прво бинарном претрагом испита његово присуство у скупу, а затим се сви елементи десно од њега померају за једно место улево. Како је за брисање најмањег елемента скупа потребно померити све елементе скупа за једно место улево, то је и ова операција сложености $O(n)$.

Све четири бинарне операције над скуповима S_1 и S_2 (унија, пресек, разлика, симетрична разлика) могу се извршити на сличан начин као што се врши обједињавање два сортирана низа у један. У тренутку када се разматрају елементи $S_1[i]$ и $S_2[j]$, приликом израчунавања:

- уније $S_1 \cup S_2$ ако је:
  - $S_1[i] = S_2[j]$, у излазни низ се копира само један број, а оба индекса i, j се повећавају за један,
  - $S_1[i] \neq S_2[j]$, у излазни низ се копира само мањи број и напредује се са индексом у том низу,
- пресека $S_1 \cap S_2$ ако је:
  - $S_1[i] = S_2[j]$, у излазни низ се копира само један број, а оба индекса i, j се повећавају за један,
  - $S_1[i] \neq S_2[j]$, напредује се са индексом у низу који садржи мањи број,
- разлике $S_1 \setminus S_2$ ако је:
  - $S_1[i] < S_2[j]$, у излазни низ се копира број $S_1[i]$ и индекс првог низа се повећава за један,
  - $S_1[i] = S_2[j]$, напредује се са индексом у оба низа,
  - $S_1[i] > S_2[j]$, напредује се са индексом у другом низу,
- симетричне разлике $S_1 \Delta S_2 = (S_1 \setminus S_2) \cup (S_2 \setminus S_1)$ ако је:
  - $S_1[i] \neq S_2[j]$, у излазни низ се копира мањи број и индекс тог низа се повећава за један,
  - $S_1[i] = S_2[j]$, напредује се са индексом у оба низа,

Све четири основне бинарне операције за рад са скуповима: налажење уније, пресека, разлике и симетричне разлике су временске сложености $O(m+n)$. На пример, ако је $S_1 = \{2,3,6\}$ и $S_2 = \{1,2,4,6,7\}$ добијају се очекивани резултати $S_1 \cup S_2 = \{1,2,3,4,6,7\}$, $S_1 \cap S_2 = \{2,6\}$, $S_1 \Delta S_2 = \{3\}$, $(S_1 \Delta S_2) = \{1,3,4,7\}$.

Уколико је максимални број елемената скупа довољно мали, скуп се може имплементирати битским низом. Правило је да елемент n припада скупу ако и само ако n-ти бит у низу има вредност 1. С обзиром на то да се у овој репрезентацији користи само један бит по елементу, ова репрезентација се може посматрати просторно ефикасном. С друге стране, дужина низа је пропорционална са могућим опсегом вредности елемената низа, што није случај код претходно описане имплементације.

Уметање елемента у скуп, односно брисање се спроводи једноставном изменом вредности одговарајућег бита. Провера припадности броја скупу своди се на очитавање одговарајућег бита (видети следећи кôд).

Sadrzi(A, x)
улаз: A - низ битова којим је представљен скуп,
x - елемент чије присуство испитујемо
излаз: *true* - ако елемент припада, *false* - ако елемент не припада скупу

```
1 if x ≥ duzina(A) then
2    return false
3 elseif A[x] = 1 then
4    return true
5 else
6    return false
```

Операције уметања елемента, брисања елемента и испитивања присуства елемента у скупу од n елемената су временске сложености $O(1)$.

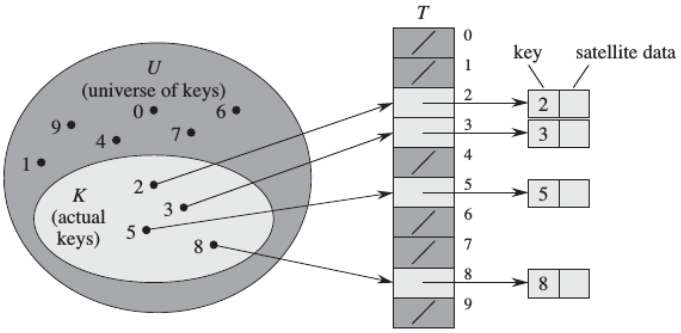
Претпоставимо да су скупови S_1 и S_2 задати низовима битова B_1 и B_2. Пресек и унија скупова се могу добити битском конјункцијом $B_1 \& B_2$, односно битском дисјункцијом $B_1 | B_2$. Операција симетричне разлике скупова се може добити операцијом ексклузивне дисјункције битова у запису скупова $B_1 \hat{} B_2$, а обична разлика скупова се може израчунати битском конјункцијом првог низа битова са негацијом другог низа битова $B_1 \& \sim B_2$. На пример, ако су дати скупови $S_1 = \{2,3,5\}$ и $S_2 = \{0,2,4\}$ и ако их представљамо низовима битова дужине 8, онда су одговарајући низови $B_1 = 00110100$ и $B_2 = 10101000$ и

- унији $S_1 \cup S_2 = \{0,2,3,4,5\}$ одговара низ $B_1 | B_2 = 10111100$,
- пресеку $S_1 \cap S_2 = \{2\}$ одговара низ $B_1 \& B_2 = 00100000$,
- разлици $S_1 \setminus S_2 = \{3,5\}$ одговара низ $B_1 \& \sim B_2 = 00010100$,
- симетричној разлици $S_1 \Delta S_2 = \{0,3,4,5\}$ одговара низ $B_1 \hat{} B_2 = 10011100$.

2.9 Хеш табеле

Хеш табеле спадају међу најкорисније структуре података. Користе се за уметање и тражење елемената, а у неким варијантама и за брисање. Стога, представљају ефикасну структуру података за имплементацију речника. Иако тражење елемента у хеш табели може да траје као и тражење елемента у повезаној листи ($O(n)$ у најгорем случају), у просеку се операције са хеш табелом извршавају прилично ефикасно. Под неким разумним претпоставкама просечно време извршавања операције којом се тражи елемент у хеш табели је $O(1)$.

Основна идеја је једноставна. Ако треба сместити податке са кључевима из опсега од 0 до $n-1$, а на располагању је низ дужине n, онда се податак са кључем i смешта на позицију i, $i = 0, 1, 2, \ldots, n-1$. Овај начин смештања података називамо **директно адресирање**. Ако су кључеви података из опсега од 0 до $2n-1$, још увек је згодно податке сместити на исти начин у низ дужине $2n$ — тиме се постиже највећа ефикасност, која надокнађује утрошак меморијског простора. Можемо искористити предности директног адресирања све док можемо да приуштимо да алоцирамо низ који има једну позицију за сваку могућу вредност кључа. У овој ситуацији тривијално је имплементирати све три операције речника – претпоставимо да користимо табелу $T[0..n-1]$, при чему свака позиција табеле одговара једној вредности кључа. Уколико овај скуп не садржи елемент са кључем k, онда је $T[k] =$ **NULL** .



Слика 2.24: Имплементација динамичког скупа коришћењем директног адресирања. Сваки кључ из скупа $U = \{0, 1, 2, \ldots, 9\}$ одговара индексу у табели.

DirektnoAdresiranje_Search(T, k)
улаз: T - табела, k - вредност кључа
излаз: показивач на податак из табеле који има вредност кључа k
или **NULL** ако такав податак не постоји
1 **return** $T[k]$

DirektnoAdresiranje_Insert(T, x)
улаз: T - табела, x - податак

излаз: измењена табела T
1 $T[x.kljuc] \leftarrow x$

DirektnoAdresiranje_Delete(T, x)
улаз: T - табела, x - податак
излаз: измењена табела T
1 $T[x.kljuc] \leftarrow$ **NULL**

Свака од ових операција има време извршавања $O(1)$.

Ситуација се мења ако је опсег вредности кључева од 1 до M толико велики да се низ дужине M не може сместити на рачунару. Тада отпада варијанта са коришћењем низа дужине M. Претпоставимо, на пример, 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 низу могу се користити само његове три последње цифре: тада је за смештање података довољан низ 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 $\mid U \mid = M \gg n$. Идеја је да кључеве сместимо у табелу величине m, тако да m није много веће од n. Потребно је дакле дефинисати функцију:

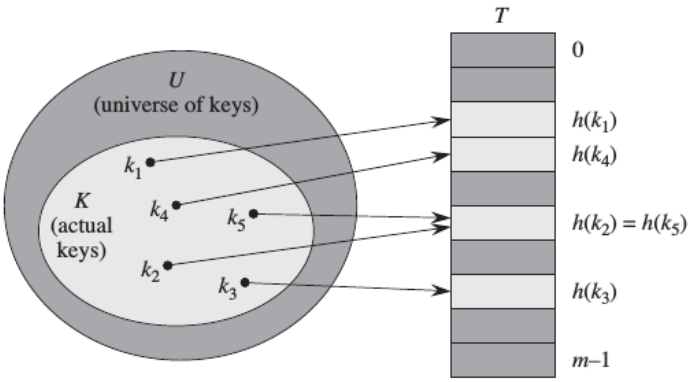
$$h : \{0, 1, \ldots, M-1\} \rightarrow \{0, 1, \ldots, m-1\}$$

тзв. **хеш функцију** која за сваки податак одређује његову позицију на основу вредности одговарајућег кључа (кажемо и да је $h(k)$ хеш вредност кључа k). 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. Без обзира на величину m табеле, 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



Слика 2.25: Коришћење хеш функције h за пресликавање кључева у позиције табеле. С обзиром на то да се кључеви k_2 и k_5 пресликавају у исту позицију, долази до колизије.

функцијом пресликава 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 је, на пример, погрешно 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.

Ако занемаримо колизије, све три основне операције са хеш табелом извршавају се за време O(1).

2.9.1 Хеш функције

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

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

С обзиром на то да захтева само једно дељење, вредности овакве хеш функције се лако израчунавају.

Ако пак m није прост број, на пример $m = 2^k$, онда би се оваквом хеш функцијом издвајало најнижих k битова кључа (што није добро осим ако поуздано знамо да су остаци кључева при дељењу са m равномерно расподељени). Ако су сви кључеви парни бројеви, онда се пола хеш табеле (са непарним индексима) не користи. Могуће решење проблема је да се користи хеш функција:

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

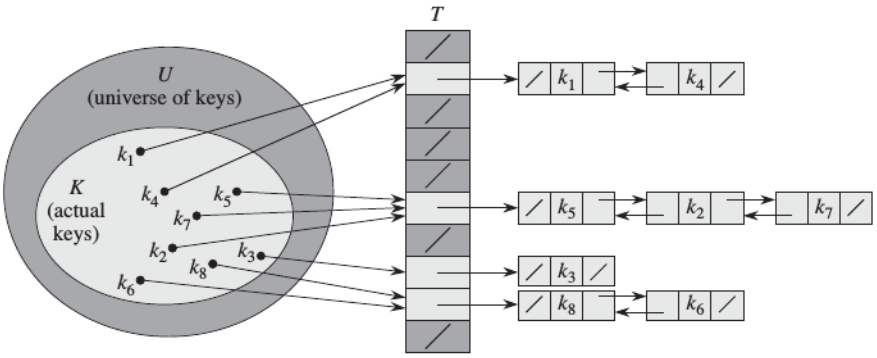
где је p прост број такав да је $m \ll p \ll M$.

Неугодна је ситуација кад су сви кључеви облика $r + kp$, за неки цели број r, јер ће сви кључеви имати исту вредност хеш функције $r \mod m$.

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.

2.9.2 Обрада колизија

Најједноставији начин за обраду колизија је тзв. **одвојено низање**. У овом приступу сваки елеменат хеш табеле је показивач на повезану листу, која садржи све кључеве 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.



Слика 2.26: Разрешавање колизије коришћењем одвојеног низања. Свака позиција $T[j]$ табеле садржи показивач на повезану листу свих елемената чија је вредност кључа j. Повезана листа може бити једностуко или двоструко повезана: овде је приказана двоструко повезана листа јер омогућава брже брисање.

Што се брисања тиче, елемент пронађен по вредности кључа може се

обрисати за време $O(1)$.

Што се претраге тиче, она зависи од **фактора попуњености** хеш табеле. Наиме, уколико имамо хеш табелу T величине m у којој је смештено n кључева, фактор попуњености је $\alpha = n/m$. У најгорем случају свих n кључева се пресликавају у исту локацију табеле и креирају листу од n елемената, те је време извршавања претраге $O(n)$ (што обухвата и време потребно за израчунавање вредности хеш функције). Понашање у просечном случају зависи од особина хеш функције. Ако претпоставимо да је једнако вероватно да се сваки од кључева пресликa у неку позицију табеле, тада је очекивана дужина сваке листе $n/m = \alpha$. Уз претпоставку да се вредност хеш функције израчунава у времену $O(1)$, може се доказати да је просечно време извршавања претраге $O(1 + \alpha)$. Под разумном претпоставком да је величина хеш табеле пропорционална броју елемената у табели, време извршавања је $O(1)$.

OdvojenoNizanje_Insert(T, x)
улаз: T - табела, x - показивач на податак
излаз: измењена табела T
1 **if** x се не налази у листи $T[h(x.kljuc)]$
2 уметни x на почетак листе $T[h(x.kljuc)]$

OdvojenoNizanje_Search(T, k)
улаз: T - табела, k - вредност кључа која се тражи
излаз: податак из табеле који има вредност кључа k
или **NULL** ако такав податак не постоји
1 тражи елемент са кључем k у листи $T[h(k)]$

OdvojenoNizanje_Delete(T, x)
улаз: T - табела, x - показивач на податак који треба обрисати
излаз: измењена табела T
1 обриши x из листе $T[h(x.kljuc)]$

Други начин за обраду колизија представља **отворено адресирање**. У овом приступу, сви елементи се смештају у низ – хеш табелу. Стога сваки елемент хеш табеле садржи или кључ или вредност **NULL** . Приликом потраге за кључем, систематски се претражују позиције табеле док се не нађе тражени кључ или се установи да он није ту. Нема листи, нити елемената који се смештају ван табеле. Да би се елемент сместио у табелу, сукцесивно се испитују позиције хеш табеле док се не пронађе празна позиција на коју се смешта кључ.

Најједноставнији метод отвореног адресирања 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 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 се 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 или се мора користити тзв. могућност "лењог брисања": тада се приликом брисања елемента из табеле уместо вредности **NULL** уписује нека друга резервисана вредност која означава да се на тој позицији налазио елемент, али је обрисан.

LinearnoPopunjavanje_Insert(T, k)
улаз: T - табела, k - вредност кључа
излаз: индекс позиције на којој је записан кључ k и измењена табела T

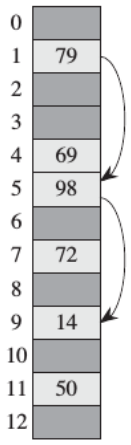
```
1 i ← 0
2 repeat
3    j ← (h(k) + i)  mod m
4    if T[j] =  NULL
5       T[j] ← k
6       return j
7    else i ← i + 1
8 until i = m
9 error "прекорачење хеш табеле"
```

LinearnoPopunjavanje_Search(T, x)
улаз: T - табела, x - податак
излаз: индекс кључа k у табели T
или **NULL** ако k није у табели

```
1 i ← 0
2 repeat
3    j ← (h(k) + i)  mod m
4    if T[j] = k
6       return j
7    i ← i + 1
8 until T[j] ≠  NULL  or i = m
9 return  NULL
```

Ефекат груписања може се ублажити **двоструким хеширањем**. Кад дође до колизије при упису елемента x на позицију i, $h(x) = i$, 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 са локацијом $(i+h_2(x)+h_2(y)) \bmod m$ уместо са $(i+h_2(x)+h_2(x)) \bmod m$. На овај начин, за разлику од линеарног попуњавања, корак са којим тражимо нову позицију није фиксан, већ зависи од самог кључа. 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 да би било који кључ могао да буде уписан на произвољно место у табели, а то важи ако су вредности $h_2(x)$ узајамно просте са m.



Слика 2.27: Пример уметања приликом двоструког хеширања. Дата је табеле величине 13 при чему је $h_1(k) = k \bmod 13$ и $h_2(k) = 1 + (k \bmod 11)$. С обзиром на то да је $14 = 1 (\bmod 13)$ и $14 = 3 (\bmod 11)$, смештамо кључ 14 на позицију 9, након испитивања позиција 1 и 5 које су заузете.

DvostrukoHeshiranje_Insert(T, k)
улаз: T - табела, k - вредност кључа
излаз: индекс позиције на којој је записан кључ k и измењена табела T
1 $i \leftarrow 0$
2 **repeat**
3 $\quad j \leftarrow (h_1(k) + i \cdot h_2(k)) \mod m$
4 $\quad$ **if** $T[j] =$ **NULL**
5 $\quad\quad T[j] \leftarrow k$
6 $\quad\quad$ **return** j
7 $\quad$ **else** $i \leftarrow i + 1$
8 **until** $i = m$
9 **error** "прекорачење хеш табеле"

DvostrukoHeshiranje_Search(T, x)
улаз: T - табела, x - податак
излаз: индекс позиције у табели T на којој је пронађен кључ k
$\quad$ или **NULL** ако k није у табели

```
1 i ← 0
2 repeat
3    j ← (h_1(k) + i · h_2(k))  mod m
4    if T[j] = k
6       return j
7    i ← i + 1
8 until T[j] ≠ NULL  or i = m
9 return  NULL
```

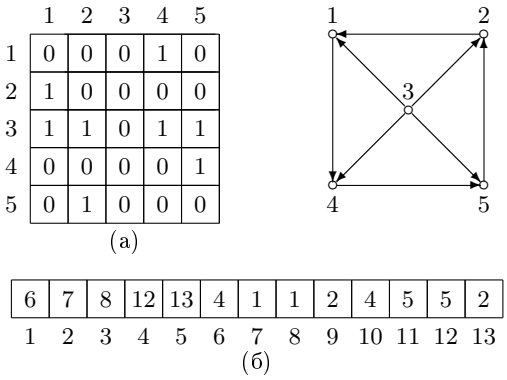
2.10 Графови

Алгоритми за рад са графовима биће детаљно разматрани у наставку текста. 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 је квадратна матрица $A = (a_{ij})$ реда n, са елементима a_{ij} који су једнаки 1 ако и само ако $(v_i, v_j) \in E$; остали елементи матрице A су нуле. Ако је граф неусмерен, матрица A је симетрична. Врста i ове матрице је дакле низ 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 низ 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 низом листа. Сваки елеменат низа 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 низовима на следећи начин. Користи се низ дужине $|V| + |E|$. Првих $|V|$ чланова низа су придружени чворовима. Компонента низа придружена чвору v_i садржи индекс почетка списка чворова суседних чвору v_i, $i = 1, 2, \ldots, n$. На слици 2.28 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.



Слика 2.28: Представљање графа матрицом повезаности (а), односно листом повезаности (б).

2.11 Задаци за вежбу

1. Допунити кодове алгоритама Enqueue и Dequeue тако да откривају грешке прекорачења, односно поткорачења.

2. Написати кодове алгоритама за извршавање четири операције са двоструким редом (редом са операцијама уметања/избацивања елемената са леве/десне стране).

3. Како се може направити стек од два реда? Како се може направити ред од два стека?

   Нека су дата два реда: Q_1 и Q_2 и нека је увек један од њих активан, нпр. на почетку нека то буде ред Q_1. Операција Push убацује елемент у активни ред (то се ради коришћењем операција за рад са редом, тј. елемент се убацује на крај реда Q_1). Операција Pop се имплементира на следећи начин: избацују се сви осим једног елемента из активног реда и убацују у неактивни. Улоге редова затим треба заменити, тако да активни ред постане неактиван, а неактиван активан, а последњи елемент у сада неактивном реду се враћа као резултат операције Pop. Овако имплементирана операција Push се извршава у времену $O(1)$, а операција Pop у времену $O(n)$, где је n тренутни број елемената на стеку.

   Нека су дата два стека: S_1 и S_2. Операција Enqueue умеће елемент у стек S_2. Операција Dequeue избацује елемент из стека S_1. Уколико је стек S_1 празан, један по један елемент стека S_2 се скида са стека S_2 и умеће у стек S_1. На овај начин се они појављују у стеку S_1 у обрнутом редоследу у односу на стек S_2. Операција Enqueue се извршава у константном времену ($O(1)$). Операција Dequeue се извршава у времену

$O(n)$, али то се дешава само када је стек S_1 празан. Просечно време извршавања ове процедуре је $O(1)$, јер се сваки елемент премешта константан број пута.

4. Како се могу имплементирати два стека коришћењем једног низа? Имплементирати основне операције за рад са овим стековима.

5. Како се може имплементирати стек коришћењем реда са приоритетом?

6. Написати кодове алгоритама за операције са стеком реализованим помоћу једноструко повезане листе. Време извршавања операција Push и Pop треба да буде $O(1)$.

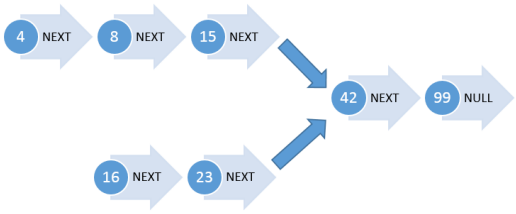
7. Написати кодове алгоритама за операције са редом реализованим помоћу једноструко повезане листе. Време извршавања операција Enqueue и Dequeue треба да буде $O(1)$.

8. Како се може извршити упис у једноструко повезану листу за време $O(1)$? Брисање?

9. Написати кодове алгоритама за операције Insert, Delete, Search са једноструко повезаном циркуларном листом. Колико је време извршавања тих операција?

10. Како се може обрнути редослед елемената у једноструко повезаној листи дужине n за време $O(n)$, тако да користи допунски меморијски простор величине $O(1)$?

11. Дати су показивачи на две једноструко повезане листе које на неком месту у имају заједнички елемент, после кога су и сви остали елементи заједнички (видети слику 11). Конструисати алгоритам линеарне временске сложености којим се одређује адреса чвора у коме се ове две листе сусрећу.



12. Структура података треба да омогући уметање елемента, брисање и тражење максималног елемента. Да бисмо имплементирали операцију за тражење максимума у константном времену, зашто не чувамо само максималну вредност која је до тада уметнута у структуру података и не вратимо њу?

13. Дат је низ у коме се редом на позицијама 1 до 11 налазе вредности: 15, 10, 12, 9, 5, 8, 2, 4, 7, 3, 1.

- Проверити да ли дати низ представља исправан имплицитно представљени хип (ако је потребно извршити потребне поправке).
- Додати у хип елемент 14
- Након тога издвојити највећи елемент хипа.

14. Где се у хипу може наћи најмањи елемент, под претпоставком да су сви елементи хипа различити?

15. Показати да у хипу са n елемената има највише $\lceil n/2^{h+1} \rceil$ чворова висине h.

16. 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 $O(\log(m+n))$.

17. Осмислити структуру података која подржава уметање елемента у времену $O(\log n)$, налажење медијане у константном времену и избацивање медијане из структуре података у времену $O(\log n)$.

18. Колико најмање чворова може да има бинарно стабло имплицитно представљено низом дужине 16?

19. За скуп кључева {1, 4, 5, 10, 16, 17, 21}, нацртати бинарно стабло претраге висине 2, 3, 4, 5, односно 6.

20. Претпоставимо да имамо бројеве између 1 и 1000 у бинарном стаблу претраге и да у стаблу тражимо број 363. Који од наредних низова не би могао да буде низ испитиваних кључева чворова:

    а) 2, 252, 401, 398, 330, 344, 397, 363.

    б) 924, 220, 911, 244, 898, 258, 362, 363.

    ц) 925, 202, 911, 240, 912, 245, 363.

    д) 2, 399, 387, 219, 266, 382, 381, 278, 363.

    е) 935, 278, 347, 621, 299, 392, 358, 363.

21. Нека се пут при тражењу кључа k завршава у листу. Разматрамо три скупа: A - кључеви лево од пута, B - кључеви на путу и C - кључеви десно од пута. Да ли за произвољна три кључа $a \in A, b \in B, c \in C$ важи $a < b < c$? Ако не, пронаћи контрапример са најмањим могућим бројем чворова.

22. Можемо сортирати дати скуп од n бројева тако што прво изградимо бинарно стабло претраге које садржи ове бројеве, а затим излистамо бројеве у растућем поретку. Које је најбоље, а које најгоре време извршавања овог алгоритма за сортирање?

23. Трансформисати једноставну рекурзивну процедуру тражења броја у бинарном стаблу претраге у нерекурзивну процедуру.

24. Конструисати нерекурзивни алгоритам који, коришћењем стека као помоћне структуре, штампа кључеве свих чворова бинарног стабла претраге у растућем редоследу.

25. Приказати промене кроз које пролази АВЛ стабло, на почетку празно, после уметања редом елемената 7,4,12,6,5,18,20,10,11.

26. Да ли важи да је у АВЛ стаблу разлика у дужини између најкраћег и најдужег пута од корена до листа највише 1?

27. Претпоставимо да су елементи скупа D смештени у АВЛ стабло. Како у времену $O(\log n)$ пронаћи број елемената скупа D за чије вредности кључева важи да је $k_1 \leq k \leq k_2$, где су k_1 и k_2 две унапред фиксиране вредности?

28. За дата два АВЛ стабла T_1 и T_2 за која важи да је највећи кључ у стаблу T_1 мањи од најмањег кључа у стаблу T_2, конструисати алгоритам за спајање ова два АВЛ стабла у једно, који је временске сложености $O(h)$, где је h већа од висина два АВЛ стабла.

29. Претпоставимо да је динамички скуп S представљен табелом T дужине m коришћењем директног адресирања. Описати процедуру којом се проналази максимални елемент скупа S. Колико је време извршавања предложене процедуре у најгорем случају?

30. Претпоставимо да користимо хеш функцију h за смештање n различитих кључева у низ T дужине m. Под претпоставком униформне расподеле вероватноће кључева, који је очекивани број колизија? Прецизније, која је очекивана кардиналност скупа $\{\{k.l\} : k \neq l \textbf{ and } h(k) = h(l)\}$?

31. Демонстрирати шта се дешава приликом уметања кључева 5, 28, 19, 15, 20, 33, 12, 17, 10 у хеш табелу у којој се колизија разрешава уланчавањем. Нека је хеш табела величине 9, а нека је одговарајућа хеш функција $h(k) = k \mod 9$.

32. Шта би се десило уколико бисмо изменили шему уланчавања тако да се свака од листа чува у сортираном редоследу? Како би то утицало на време извршавања успешне претраге, неуспешне претраге, уметања и брисања?

33. Предложити како алоцирати и деалоцирати простор за елементе хеш табеле повезивањем свих неискоришћених позиција у листу празних позиција. Претпоставити да једна позиција може да чува флег и или један елемент плус показивач или два показивача. Све операције речника и операције над листом празних позиција треба да се извршавају у константном времену. Да ли листа празних позиција треба да буде двоструко повезана или је довољно да буде једноструко повезана?

34. Размотримо уметање кључева 10, 22, 31, 4, 15, 28, 17, 88, 59 у хеш табелу дужине $m = 11$ коришћењем отвореног адресирања и помоћне хеш функције $h'(k) = k$. Приказати резултат уметања датих кључева коришћењем линеарног попуњавања и двоструког хеширања коришћењем хеш функција: $h_1(k) = k, h_2(k) = 1 + (k \mod (m + 1))$.

35. Конструисати алгоритме за утврђивање да ли је скуп A подскуп скупа B коришћењем имплементације скупа (а) помоћу уређених низова (б) помоћу битских низова.

36. Имплементирати поступак издвајања свих простих бројева мањих или једнаких од задатог броја n Ератостеновим ситом коришћењем битског низа.

Глава 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.

Основни задатак који се решава је следећи: датих n бројева $x_1, x_2, \ldots, x_n$ треба уредити у неопадајући низ. Другим речима, треба пронаћи низ различитих индекса $1 \leq i_1, i_2, \ldots, i_n \leq n$ таквих да је $x_{i_1} \leq x_{i_2} \leq \cdots \leq x_{i_n}$.

Због једноставности, ако се не нагласи другачије, претпостављамо да су бројеви x_i различити (иако, наравно, алгоритме конструишемо тако да 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), односно дозвољено је користити меморијски простор величине $O(1)$.

3.1 Хип сорт

Хип сорт је један од ефикасних алгоритама за сортирање. У пракси он, за велике n, обично није тако брз као сортирање раздвајањем (квик сорт), али није ни много спорији. С друге стране, за разлику од сортирања раздвајањем (видети одељак 3.3), његова ефикасност је гарантована. Као код сортирања обједињавањем, време извршавања хип сорта у најгорем случају је $O(n \log n)$.

За разлику од сортирања обједињавањем, хип сорт је алгоритам сортирања у месту. У вези са хип сортом обратићемо посебно пажњу на почетни део алгоритма — формирање хипа. Алгоритам за формирање хипа илуструје начин на који треба комбиновати конструкцију и анализу алгоритма.

Претпоставићемо да се хип представља имплицитно; његови елементи смештени су у низ 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 хип сорт извршава се на исти начин као и сортирање избором; разлика је у употребљеној структури података за смештање низа. Најпре се

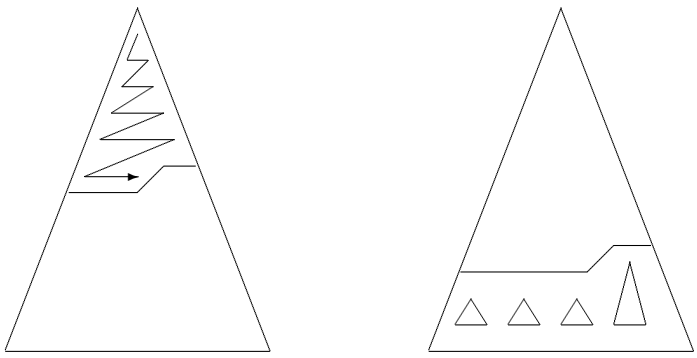
елементи низа преуређују тако да чине хип; формирање хипа размотрићемо нешто касније. Ако је хип у низу A, онда је $A[1]$ највећи елемент хипа. Заменом $A[1]$ са $A[n]$ постиже се да је највећи елемент низа доведен на своје место $A[n]$ у сортираном редоследу. Затим се разматра низ $A[1], A[2], \ldots, A[n-1]$; преуређује се, тако да и даље задовољава услов хипа (услов хипа може да не задовољава само нови елемент $A[1]$). 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. Формирање хипа је само по себи интересантан проблем, па ће бити посебно разматран. Временска сложеност сортирања полазећи од формираног хипа је дакле $O(n \log n)$ ($O(\log n)$ по замени). Јасно је да је хип сорт сортирање у месту.

$Hipsort(A, n)$
улаз: A - низ од n елемената
излаз: A - сортирани низ
1 преуредити A тако да буде хип
2 **for** $i \leftarrow n$ **downto** 2
3 $\quad zameni(A[1], A[i])$
4 $\quad$ преуредити део низа $A[1..i-1]$ тако да буде исправан хип

Формирање хипа

Сада ћемо се позабавити проблемом формирања хипа од произвољног низа. Дат је низ $A[1], A[2], \ldots, A[n]$ произвољно уређених бројева. Треба преуредити његове елементе тако да задовољавају услов хипа.

Постоје два природна приступа формирању хипа: одозго надоле и одоздо нагоре. Они одговарају проласку кроз низ A слева удесно, односно здесна улево (видети слику 3.1). Пошто ови методи буду описани применом индукције, биће показано да између њих постоји знатна разлика у ефикасности.



Слика 3.1: Формирање хипа одозго надоле и одоздо нагоре.

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

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

Табела 3.1: Пример формирања хипа одоздо нагоре. Бројеви у првој врсти су индекси. Бројеви који су управо учествовали у замени, подебљани су.

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 $A[1..i]$. Елемент $A[i+1]$ упоређује се са својим оцем, и замењује се са њим ако је већи од њега; нови елемент 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 $A[i+1..n]$ хип, у који треба убацити елеменат $A[i]$. Али низ $A[i+1..n]$ не одговара једном, него колекцији хипова (низ $A[i+1..n]$ посматрамо као део стабла представљеног низом $A[1..n]$). Због тога је индуктивна хипотеза нешто компликованија.

Индуктивна хипотеза (одоздо нагоре): Сва стабла са коренима на индексима $i+1, i+2, \ldots, n$ задовољавају услов хипа.

Индукција је по i, али обрнутим редоследом, $i = n, n-1, \ldots, 1$. Елемент $A[n]$ очигледно представља хип, што представља базу индукције. Може се закључити и нешто више. Елементи низа A са индексима од $\lfloor n/2 \rfloor + 1$ до n су листови стабла. Због тога се подстабла која одговарају елементима тог низа 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.

Размотримо сада елемент $A[i]$. Он има највише два сина $A[2i+1]$ и $A[2i]$, који су, према индуктивној хипотези, корени исправних хипова. Због тога је јасан начин сређивања хипа са кореном на индексу i: $A[i]$ се упоређује са већим од синова, и замењује се са њим ако је потребно. Ово је слично брисању из хипа. Са заменама се наставља наниже низ стабло, док претходна вредност елемента $A[i]$ не дође до листа или до места на коме је већа од вредности оба сина. Конструкција хипа одоздо нагоре илустрована је примером у табели 3.1.

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

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

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 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. сменом $F(i) = G(i) + 2$ добија се $F(i) = 2F(i-1)$ 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 $n = 2^{i+1} - 1 > H(i)$, закључујемо да је за хипове са $n = 2^{i+1} - 1$ чворова временска сложеност формирања хипа одоздо нагоре $O(n)$.

Формирање хипа од n елемената, при чему је $2^i \leq n \leq 2^{i+1} - 1$ траје мања или једнако од трајања формирања хипа од $2^{i+1} - 1$ елемената, које 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.

Овај пример 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.

3.2 Сортирање разврставањем

Најједноставнији поступак сортирања састоји се у томе да се обезбеди довољан број локација, и да се онда сваки елемент 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.

[1] За дефиницију асимптотске ознаке Ω погледати поглавље 4.2

Нека је дато n елемената, који су цели бројеви из опсега од 1 до $m \geq n$. 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 **линеарне 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 велико m.

3.3 Сортирање раздвајањем (квик сорт)

Сортирање обједињавањем пример је приступа конструкцији алгоритама заснованог на разлагању. Ако нам пође за руком да поделимо проблем на 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 елемент где завршити. Може ли се декомпозиција применити на други начин, тако да се положаји елемената могу предвидети? Идеја сортирања раздвајањем је да се *највећи део 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 се може упоредити са свим елементима низа (коришћењем укупно $n-1$ упоређивања) и тиме се елементи могу 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 елементи првог, односно другог дела после сортирања остају у првом, односно другом делу. Према томе, није потребан допунски меморијски простор (изузев простора на стеку за памћење рекурзивних позива).

При описаној конструкцији претпостављено је да је вредност x позната, што обично није случај. Може се показати да предложени алгоритам добро ради без обзира на то који се број x (називамо га **пивот**) користи за раздвајање. Циљ је раздвојити елементе низа на два дела, један са бројевима мањим или једнаким, а други са бројевима већим од пивота. Раздвајање се може остварити коришћењем следећег алгоритма. Користе се два индекса за низ, L и D. На почетку L показује на леви, а D на десни крај низа. Индекси 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 индекс 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 и x_D и померамо оба индекса даље према унутра. У оба случаја имамо померање бар једног индекса. Према томе, индекси ће се у једном тренутку сусрести, и циљ је постигнут.

Остаје проблем избора доброг пивота и прецизирања последњег корака алгоритма кад се индекси сретну. Алгоритми засновани на разлагању најбоље раде кад делови имају приближно исте величине, што сугерише да што је пивот ближи средини, то ће алгоритам бити ефикаснији. Као што ћемо видети у оквиру анализе алгоритма, избор случајног елемента низа је добро решење. Ако је полазни низ добро "испретуран", онда се за пивот може узети први елемент низа. Због једноставности је у приказаном алгоритму управо тако и урађено.

$Razdvajanje(X, Levi, Desni)$
улаз: X - низ, $Levi$ - лева граница низа, $Desni$ - десна граница низа
излаз: X - измењени низ и индекс S такви да важи:
$X[i] \leq X[S]$ за све $i \leq S$ и $X[j] > X[S]$ за све $j > S$

```
1  pivot ← X[Levi]
2  L ← Levi
3  D ← Desni
4  while L < D do
5     while X[L] ≤ pivot and L ≤ Desni do
6        L ← L + 1
7     while X[D] > pivot and D ≥ Levi do
8        D ← D − 1
9     if L < D then
10       zameni(X[L], X[D])
11 {стали смо када је вредност елемента са индексом D мања од пивота,
12  па њу треба ставити на прву позицију низа}
13 S ← D
14 zameni(X[Levi], X[S])
15 return S
```

Ако се први елемент 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.

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

Табела 3.2: Раздвајање низа око пивота 6.

Пример рада алгоритма *Razdvajanje* дат је у табели 3.2. Пивот је први број (6). Бројеви који су управо замењени су подебљани. После три замене и померања L је индекс елемента $X[7] = 12$, а D индекс елемента $X[6] = 1$ (индекс 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 (са индексима од 1 до 6, односно од 7 до 16) могу да се рекурзивно сортирају. У најгорем случају дубина рекурзије је $O(n)$, а у просеку $O(logn)$. Пример рада алгоритма приказан је у табели 3.3.

$Quicksort(X, n)$
улаз: X - низ од n бројева
излаз: X - сортирани низ
1 $QSort(X, 1, n)$

$QSort(X, Levi, Desni)$
улаз: X - низ са индексима у интервалу $[Levi, Desni]$
излаз: X - сортирани низ у интервалу $[Levi, Desni]$
2 **if** $Levi < Desni$ **then**
3 $S = Razdvajanje(X, Levi, Desni)$ {S је индекс пивота после раздвајања}
4 $QSort(X, Levi, S - 1)$
5 $QSort(X, S + 1, Desni)$

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

Табела 3.3: Пример сортирања раздвајањем. Пивоти су подебљани. Елементи који се у некој фази не мењају, нису приказани.

Сложеност: Време извршавања 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. На пример, ако је пивот најмањи елемент у низу, онда прво раздвајање захтева $n - 1$ упоређивања а резултат је раздвајање бројева на две групе, са 0 односно $n - 1$ елемената. Због тога, ако је низ већ уређен растуће и за пивот се увек бира први елемент, 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 елемент низа. Ипак, вероватноћа да дође до овог најгорег случаја је мала. Размотрићемо то сада мало прецизније.

Претпоставимо да се сваки елемент 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* 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 њено решење је $T(n) = O(n \log n)$ (као што ћемо видети у делу курса о анализи алгоритама). Дакле, сортирање раздвајањем је заиста ефикасно у просеку.

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

3.4 Доња граница сложености за сортирање

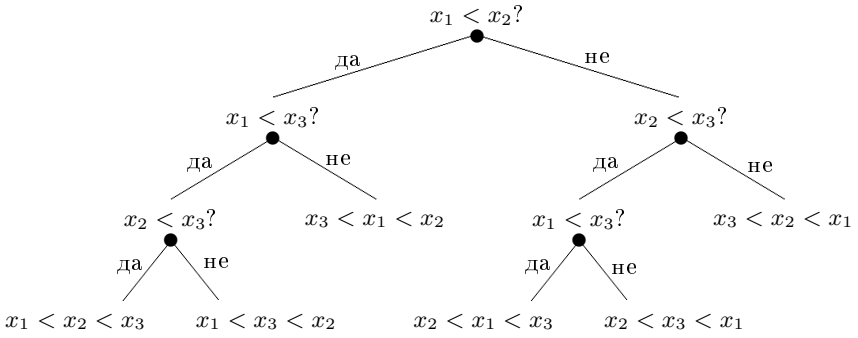
Општи алгоритми за сортирање које смо размотрили имају време извршавања $O(n^2)$ или $O(n \log n)$; сортирање разврставањем није општи алгоритам, јер се не може применити на произвољан низ. Поставља се питање да ли је могуће даље побољшање, односно да ли је сложеност сваког алгоритма за сортирање бар реда $n \log n$?

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 дефинисати *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 3.2 приказано је сортирање помоћу стабла одлучивања за $n = 3$. Временска сложеност (у најгорем случају) алгоритма дефинисаног стаблом T једнака је висини T, односно највећем 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 само на алгоритме *тог облика*. Сада ћемо искористити стабла одлучивања да бисмо одредили доњу границу за сортирање.

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

Доказ: 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 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 ње-



Слика 3.2: Сортирање три елемента помоћу стабла одлучивања.

говим 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 $\log_2(n!)$. Ако искористимо чињеницу да важи $k > n/2$ за $k = n/2, \ldots, n$ добијамо:

$$\log_2(n!) > \log_2(n/2 \cdot n/2 \cdot \ldots \cdot n/2)$$

при чему се фактор $n/2$ јавља $n/2$ пута. Одавде следи:

$$\log_2(n!) > \log_2(n/2)^{n/2} = n/2 \cdot \log_2(n/2)$$

односно:

$$\log_2(n!) = \Omega(n \log n)$$

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 што чак и једноставни алгоритми линеарне временске сложености могу да одговарају стаблима одлучивања са експоненцијалним бројем чворова. Величина стабла није битна, јер се

стабло у ствари не формира експлицитно. Показало 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.

3.5 Задаци за вежбу

1. Детаљно описати поступак формирање хипа

   а) одозго надоле

   б) одоздо нагоре

   од низа елемената $A = (5, 13, 2, 25, 7, 17, 20, 8, 4)$.

2. Доказати коректност алгоритма *Hipsort* коришћењем наредне инваријанте петље:

   на почетку сваке итерације **for** петље (у линијама 2–4) подниз $A[1..i]$ је хип који садржи i најмањих елемената низа $A[1..n]$, а подниз $A[i+1..n]$ садржи $n-i$ највећих елемената низа $A[1..n]$ у сортираном редоследу.

3. Колико је време извршавања алгоритма *Hipsort* на низу A дужине n који је већ сортиран растуће, а колико ако је низ сортиран опадајуће?

4. Показати како је могуће коришћењем хипа реализовати наредну операцију:

   $Povecaj\_kljuc(S, i, k)$ – којом се вредност кључа елемента са индексом i повећава на вредност k, за коју се претпоставља да има вредност већу или једнаку претходној вредности кључа са индексом i.

5. Конструисати алгоритам временске сложености $O(n \log k)$ за обједињавање k сортираних низова у један сортирани низ од укупно n елемената.

6. Демонстрирати извршавање алгоритма *Razdvajanje* на низу: $(13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21)$, уколико се први елемент бира за пивот.

7. Која је најмања могућа дубина листа у стаблу одлучивања за проблем сортирања заснован на упоређивању?

8. Дат је скуп S од n различитих целих бројева. Конструисати алгоритам сложености $O(n)$ за налажење неког броја који није у S. Доказати да је $\Omega(n)$ доња граница сложености за број корака потребних да се реши овај проблем.

Глава 4

Анализа алгоритама

4.1 Увод

Циљ анализе алгоритма је да се предвиди његово понашање, посебно брзина извршавања, не реализујући га на (неком конкретном) рачунару. Дакле потребно је проценити брзину рада алгоритма и то тако да та 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**. Међутим, на тај начин се ипак 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 приближно за константни фактор. Од интереса 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 линеарна, односно пропорционална са n; ако је број корака $2n^2 + 50$, онда се каже да је сложеност алгоритма квадратна, односно пропорционална са 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 $n > 50$. Слично, ако се алгоритам састоји од $100n^{1.8}$ корака, неједнакост $100n^{1.8} < 2n^2 + 50$ почиње да важи тек за $n > 3 \cdot 10^8$. 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. битовима. Непостојање опште дефиниције величине улаза не уноси забуну, јер се обично упоређују различити алгоритми за решавање истог проблема.

Дакле, анализа алгоритма треба да као резултат дâ израз за утрошено време у зависности од величине улаза 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.

4.2 Асимптотска ознака O

Дефиниција: 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 у ствари 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 $5n^2 + 15 \leq 6n^3$ за $n \geq 4$). Ова нотација омогућује игнорисање мултипликативних константи: уместо $O(5n + 4)$ може се писати $O(n)$ јер класе $O(5n + 4)$ и $O(n)$ садрже исте функције. Једнакост $O(5n + 4) = O(n)$ је у ствари једнакост две класе функција. 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**.

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

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

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:

$$\begin{aligned} O(f(n)) + O(g(n)) &= O(f(n) + g(n)), \\ O(f(n))O(g(n)) &= O(f(n)g(n)). \end{aligned}$$

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 $O(f(n)g(n)))$.

Покажимо да важе ове две једнакости. 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 $r(n) \leq c_1 f(n)$ и за $n > N_2$ важи $s(n) \leq c_2 g(n)$. Међутим, за $N = \max\{N_1, N_2\}$ и $c = \max\{c_1, c_2, c_1 c_2\}$ важи:

- $r(n) + s(n) \leq c\big(f(n) + g(n)\big)$ и
- $r(n)s(n) \leq cf(n)g(n)$,

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 $f(n) - g(n) = O\big(r(n) - s(n)\big)$.

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 4.1 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.

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 позитивне константе c и N, такве 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

$f(n)$ / 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}

Табела 4.1: Трајање извршавања $f(n)$ операција на рачунару који извршава m операција у секунди, за $n = 1000$.

$f(n) = O(g(n))$ 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 може се прецизирати: ако је $f(n)$ монотоно растућа функција која није ограничена, $a > 1$ и $c > 0$, онда је

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

4.3 Временска и просторна сложеност

Оцена (временске) сложености алгоритма своди се на бројање рачунских операција 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 извршавање алгоритма, при чему се простор за смештање улазних података 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: израчунавање коначних сума и решавање диференцних једначина.

4.4 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: Ако је $f(i) = i$, односно у i-том проласку кроз петљу извршава се i корака, онда је укупан број корака:

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

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 број извршених корака приближно два пута мањи.

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

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

у којој је вредност $S_k(n)$ изражена преко свих претходних вредности $S_j(n)$, $j = 0, \ldots, k-1$.

Ова једнакост добија се на следећи начин:

$$(i+1)^{k+1} = \sum_{j=0}^{k+1} \binom{k+1}{j} i^j = \sum_{j=0}^{k-1} \binom{k+1}{j} i^j + (k+1) i^k + i^{k+1}$$

те ако обе стране поделимо са $(k+1)$ и изразимо i^k добијамо:

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

Сумирањем ових једнакости за $i = 1, 2, \ldots, n$ добијамо тражени израз за $S_k(n)$.

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

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

Пример: Сума геометријске прогресије

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

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: Полазећи од израза за суму геометријске прогресије може се израчунати сума

$$G(n) = \sum_{i=1}^{n} i2^i$$

Заиста, диференцирањем, па множењем са x једнакости

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

добија се

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

Одатле се за $x = 2$ добија $G(n) = \sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2$.

Пример: Свођењем на претходну суму 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:

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

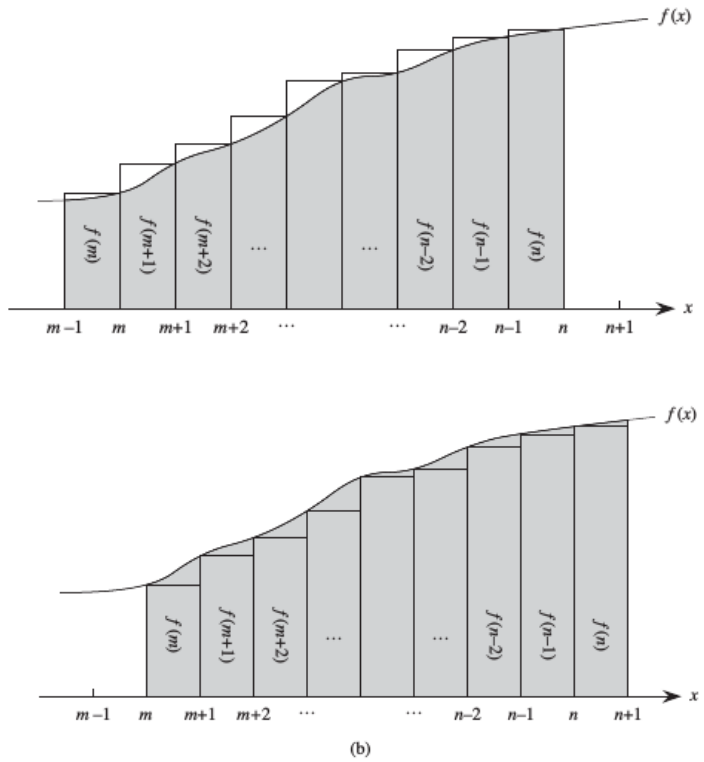
4.4.1 Оцењивање сума помоћу интеграла

У случају кад је тешко израчунати суму $\sum_{i=1}^{n} f(i)$, а функција $f(x)$ од реалног аргумента је монотоно неопадајућа непрекидна функција за $x \geq 1$, тада се сумирањем левих страна за $i = 1, 2, \ldots, n$, а десних страна за $i = 1, 2, \ldots, n-1$ неједнакости:

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

добијају границе интервала у коме лежи сума $\sum_{i=1}^{n} f(i)$:

$$\int_1^n f(x)\,dx + f(1) \leq \sum_{i=1}^{n} f(i) \leq \int_1^{n+1} f(x)\,dx.$$



Слика 4.1: Апроксимација суме $\sum_{k=m}^{n} f(k)$ интегралима. Укупна површина свих правоугаоника представља вредност суме, док је вредност интеграла једанака осенченој површини испод криве. Обратити пажњу да је хоризонтална страница сваког од правоугаоника дужине 1.

видети слику 4.1.

Ако је пак функција $f(x)$ монотоно нерастућа, онда је функција $-f(x)$ неопадајућа, па се применом ових неједнакости на $-f(x)$ добија да у овим неједнакостима само треба променити знак "$\leq$" у знак "$\geq$".

Пример: Размотримо проблем рачунања суме $S_k(n) = \sum_{i=1}^{n} i^k$ за $k > 0$. У овом случају функција $f(x) = x^k$ је монотоно неопадајућа, па се добија процена:

$$\int_1^n f(x)\,dx + f(1) \leq \sum_{i=1}^{n} f(i) \leq \int_1^{n+1} f(x)\,dx$$

односно:

$$\int_1^n x^k\,dx + 1 \leq \sum_{i=1}^{n} i^k \leq \int_1^{n+1} x^k\,dx$$

За $k \neq -1$ је:

$$\int x^k \, dx = \frac{x^{k+1}}{k+1} + C$$

Одавде за $k \neq -1$ добијамо наредну оцену тражене суме:

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

односно:

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

тј:

$$S_k(n) = \sum_{i=1}^{n} i^k = \Theta(n^{k+1})$$

Специјално, за $k = 2$ добијамо:

$$\sum_{i=1}^{n} i^2 = \Theta(n^3),$$

а за $k = 1/2$:

$$\sum_{i=1}^{n} \sqrt{i} = \Theta(n^{3/2}) = \Theta(n\sqrt{n}).$$

Уколико је $k < 0$, функција $f(x) = x^k$ је монотоно опадајућа (за $x > 0$) па за $k \neq -1$ важи наредна оцена:

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

Према томе, за $-1 < k < 0$ је $S_k(n) = \Theta(n^{k+1})$, а за $k < -1$ је $S_k(n) = \Theta(1)$. У случају $k = -1$ важи да је:

$$\int \frac{1}{x} \, dx = \ln x + C$$

и:

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

те важи:

$$\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$$

Пример: Размотримо проблем оцене вредности $\ln(n!) = \sum_{i=1}^{n} \ln i$. Функција $f(x) = \ln x$ је монотоно неопадајућа те можемо користити наредну процену:

$$\int_1^n \ln x\,dx + \ln 1 \leq \sum_{i=1}^{n} \ln i \leq \int_1^{n+1} \ln x\,dx$$

Пошто важи да је $\ln 1 = 0$ и да је $\int \ln x = x \ln x - x$, добијамо оцену:

$$(x \ln x - x)|_1^n \leq \ln n! \leq (x \ln x - x)|_1^{n+1}$$

односно:

$$n \ln n - n + 1 \leq \ln n! \leq (n+1)\ln(n+1) - (n+1) + 1$$

Из ове процене, ако сваку страну неједнакости узмемо као експонент броја e добијамо:

$$e\left(\frac{n}{e}\right)^n \leq n! \leq e\left(\frac{n+1}{e}\right)^{n+1} = e\left(\frac{n}{e}\right)^{n+1}\left(1+\frac{1}{n}\right)^{n+1} \leq e\left(\frac{n}{e}\right)^n (n+1)$$

Прецизнији израз за $n!$:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

зове се Стирлингова формула.

4.5 Диференцне једначине

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

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

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

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

онда је k **ред** те диференцне једначине. Математичком индукцијом се непосредно доказује да је другом диференцном једначином и са првих k чланова F_1, F_2, $\ldots$, F_k низ F_n једнозначно одређен. Другим речима, 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,

$$F_n = F_{n-1} + F_{n-2}, \quad F_1 = F_2 = 1. \tag{4.1}$$

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.

4.5.1 Доказивање претпостављеног решења диференцне једначине

Често се диференцне једначине решавају тако што 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: Посматрајмо диференцну једначину

$$T(2n) = 2T(n) + 2n - 1, \quad T(2) = 1. \tag{4.2}$$

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 надаље 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 покушати са неким изразом 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 (4.2) је решење ове диференцне неједначине — докази индукцијом су практично идентични. Ова диференцна неједначина може се, као и (4.2), 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 се 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 $m = 2k + 1$:

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

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

4.5.2 Линеарне диференцне једначине

Размотрићемо сада хомогене линеарне диференцне једначине облика

$$F(n) = a_1 F(n-1) + a_2 F(n-2) + \cdots + a_k F(n-k). \tag{4.3}$$

Решења се могу тражити у облику:

$$F(n) = r^n, \tag{4.4}$$

где је r погодно изабрани (у општем случају) *комплексни* број. Заменом у (4.3) 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 (4.3); 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 (4.3) облика (4.4). 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: 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:

$$F_n = \frac{1}{\sqrt{5}}\Big(\Big(\frac{1+\sqrt{5}}{2}\Big)^n - \Big(\frac{1-\sqrt{5}}{2}\Big)^n\Big). \tag{4.5}$$

У општем случају, ако су корени карактеристичне једначине $r_1, r_2, \ldots, r_s$ вишеструкости редом $m_1, m_2, \ldots, m_s$ ($\sum m_i = k$), онда се свако решење једначине (4.3) 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: 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 $F(3)$.

4.5.3 Диференцне једначине које се решавају сумирањем

Често се наилази на линеарну нехомогену диференцну једначину облика

$$F(n+1) - F(n) = f(n), \tag{4.6}$$

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 $\sum_{i=1}^{n-1} f(i)$. 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 (4.6) дато је изразом

$$F(n) = \sum_{i=0}^{n-1} f(i) + F(0). \tag{4.7}$$

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

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

4.5.4 Диференцне једначине за алгоритме засноване на разлагању

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 облика

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k, \tag{4.8}$$

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 разлагању. Обично је у оваквим диференцним једначинама b природан број.

Теорема [Мастер теорема]: Асимптотско понашање низа $T(n)$, решења диференцне једначине (4.8) дато је једнакошћу:

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{за } a > b^k \\ O(n^k \log n) & \text{за } a = b^k \\ O(n^k) & \text{за } a < b^k \end{cases}. \tag{4.9}$$

Доказ: Доказ теореме биће спроведен само за подниз $n = b^m$, где је m цели ненегативни број. Помноживши обе стране једнакости (4.8) са a^{-m}/c, добијамо диференцну једначину типа (4.6)

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

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 4.5.3). 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

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

чиме је тврђење теореме доказано.

Алгоритми овог типа имају широку примену због своје ефикасности, па је корисно знати асимптотско понашање решења диференцне једначине (4.8).

4.5.5 Потпуна рекурзија

Диференцна једначина најопштијег облика (4.5) зове се **потпуна рекурзија** или **диференцна једначина са потпуном историјом**. Размотрићемо два примера оваквих диференцних једначина.

Пример: 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 $T(n-1) = c + \sum_{i=1}^{n-2} T(i)$. Одузимањем друге од прве једнакости постиже се жељени ефекат:

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

(за $n \geq 2$), и коначно

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

Приметимо да се једнакост $T(n) = 2T(n-1)$ логаритмовањем и сменом $t_n = \log T(n)$ своди на једначину $t_n = t_{n-1} + \log 2$, односно диференцну једначину типа (4.6).

Пример: Размотрићемо сада сложенији пример. Диференцна једначина

$$T(n) = n - 1 + \frac{2}{n} \sum_{i=1}^{n-1} T(i), \quad n \geq 2,\ T(1) = 0, \tag{4.10}$$

је важна, јер се до ње долази при анализи просечне сложености сортирања раздвајањем. 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 на познат проблем (4.6) 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 $n-1$ добија се ($t_1 = 0$)

$$t_n - t_1 = \sum_{i=1}^{n-1} \frac{2i}{(i+1)(i+2)}. \tag{4.11}$$

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

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

где је са

$$H(n) = \sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n), \tag{4.12}$$

означена парцијална сума хармонијског реда. Заменом вредности суме у (4.11) добија се решење диференцне једначине (4.10)

$$T(n) = (n+1)t_n = \Theta(n \log n)$$

4.6 Задаци за вежбу

1. Да ли важи: $2^{n+1} = O(2^n)$? Да ли важи: $2^{2n} = O(2^n)$?

2. За сваки пар функција (A, B) у датој табели означити да ли важи да је функција A O, o, Ω, Θ од функције B. Претпоставити да су k, ϵ и c константе које задовољавају услове $k \geq 1, \epsilon > 0, c > 0$. У свако од поља табеле уписати да или не.

A	B	O	o	Ω	Θ
$\log^k n$	n^ϵ				
n^k	c^n				
$\sqrt{n}$	$n^{\sin n}$				
2^n	$2^{n/2}$				
$n^{\log c}$	$c^{\log n}$				
$\log(n!)$	$\log(n^n)$				

3. Рангирати наредне функције према брзини раста, тј. пронаћи уређење функција тако да важи $g_1 = \Omega(g_2), g_2 = \Omega(g_3), \ldots$. Партиционисати листу у класе еквиваленција тако да су функције $f(n)$ и $g(n)$ у истој класи ако и само ако важи $f(n) = \Theta(g(n))$:
   $(\sqrt{2})^{\log_2 n}$, n^2, $n!$, $(\log_2 n)!$, $(3/2)^n$, n^3, $\log_2^2 n$, 2^{2^n}, $\ln\ln n$, $n \cdot 2^n$, $\ln n$, 1, $2^{\log_2 n}$, $(\log_2 n)^{\log_2 n}$, e^n, $4^{\log_2 n}$, $(n+1)!$, $\sqrt{\log_2 n}$, n, 2^n, $n \log_2 n$, $2^{2^{n+1}}$.

4. Коришћењем мастер теореме дати асимптотски блиске границе за наредне рекурентне једначине:
   а) $T(n) = 2T(n/4) + 1$
   б) $T(n) = 2T(n/4) + \sqrt{n}$
   ц) $T(n) = 2T(n/4) + n$
   д) $T(n) = 2T(n/4) + n^2$

5. Дати асимптотске горње и доње границе за $T(n)$ ако је:
   а) $T(n) = T(n-1) + 1/n$
   б) $T(n) = T(n-1) + \log n$
   ц) $T(n) = \sqrt{n}T(\sqrt{n}) + n$

6. За сваки од наредних израза за $f(n)$ пронаћи једноставан облик функције $g(n)$ тако да важи: $f(n) = \Theta(g(n))$.
   а) $f(n) = \sum_{i=1}^{n} \frac{1}{i}$
   б) $f(n) = \sum_{i=1}^{n} \lceil \frac{1}{i} \rceil$

7. Поређати наредне функције растуће према брзини раста:
   $f_1(n) = \sum_{i=1}^{n} \sqrt{i}$, $f_2(n) = \sqrt{n}\log n$, $f_3(n) = n\sqrt{\log n}$, $f_4(n) = 12n^{3/2} + 4n$.

8. Одредити функцију f тако да важи $\sum_{i=1}^{n} i \log i = \Theta(f(n))$.

9. За сваки од наредних израза $f(n)$ пронаћи једноставан облик функције $g(n)$ тако да је $f(n) = \Theta(g(n))$.
   а) $f(n) = \sum_{i=1}^{n}(3i^4 + 2i^3 - 19i + 20)$
   б) $f(n) = \sum_{i=1}^{n}(3 \cdot 4^i + 2 \cdot 3^i - i^{19} + 20)$
   ц) $f(n) = \sum_{i=1}^{n}(5^i + 3^{2i})$

10. За сваки од наредних израза $f(n)$ пронаћи једноставан облик функције $g(n)$ тако да је $f(n) = \Theta(g(n))$.
    а) $f_1(n) = 1000 \cdot 2^n + 4^n$
    б) $f_2(n) = n + n\log n + \sqrt{n}$
    ц) $f_3(n) = \log(n^{20}) + (\log n)^{10}$
    д) $f_4(n) = 0.99^n + n^{100}$

Глава 5

Графовски алгоритми

Подсетимо се неких основних појмова у вези са графовима. Граф $G = (V, E)$ се састоји 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). Граф може бити **неусмерен** или **усмерен**. Гране усмереног графа су уређени парови чворова; редослед два чвора које повезује грана је битан. Ако се граф представља цртежом, онда се гране усмереног графа цртају као стрелице усмерене од једног чвора (почетка) ка другом чвору (крају гране). 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** чвора v, број грана за које је чвор v крај, односно **излазни степен** чвора v, број грана за које је чвор v почетак.

Пут од чвора v_1 до чвора v_k је низ чворова $v_1, v_2, \ldots, v_k$ повезаних гранама (v_1, v_2), (v_2, v_3), $\ldots$, (v_{k-1}, v_k). 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 (тако да су парови чворова у E неуређени). 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. Стабло са изабраним чвором – кореном једнозначно одређује коренско стабло.

Граф $H = (U, F)$ је **подграф** графа $G = (V, E)$ ако је $U \subseteq V$ и $F \subseteq E$. **Повезујуће стабло** неусмереног графа G је његов подграф који је стабло и садржи све чворове графа G. **Повезујућа шума** неусмереног графа G је његов подграф који је шума и садржи све чворове графа G. Ако неусмерени 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.

Многе дефиниције појмова 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 прецизирају 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.

5.1 Обиласци графова

Први проблем на који се наилази при конструкцији било ког алгоритма за обраду графа је како *прегледати улаз*. У случају низова и скупова тај проблем је тривијалан због једнодимензионалности улаза — низови и скупови могу се лако прегледати линеарним редоследом. Прегледање графа, односно његов **обилазак**, није тривијалан проблем. Постоје два основна алгоритма за обилазак графа: **претрага у дубину** и **претрага у ширину**.

5.1.1 Претрага у дубину

Претрага у дубину (DFS, скраћеница од depth–first–search) 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.

Неусмерени графови

Претпоставимо да граф $G = (V, E)$ одговара уметничкој галерији, која се састоји од низа ходника са сликама на зидовима. Гране графа G одговарају ходницима, а чворови одговарају раскрсницама – 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. Притом се дозвољава 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 водећи рачуна 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 ходник којим смо дошли – 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 који воде из раскрснице. Овај приступ зове се **претрага у дубину** (DFS), што је у вези са правилом да се посећују увек нови ходници (идући све дубље у галерију). Основни разлог корисности претраге у дубину лежи у њеној једноставности и лакој реализацији рекурзивним алгоритмом.

После описа претраге у дубину на примеру обиласка галерије, размотримо проблем претраге у дубину када је листом повезаности задат граф и чвор r графа из кога се започиње претрага. Чвор r се **означава** као посећен. Затим се у листи суседа чвора r проналази први неозначени сусед r_1 чвора r, па се из чвора r_1 рекурзивно покреће претрага у дубину. Из неког нивоа рекурзије, покренутог из чвора v, излази се ако су сви суседи (ако их има) чвора v из кога је претрага покренута 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 се у листи суседа чвора r проналази следећи неозначени сусед r_2, извршава се претрага полазећи од r_2, итд.

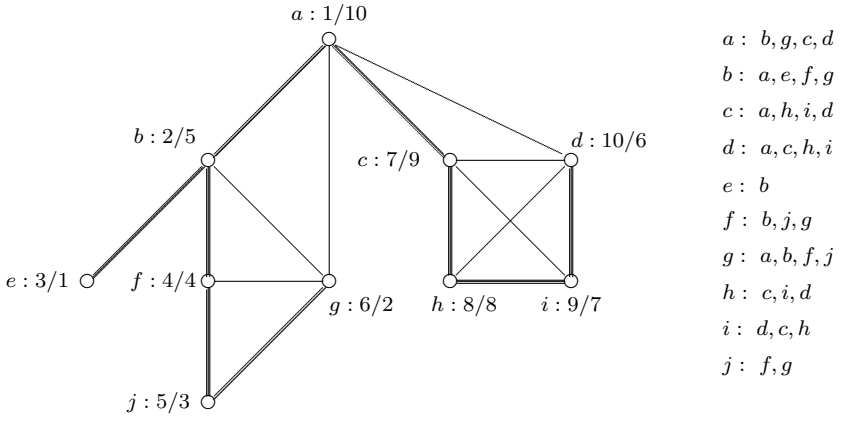
Претрага графа увек се врши са неким *циљем*. Да би се различите примене уклопиле у претрагу у дубину, *посети чвора или гране* придружују се две врсте обраде, **улазна обрада** и **излазна обрада**. Улазна обрада врши се у тренутку означавања чвора. Излазна обрада врши се после повратка неком граном (после завршетка рекурзивног позива), или кад се открије да нека грана води већ означеном чвору. Улазна и излазна обрада зависе од конкретне примене DFS. На тај начин могуће је решавање различитих проблема једноставним дефинисањем улазне и излазне обраде. Алгоритам претраге у дубину дат је у наставку. Полазни чвор за рекурзивну претрагу графа $G = (V, E)$ је v. Због једноставности се за сада претпоставља да је граф G повезан.

$DFS(G, v)$
улаз: $G = (V, E)$ - неусмерени повезан граф и v - чвор графа G
излаз: зависи од примене
1 означи v
2 изврши улазну обраду за v
3 {улазна обрада зависи од примене DFS}
4 **for all** гране (v, w) **do**
5 {редослед суседа чвора v у петљи одређен је листом повезаности}
6 **if** w је неозначен **then**
7 $DFS(G, w)$
8 изврши излазну обраду за грану (v, w)
9 {излазна обрада зависи од примене DFS}
10 {она се понекад врши само за гране ка новоозначеним чворовима}

Пример претраге графа у дубину приказан је на слици 5.1.

Лема: Ако је граф G повезан, онда су по завршетку претраге у дубину сви чворови означени, а све гране графа G су прегледане бар по једном.

Доказ: Претпоставимо супротно, и означимо са U скуп неозначених чворова заосталих после извршавања алгоритма. Пошто је G повезан, бар један чвор u из U мора бити повезан граном са неким означеним чвором w (скуп означених чворова је непразан јер садржи бар чвор v). Међутим, овако нешто је немогуће, јер кад се посети чвор w, морају бити посећени (па дакле и означени) сви његови неозначени суседи, дакле и чвор u. 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.



Слика 5.1: Пример претраге у дубину. Два броја уз чвор једнака су његовим редним бројевима при долазној, односно одлазној DFS нумерацији.

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. Алгоритам се може приказати следећим кодом.

Komponente_povezanosti(G)
улаз: $G = (V, E)$ - неусмерени граф
излаз: $v.Komp$ за сваки чвор добија вредност једнаку редном броју
компоненте повезаности која садржи v
1 $Rb\_komp \leftarrow 1$
2 **while** постоји неозначен чвор v **do**
3 $DFS(G, v)$
4 {са следећом улазном обрадом: $v.Komp \leftarrow Rb\_komp$}
5 $Rb\_komp \leftarrow Rb\_komp + 1$

Ми ћемо најчешће разматрати само повезане графове, јер се у општем случају проблем своди на посебну обраду сваке компоненте повезаности.

Сложеност: Лако је уверити се да се свака грана прегледа тачно два пута, по једном са сваког краја. Према томе, укупан број извршавања тела **for** петље у свим рекурзивним позивима алгоритма DFS је $O|E|)$. С друге стране, број рекурзивних позива је $|V|$, па се сложеност алгоритма може описати изразом $O(|V| + |E|)$.

Конструкција DFS стабла

Приказаћемо сада две једноставне примене алгоритма DFS — нумерацију чворова графа DFS **бројевима**, и формирање специјалног повезујућег стабла, такозваног DFS **стабла**. Приликом обиласка графа G могу се у петљи

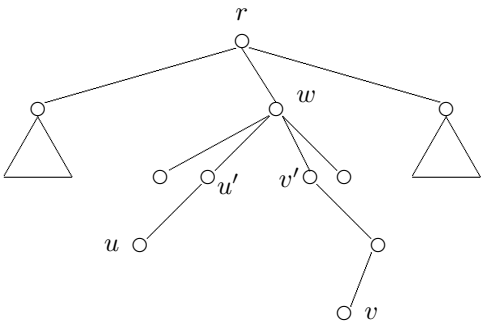
којом се пролазе суседи чвора v издвојити све гране ка новоозначеним чворовима w. Преко издвојених грана достижни су сви чворови повезаног неусмереног графа, па је подграф кога чине издвојене гране повезан. Тај подграф нема циклусе, јер се од свих грана које воде у неки чвор, може издвојити само једна. Према томе, издвојене гране су гране подграфа графа G који је *стабло* — DFS стабло графа G. Полазни чвор је корен 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 описали алгоритми за нумерацију чворова, довољно је задати улазну, односно излазну обраду. Постоје две варијанте DFS нумерације: чворови се могу нумерисати према редоследу означавања (**долазна** DFS **нумерација**, односно **preOrder** нумерација), или према редоследу напуштања (**одлазна** DFS **нумерација**, односно **postOrder** нумерација). Пример графа са чворовима нумерисаним на два начина приказан је на слици 5.1. Алгоритми нумерације (израчунавања редних бројева $v.Pre$ и $v.Post$ за сваки чвор v графа), односно изградње DFS стабла, описани су следећим кодом.

$DFS\_numeracija(G, v)$
улаз: $G = (V, E)$ - неусмерени граф и v - чвор графа G
излаз: за сваки чвор v рачунају се његови редни бројеви $v.Pre$ и $v.Post$
при долазној, односно одлазној нумерацији
Иницијализација: $DFS\_pre \leftarrow 1; DFS\_post \leftarrow 1;$
Покренути DFS са следећом улазном и излазном обрадом:
улазна обрада:
$v.Pre \leftarrow DFS\_pre$
$DFS\_pre \leftarrow DFS\_pre + 1$
излазна обрада:
if w је последњи на листи суседа v **then**
$v.Post \leftarrow DFS\_post$
$DFS\_post \leftarrow DFS\_post + 1$

$DFS\_stablo(G, v)$
улаз: $G = (V, E)$ - неусмерени граф и v - чвор графа G
излаз: T - DFS стабло графа G; пре првог позива T је празан скуп
Покренути DFS са следећом излазном обрадом:
излазна обрада:
if w је неозначен **then**
додати грану (v, w) у T
{горњу наредбу треба додати у **if** наредбу алгоритма DFS}

Чвор v зове се **предак** чвора w у стаблу T са кореном r ако је v на јединственом путу од r до w у T. Ако је v предак w, онда је чвор w **потомак** чвора v. Претрага из чвора v почиње пре претраге из чвора w, па је $v.Pre < w.Pre$. С друге стране, претрага из чвора v завршава се после претраге из чвора w, па је $v.Post > w.Post$. DFS стабло обухвата све чворове повезаног графа G. Редослед синова сваког чвора у стаблу одређен

је листом повезаности која задаје граф G, па се за свака два сина може рећи који је од њих леви (први по том редоследу), а који десни. Релација леви–десни се преноси на произвољна два чвора u и v који нису у релацији предак-потомак, видети илустрацију на слици 5.2. За чворове u и v 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. Ако је чвор u лево од чвора v, онда је он приликом претраге означен 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). С друге стране, претрага из чвора u завршава се пре претраге из чвора v, па је $u.Post < v.Post$.



Слика 5.2: Илустрација релације "лево–десно" на скупу чворова DFS стабла.

Лема: [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.

Доказ: 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 започиње DFS, па u постаје син v у стаблу T.

Тврђење леме може се преформулисати на следећи начин: гране графа не могу бити **попречне гране** у односу на DFS стабло, односно гране које повезују чворове на раздвојеним путевима од корена (тј. таква два чвора u и v која нису у релацији предак-потомак).

Пошто је DFS врло важан алгоритам, дајемо и његову нерекурзивну варијанту. 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 променљиве придружене сваком позиву рекурзивне процедуре. Због тога, кад се заврши са једним рекурзивним позивом процедуре, могућ је повратак на тачно одређено место у позивајућој процедури. Понекад није неопходно све локалне

променљиве чувати на стеку, па одговарајућа нерекурзивна процедура може да буде ефикаснија.

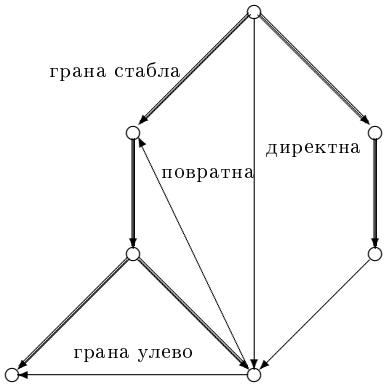
```
Nerekurzivni_DFS(G, v)
улаз: G = (V, E) - повезани граф, v - чвор графа G
излаз: зависи од примене
1  while постоји неозначени чвор w do
2    if v неозначен then w ← v {претрага почиње од v}
3    означи w
4    изврши улазну обраду за w
5    Grana ← w.Prvi {показивач на прву грану из w}
6    упиши на стек w и Grana
7    Otac ← w
8    {почетак основне петље рекурзије}
9    while стек није празан do
10     скини Grana са врха стека
11     while Grana ≠ NULL do {док има непрегледаних грана из чвора}
12       Sin ← Grana.Cvor
13       if Sin није означен then {корак у дубину}
14         означи Sin
15         изврши улазну обраду на Sin
16         упиши Grana.Naredni на врх стека { следећа, кад се обради Sin}
17         {за наставак по завршетку обраде Sin}
18         Grana ← Sin.Prvi
19         Otac ← Sin
20         {упиши Otac на врх стека} {новоозначени чвор}
21       else {Grana је повратна грана}
22         изврши излазну обраду за (Otac, Sin)
23         {ово се прескаче ако се излазна обрада ради само за гране стабла}
24         Grana ← Grana.Naredni
25       {на врху стека су последњи означени чвор и следећа грана из њега}
26     скини Sin са врха стека {корак назад}
27     if стек није празан then {стек постаје празан кад је Sin корен}
28       нека су Grana и Otac на врху стека {не уклањати их са стека}
29       изврши излазну обраду за (Otac, Sin)
```

Основна потешкоћа при превођењу рекурзивне у нерекурзивну верзију потиче од потребе експлицитног памћења промењљивих, 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 елемент листе је слог који садржи две промењљиве: $Cvor$ и $Naredni$; $Cvor$ је име чвора на другом крају гране, а $Naredni$ је показивач на наредни елемент листе. Последњи елемент листе $Naredni$ има вредност **NULL**. DFS се извршава као и у рекурзивној верзији, 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.

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 5.3. У односу на DFS стабло гране графа припадају једној од четири категорије: **гране стабла**, **повратне**, **директне** и **попречне** гране. 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 који нису "сродници" у стаблу. Попречне гране, међутим, морају бити усмерене "здесна улево", као што показује следећа лема.



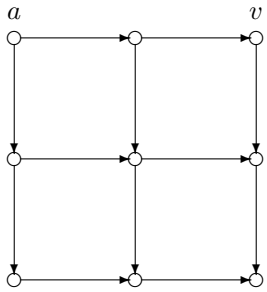
Слика 5.3: DFS стабло усмереног графа.

Лема: [Основна особина DFS стабла усмереног графа] 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.

Доказ: Пошто према долазној DFS нумерацији v претходи 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 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 је w потомак v у стаблу T.

Алгоритам DFS за повезан *неусмерени* граф, започет *из произвољног чвора*, обилази цео граф. Аналогно тврђење не мора бити тачно за усмерене графове. Посматрајмо усмерени граф на слици 5.4. 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 у графу не постоји чвор из кога DFS обилази цео граф. Према томе, увек

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



Слика 5.4: Пример кад DFS усмереног графа (ако се покрене из чвора v) не обилази све чворове графа.

Непостојање грана графа које иду слева удесно говори нешто корисно о *одлазној нумерацији* чворова графа и о четири врсте грана у односу на DFS стабло. На слици 5.5(а) приказана су три чвора графа u, v и w у оквиру DFS стабла графа. Чворови v и w су синови чвора u, а чвор w је десно од чвора v. На слици 5.5(б) приказани су временски интервали трајања рекурзивних позива DFS за сваки од ових чворова. Запажамо да је DFS из чвора v, потомка чвора u, активан само у подинтервалу времена за које је активан DFS из чвора u (претка чвора v). Специјално, DFS из v завршава се пре завршетка DFS из u. Према томе, из чињенице да је v потомак u следи да је $v.Post < u.Post$. Поред тога, ако је w десно од v, онда позив DFS из w не може бити покренут 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 5.5, 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.

Размотримо сада за произвољну грану (u, v) графа 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 $v.Post < u.Post$.

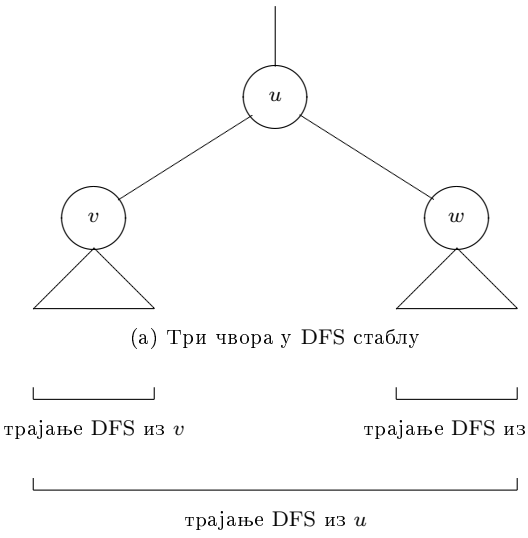
2. Ако је (u, v) попречна грана, онда је због тога што је v лево од u, поново $v.Post < u.Post$.

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: 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.



(а) Три чвора у DFS стаблу

(б) Активни интервали за њихове позиве DFS

Слика 5.5: Однос између положаја чворова у DFS стаблу и трајања рекурзивних позива покренутих из ових чворова.

Проблем: За задати усмерени граф $G = (V, E)$ установити да ли садржи усмерени циклус.

Лема: 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.

Доказ: Ако је грана (u, v) повратна, онда она заједно са гранама стабла на путу од v до u чини циклус. Супротно тврђење је такође тачно: ако у графу постоји циклус, тада је једна од његових грана повратна. Заиста, претпоставимо да у графу постоји циклус који чине гране (v_1, v_2), (v_2, v_3),. . . , (v_k, v_1), од којих ни једна није повратна у односу на T. 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),. . . (v_{k-1}, v_k) није повратна. Према претходној леми важе неједнакости $v_1.Post > v_2.Post > \cdots > v_k.Post$, из којих следи да је $v_k.Post < v_1.Post$, па је грана (v_k, v_1) повратна — супротно претпоставци. Тиме је доказано да у сваком циклусу постоји повратна грана у односу на DFS стабло.

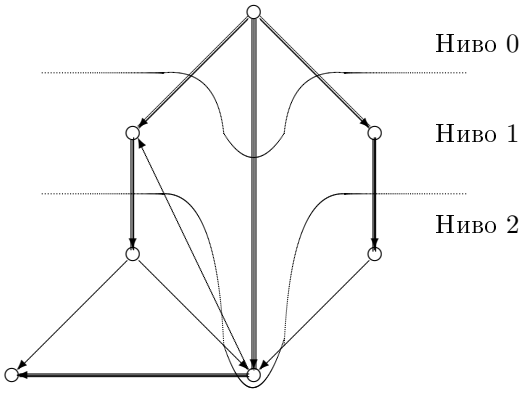
Алгоритам за проверу да ли граф садржи циклус своди се на DFS нумерацију и проверу постојања повратне гране на основу леме. $v.Na\_putu$ има вредност *true* за чвор v ако и само ако је чвор на путу кроз стабло од полазног до текућег чвора. Претпоставља се да претрага из полазног чвора обилази цео граф, односно да DFS стабло постоји.

Aciklicki(G, v)
улаз: $G = (V, E)$ - усмерени граф, v - полазни чвор, корен DFS стабла
излаз: *Postoji_ciklus* - тачно ако G садржи циклус, а нетачно у противном
Иницијализација: $w.Na\_putu \leftarrow false$ за све чворове w различите од v, $v.Na\_putu \leftarrow true$ и $Postoji\_ciklus \leftarrow false$

Покренути DFS из произвољног чвора, са следећом улазном и излазном обрадом:
улазна обрада:
  if из v излази бар једна грана **then**
    $v.Na\_putu \leftarrow true$
    $\{x.Na\_putu$ је тачно ако је x на путу од корена до текућег чвора$\}$
излазна обрада:
  if $w.Na\_putu$ **then**
    $Postoji\_ciklus \leftarrow true$
    $halt$
  if w последњи чвор у списку суседа v **then**
    $v.Na\_putu \leftarrow false$

5.1.2 Претрага у ширину

Претрага у ширину (или BFS, што је скраћеница од breadth–first–search) је обилазак графа на систематичан начин, ниво по ниво, при чему се успут формира стабло претраге у ширину (BFS стабло). Ако полазимо од чвора v (v је корен BFS стабла), онда се најпре посећују сви суседи чвора v редоследом одређеним редоследом у листи повезаности графа (синови чвора v у стаблу претраге, ниво један). Затим се долази до свих "унука" (ниво два), и тако даље (видети пример на слици 5.6). Обилазак се реализује слично као у *нерекурзивној* верзији DFS, сем што је стек замењен редом (листом FIFO, што је 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. Запажа се да излазна обрада код BFS, за разлику од DFS, нема смисла; претрага нема повратак "навише", већ се, полазећи од корена, креће само наниже.



Слика 5.6: BFS стабло усмереног графа.

$BFS(G, v)$
улаз: $G = (V, E)$ - неусмерени повезан граф и v - чвор графа G
излаз: зависи од примене

```
1 означи v
2 упиши v у ред Q {FIFO листа}
3 while Q је непразан do
4   скини први чвор w из реда Q
5   изврши улазну обраду за w
6   {улазна обрада зависи од примене BFS}
7   for all гране (w,x) за које x није означен do
8     означи x
9     додај (w,x) у стабло T
10    упиши x у ред Q
```

Сложеност: Лако је уверити се да се сваки чвор обрађује по једном и да се свака грана прегледа по једном. Стога је временска сложеност алгоритма BFS $O(|V| + |E|)$.

Лема: Ако грана (u, w) припада BFS стаблу и чвор u је отац чвора w, онда чвор u има најмањи BFS број међу чворовима из којих постоји грана ка w.

Доказ: Ако би у графу постојала грана (v, w), таква да v има мањи BFS број од u, онда би у тренутку обраде чвора v чвор w морао бити уписан у ред, па би грана (v, w) морала бити укључена у BFS стабло, супротно претпоставци.

Дефинишимо **растојање** $d(u, v)$ између чворова u и v као дужину најкраћег пута од u до v; под дужином пута подразумева се број грана које чине тај пут.

Лема: Пут од корена r BFS стабла до произвољног чвора w кроз BFS стабло најкраћи је пут од r до w у графу G.

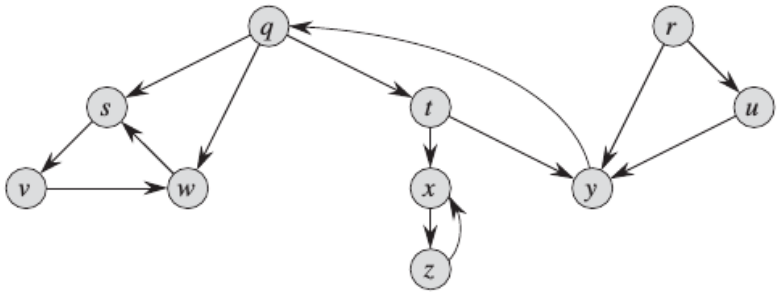
Доказ: Индукцијом по d доказаћемо да до сваког чвора w на растојању d од корена r (јединствени) пут кроз стабло од r до w има дужину d. За $d = 1$ тврђење је тачно: грана (r, w) је обавезно део стабла, па између r и w постоји пут кроз стабло дужине 1. Претпоставимо да је тврђење тачно за све чворове који су на растојању мањем од d од корена, и нека је w неки чвор на растојању d од корена; другим речима, постоји низ чворова $w_0 = r$, w_1, $w_2, \ldots, w_{d-1}, w_d = w$ који чине пут дужине d од r до w, и не постоји краћи пут од r до w. Пошто је дужина најкраћег пута од r до w_{d-1} једнака $d - 1$ према индуктивној хипотези пут од r до w_{d-1} кроз стабло има дужину $d-1$. У тренутку обраде чвора w_{d-1}, ако чвор w није означен, 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. У противном, ако је у том тренутку w_d већ означен, онда до w кроз стабло води грана из неког чвора w'_{d-1}, означеног пре w_{d-1}, из чега следи да је ниво чвора w'_{d-1} највише $d-1$, ниво чвора w највише d, односно до w води пут кроз стабло дужине d.

Лема: Ако је $(v, w) \in E$ грана неусмереног графа $G = (V, E)$, онда та грана спаја два чвора чији се нивои разликују највише за један.

Доказ: Нека је нпр. чвор v први достигнут претрагом и нека је његов ниво d. Тада је ниво чвора w већи или једнак од d. С друге стране, ниво чвора w није већи од $d + 1$, јер до њега води грана стабла или из чвора v, или из неког чвора који је означен пре v. Дакле, ниво чвора w је или d или $d + 1$.

5.2 Задаци за вежбу

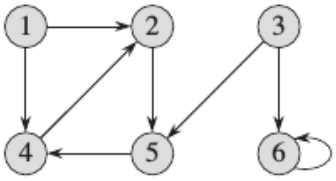
1. Показати како обилазак у дубину покренут из чвора q ради за усмерени граф приказан на слици. Претпоставити да су чворови у листама суседа уређени лексикографски према ознакама чворова. Одредити одлазну и долазну нумерацију чворова и класификовати све гране графа.



2. Дати контрапример за тврђење да ако усмерени граф G садржи пут од чвора u до чвора v и ако за DFS бројеве ових чворова важи да је $u.Pre < v.Pre$, онда је v потомак чвора u у одговарајућем DFS стаблу.

3. Дати контрапример за тврђење да ако усмерени граф G садржи пут од чвора u до чвора v онда у свакој DFS претрази важи да је $v.Pre \leq u.Post$.

4. Одредити BFS стабло и BFS нумерацију чворова у графу са слике ако је BFS покренут из чвора 3. Чворови су у листи повезаности уређени у растући низ према својим ознакама.



5. Колико је време извршавања алгоритма BFS ако је улазни граф задат матрицом повезаности?

6. Дати пример усмереног графа $G = (V, E)$, истакнутог чвора $s \in V$ и скупа грана стабла $E_T \subset E$ тако да важи да за сваки чвор $v \in V$, јединствени прости пут у графу (V, E_T) из s до v је најкраћи пут у G, али се скуп грана E_T не може добити извршавањем алгоритма BFS на графу G, без обзира на то како су чворови уређени у листи повезаности.

7. Постоје две групе навијача: навијачи Партизана и навијачи Звезде. Између сваког пара навијача може или не мора постојати ривалство.

Претпоставимо да имамо n навијача и листу од r парова навијача између којих постоји ривалство. Конструисати алгоритам сложености $O(n+r)$ којим се одређује да ли је могуће означити неке од навијача као навијаче Партизана, а неке од њих као навијаче Звезде тако да свако ривалство буде између навијача различитих тимова.

8. Показати да у графу за који важи да је растојање између свака два чвора највише d, свако BFS стабло има дубину највише d, али DFS стабло може бити веће дубине.

9. Да ли је тачно да DFS обилазак усмереног графа увек производи једнак број грана стабла (независно од редоследа у коме су дати чворови и независно од редоследа чворова у листама повезаности)?

10. Да ли важи да ако из усмереног графа G избацимо све повратне гране у односу на DFS стабло, граф постаје ациклички?

11. Да ли код усмереног графа важи да ако не постоји повратна грана у односу на BFS стабло онда граф не садржи циклус?

12. Описати како се могу израчунати улазни и излазни степен свих чворова у графу ако је граф представљен (а) листом повезаности (б) матрицом повезаности.

13. Дат је неусмерен повезан граф G. Конструисати алгоритам временске сложености $O(|V| + |E|)$ за одређивање пута који пролази кроз све гране графа G по два пута – по једном у сваком смеру.

14. Доказати или оповргнути следеће тврђење: ако је $G = (V, E)$ повезан неусмерен граф, онда је висина DFS стабла графа G увек већа или једнака од висине BFS стабла графа G (размотрити случај када оба алоритма крећу из истог чвора и када то није случај).

15. Доказати да су за неусмерен повезан граф $G = (V, E)$, DFS стабло и BFS стабло исти ако и само ако је граф G стабло.

Глава 6

Алгоритамске стратегије

6.1 Алгоритми грубе силе

Алгоритми грубе силе (енгл. brute force, односно exhaustive search) се обично примењују на оптимизационе проблеме. Рецимо у случају када је потребно одредити максималну вредност неке величине. Поступак за решавање ових проблема укључује прегледање свих варијанти. Алгоритми грубе силе су корисни за врло мале улазе; за велике улазе мора се наћи ефикасније решење.

Пример 1: За дати низ $x_1, x_2, \ldots, x_n$ пронаћи индекс i тако да је x_i максимални елемент низа. Проблем се може решити проласком кроз цео низ; сложеност алгоритма је $O(n)$.

Пример 2: Од датих n тачака $P_i(x_i, y_i)$ треба пронаћи две најближе тачке, тј. треба одредити индексе i и j тако да важи:

$$d(P_i, P_j)_{i \neq j} \leq d(P_r, P_s)_{r,s=1,2,\ldots,n, r \neq s}$$

Укупно има $\binom{n}{2} = \frac{n(n-1)}{2}$ различитих парова тачака и алгоритам грубе силе би прегледао све парове, па је сложеност алгоритма $O(n^2)$. Аналогно би се решавао и проблем налажења две најудаљеније тачке.

Пример 3: **Проблем трговачког путника** (TSP од енгл. Traveling Salesman Problem). У поставци овог проблема дато је n градова и потребно је све их обићи, тако да ни у који град не дођемо два пута, а да дужина укупног пута буде што мања. Формално гледано, дата је квадратна матрица реда n растојања између градова $D = [d_{ij}]$ и потребно је одредити пермутацију индекса $i_1, i_2, \ldots, i_n$ тако да сума $d_{i_1,i_2} + d_{i_2,i_3} + \ldots + d_{i_{n-1},i_n} + d_{i_n i_1}$ има минималну вредност. Да би се одредило решење потребно је размотрити $(n-1)!$ маршрута (јер град из кога започињемо обилазак можемо фиксирати, с обзиром на то да он мора припадати сваком обиласку). Ово решење није јако неефикасно за улазе величине до 10. За веће улазе требало би наћи неки ефикаснији алгоритам.

У алгоритму грубе силе за решавање TSP проблема потребно је да за сваку пермутацију умемо да одредимо следећу, памтимо тренутно најкраћи пут и на крају вратимо као резултат пермутацију којој одговара најкраћи пут.

TSP_brute_force(D)
улаз: D - матрица растојања градова
излаз: p_{max} - пермутација која даје најкраћи пут
1 $p \leftarrow \{1, 2, \ldots, n-1\}$ { разматрамо пермутације првих $n-1$ градова,
после чега се увек иде у n-ти град, па назад у први}
2 $p_{max} \leftarrow p$
3 $s \leftarrow d_{1,2} + d_{2,3} + \ldots + d_{n,1}$
4 **while** *sledeca_permutacija*$(p, n-1)$ **do**
5 {претпоставка је да ако постоји наредна пермутација,
она је позивом претходне функције смештена опет у p}
6 **if** $d_{p_1,p_2} + d_{p_2,p_3} + \ldots + d_{p_{n-2},p_{n-1}} + d_{p_{n-1},n} + d_{n,p_1} < s$ **then**
7 $p_{max} \leftarrow p$
8 $s \leftarrow d_{p_1,p_2} + d_{p_2,p_3} + \ldots + d_{p_{n-2},p_{n-1}} + d_{p_{n-1},n} + d_{n,p_1}$
9 **return** p_{max}

Како пронаћи наредну пермутацију? На овај проблем вратићемо се у глави 7.4. Ако рецимо уочимо пермутацију $(2, 6, 8, 4, 7, 5, 3, 1)$ бројева $1, 2, \ldots, 8$, наредна пермутација, гледано лексикографски, требало би да буде пермутација $(2, 6, 8, 5, 1, 3, 4, 7)$. Њу добијамо полазећи од максималног опадајућег суфикса низа којим је представљена задата пермутација. Дакле, идемо од краја низа и тражимо два суседна елемента за које важи да је први од њих мањи од другог – означимо их са $p[i]$ и $p[i+1]$ (у горњем случају то су елементи 4 и 7). Када их нађемо (ако таква два елемента не постоје, онда је задата пермутација опадајући низ, па не постоји наредна пермутација), онда почев од последњег елемента низа тражимо најмањи елемент који је већи од елемента $p[i]$ – нека је то $p[j]$ (важи $j > i$). У примеру то је број 5. Након тога елементима $p[i]$ и $p[j]$ мењамо места, а затим свим обрћемо све елементе подниза низа p који се састоји од елемената почев од позиције $i+1$ све до краја низа. На овај начин од опадајућег подниза добијамо растући (у примеру од подниза $(7, 4, 3, 1)$ добијамо подниз $(1, 3, 4, 7)$). Овај алгоритам описан је следећим кодом.

sledeca_permutacija(p, n)
улаз: p - текућа пермутација, n - дужина низа p
излаз: p - садржи наредну пермутацију, лексикографски гледано
елемент p_n је фиксиран, тј. не мења се
1 $i \leftarrow n-1$
2 **while** $i > 0$ **and** $p[i] > p[i+1]$ **do**
3 $i \leftarrow i-1$
4 **if** $i = 0$ **then**
5 *print* "грешка: не постоји следећа пермутација"
6 **return**
7 $j \leftarrow n$
8 **while** $p[i] > p[j]$ **do**
9 $j \leftarrow j-1$
10 *zameni*$(p[i], p[j])$
11 обрни елементе низа p од позиције $i+1$ до позиције $n-1$

Сложеност алгоритма *TCP_brute_force* је $O(n!)$ јер је број разматраних пермутација $(n-1)!$, а обрада пермутације траје $O(n)$.

Пример 4: Нека је граф G задат листом повезаности. Потребно је одредити најдужу грану у графу. То можемо урадити тако што прегледамо све гране графа и тражимо најдужу.

6.2 Похлепни алгоритми

Оптимизациони проблеми се обично састоје из низа корака са скупом избора у сваком кораку. Често је ово превише временски захтевно па се траже ефикаснији алгоритми. **Похлепни алгоритми** увек праве избор који делује најбољи у том тренутку. Дакле, прави се *локално* оптимални избор у нади да ће овај избор водити и *глобално* најбољем решењу. Похлепни алгоритми за неке (не све) проблеме проналазе оптимално решење.

Пример 1: Пењемо се на планину и циљ нам је да дођемо на највиши врх. У сваком моменту на располагању су нам стрмији и мање стрми путеви. Ако увек идемо најстрмијом стазом попећемо се на неки врх, не обавезно највиши.

Пример 2: Имамо на располагању новчанице од 1,2,5,10,20,50,100,200,500 динара и хоћемо да вратимо кусур. Похлепни алгоритам би увек враћао највећу новчаницу која је мања од текуће вредности кусура (нпр. $188 = 100+50+20+10+5+2+1$). Овакав приступ у општем случају не гарантује да ћемо вратити кусур који се састоји од најмањег броја новчаница. Рецимо ако би нам на располагању биле новчанице од 5,4,2 и 1 динара, а треба да вратимо кусур од 8 динара, похлепни алгоритам би вратио комбинацију $5+2+1$, а решење које укључује најмањи број новчаница је $4+4$.

Пример 3: Претпоставимо да је дат скуп $S = \{a_1, a_2, \ldots, a_n\}$ од n предложених **активности** које претендују на коришћење истог ресурса, на пример салу за предавања, која може да услужи само једну активност у једном тренутку. Свака активност a_i има своје **време почетка** s_i и **време завршетка** f_i, при чему важи: $0 \leq s_i \leq f_i$. Ако је одабрана, активност a_i се одржава током полузатвореног интервала $[s_i, f_i)$. Активности a_i и a_j су **компатибилне** ако се интервали $[s_i, f_i)$ и $[s_j, f_j)$ не преклапају, тј. ако је $s_i \geq f_j$ или $s_j \geq f_i$. У **проблему избора активности** потребно је одредити подскуп максималне величине међусобно компатибилних активности.

Нека су активности поређане у монотоно неопадајућем редоследу времена завршетка:

$$f_1 \leq f_2 \leq \ldots f_n$$

На пример, нека је задат следећи скуп активности:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

У горе наведеном примеру, подскуп $\{a_3, a_9, a_{11}\}$ се састоји од међусобно компатибилних активности. Међутим, ово није максимални подскуп јер исто важи и за подскуп $\{a_1, a_4, a_8, a_{11}\}$ који је већи од њега и јесте максималан такав скуп.

Коју би активност требало изабрати да буде прва? Интуиција нам сугерише да би требало изабрати активност која оставља ресурс слободним за што већи број активности. Исправна реализација ове идеје је избор активности са најранијим временом завршетка, односно да треба изабрати активност a_1, јер се она прва завршава. После уклањања са списка оних активности које нису компатибилне са изабраном, даље се може наставити на исти начин са избором друге активности итд.

Остаје само још једно питање: да ли је наша интуиција коректна, тј. да ли је похлепни избор увек део неког оптималног решења? Показује се да ово важи.

Теорема: Посматрамо непразни проблем S_k и нека је a_m активност у S_k са најранијим временом завршетка. Нека се оптимално решење овог проблема састоји од r активности. Тада постоји подскуп од r активности који садржи активност a_m.

Доказ: Нека је A_k подскуп међусобно компатибилних активности из S_k максималне величине r и нека је a_j активност из A_k са најранијим временом завршетка. Ако је $a_j = a_m$ завршили смо доказ јер важи да је a_m у неком подскупу међусобно компатибилних активности из S_k максималне величине. Ако је $a_j \neq a_m$ нека је $A'_k = A_k \setminus \{a_j\} \cup \{a_m\}$ скуп A_k у коме смо a_m заменили са a_j. Активности у A'_k су дисјунктне, што следи из тога што су активности у A_k дисјунктне, a_j је прва активност у A_k која завршава и $f_m \leq f_j$. С обзиром на то да је $|A'_k| = |A_k|$ закључујемо да је A'_k скуп међусобно компатибилних активности из S_k максималне величине и он укључује a_m.

Izbor_aktivnosti_greedy(s, f, n)
улаз: s - низ почетних времена активности, f - низ одговарајућих завршних времена активности, n - дужина ових низова; низови су сортирани неопадајуће према временима завршетка
излаз: A - скуп максималне величине међусобно компатибилних активности

```
1 A ← {a_1}
2 poslednja ← 1 {индекс последње изабране активности}
3 for i ← 2 to n do
4    if s[i] ≥ f[poslednja] then
5       A ← A ∪ {i}
6       poslednja ← i
7 return A
```

Сложеност овог алгоритма је $O(n \log n)$, а у случају да су активности већ сортиране према завршним временима $O(n)$.

Проблем 4: У магацину који ће бити поплављен кроз највише један сат налази се n врста робе (у расутом облику). Власник има могућност да спасе само толико робе колико може да стане у ранац, носивости W. Нека i-ти артикал из магацина, $i = 1, 2, \ldots, n$ има вредност v_i и тежину w_i. Власник жели да покупи највреднију робу, притом је могуће да не покупи сву количину неког артикла (као да посматрамо робу на мерење). Како то да постигне?

Овај проблем називамо **разломљени проблем ранца** (енгл. fractional knapsack problem) и он је пример проблема који се успешно решава похлепним алгоритмом.

Потребно је најпре да за сваки артикал израчунамо вредност по јединици тежине, као однос: $\frac{v_i}{w_i}$. Похлепна стратегија подразумева да се узме што узима што више може од артикла који има највећу вредност по јединици масе. Ако узме сву количину тог артикла и још увек има места у ранцу, он узима што више може од артикла који има наредну највећу вредност по јединици масе итд.

Razlomljeni_ranac(W, v, w, n)
улаз: W - носивост ранца, v - вектор вредности артикала,
w - вектор тежина артикала, n - број артикала
претпоставља се да су артикли сортирани нерастуће
према вредности по јединици мере
излаз: p - проценат сваког предмета који је стављен у ранац

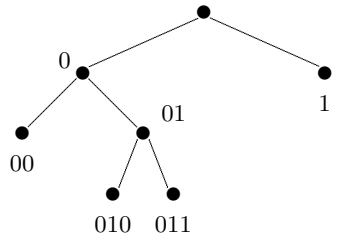
```
1   S ← 0
2   i ← 1
3   for j ← 1 to n do
4     p[j] ← 0
5   while S ≤ W do
6     S ← S + w[i]
7     p[i] ← 1
8     i ← i + 1
9   if S > W then
10    i ← i − 1
11    {ако последњи артикал не може цео да стане, узимамо само део}
12    p[i] ← 1 − (S − W)/w[i]
```

Дакле, сортирањем артикала у нерастући низ према вредности по јединици масе тако да важи: $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$, похлепни алгоритам се извршава у времену $O(n \log n)$.

Проблем 5: Нека је задат текст који је потребно записати са што мање битова. Притом је сваки знак текста представљен јединственим низом битова — **кодом** тог знака. Ако су дужине кодова свих знакова једнаке (што је управо случај са стандардним кодовима, као што је ASCII), број битова 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 битова.

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 битова у коду знака s_i, у коду означеном са E, онда је укупан број битова 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. Основни услов који код E мора да задовољи је да омогућује једнозначно декодирање, односно реконструкцију полазног фајла на основу његове кодиране верзије. Један од начина да се једнозначност 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) кôд може се представити бинарним стаблом, видети пример на слици 6.1. 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, кôд задат кодним стаблом, у коме су кодне речи листови, префиксни је кôд. 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 кôд префиксни. На пример, низ битова 011100010001, може се у коду представљеним стаблом на слици 6.1 декодирати само на један начин: $011 - 1 - 00 - 010 - 00 - 1$. Проблем ефикасног кодирања може се сада формулисати на следећи начин: Задат је текст са n различитих знакова, тако да су учестаности знакова задате низом $F = f_1, f_2, \ldots, f_n$. Одредити префиксни кôд E који минимизира број бита $L(E, F)$ употребљених за кодирање.

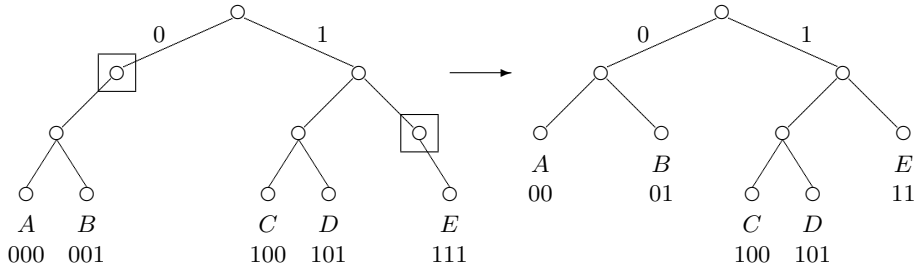


Слика 6.1: Представљање префиксног кода стаблом.

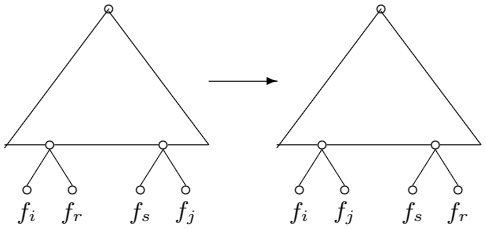
За префиксни кôд који минимизира вредност $L(E, F)$ каже се да је **оптимални кôд**; он очигледно зависи од низа учестаности знакова F. Циљ је за дати низ F одредити оптимални префиксни кôд (било који; у општем случају постоји више различитих оптималних кодова). Размотримо због тога мало детаљније особине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

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 6.2. 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 6.3).



Слика 6.2: Кодно стабло у коме неки чвор има само једног сина није кодно стабло оптималног кода.



Слика 6.3: Трансформација кодног стабла, после које два знака са најмањим фреквенцијама f_i и f_j постају синови истог оца.

Алгоритам кодирања (налажења оптималног кода) заснива се на свођењу проблема са n знакова на проблем са $n-1$ знаком; базни случај је тривијалан. Свођење на мању азбуку изводи се заменом два најређа знака једним новим знаком. 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 z 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 z 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 Хафменов кôд (према имену аутора алгоритма, D. Huffman).

Операције које се извршавају приликом формирања Хафменовог кода су

- уметање у структуру података,

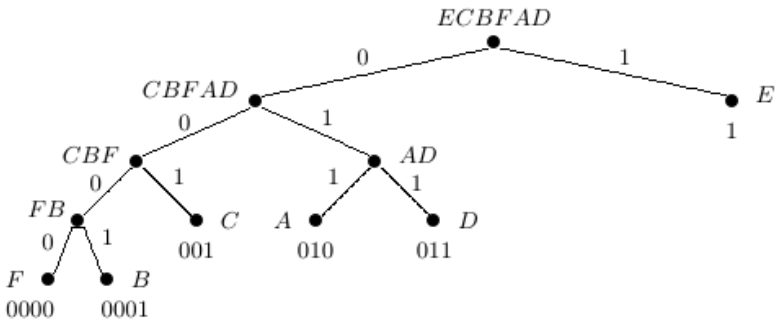
- брисање два знака са најмањим фреквенцијама из структуре, и
- конструкција кодног стабла.

Хип је погодна структура података за прве две операције, јер се тада те операције у најгорем случају извршавају за $O(\log n)$ корака. Сложеност додавања новог чвора стаблу ограничена је константом. Уметања и брисања из хипа извршавају се у $O(\log n)$ корака. Сложеност алгоритма је дакле $O(n \log n)$.

Пример: 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 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 6.4 приказано је добијено кодно стабло.



Слика 6.4: Стабло Хафменовог кода из примера.

6.3 Алгоритми засновани на разлагању

Нека је дат проблем димензије n који треба решити. **Стратегија разлагања** (декомпозиције, енгл. divide-and-conquer) састоји се у томе да се формира неколико потпроблема мањих димензија. Потребно је решити ове потпроблеме, а затим наћи метод за комбиновање решења потпроблема у решење полазног проблема. Уколико су потпроблеми и даље великих

димензија, могуће је поново вршити разлагање. Најчешће су потпроблеми добијени разлагањем истог типа као и полазни проблем. Поновна примена принципа разлагања се природно изражава *рекурзивним алгоритмом*. На овај начин добијају се све мањи и мањи потпроблеми све док се не добију потпроблеми који су довољно мали да се могу решити без разлагања. Сложеност већине алгоритама заснованих на разлагању се може изразити рекурентном једначином облика:

$$T(n) = \begin{cases} T(1), & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$$

при чему се од проблема димензије n формира a потпроблема димензије b пута мање, а укупно време потребно за разлагање проблема на потпроблеме и за обједињавање решења потпроблема у решење проблема је $f(n)$.

У наставку ћемо видети неке примере проблема који се могу ефикасно решити алгоритмима заснованим на разлагању.

6.3.1 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 наставку ћемо размотрити неколико варијација бинарне претраге.

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 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 $x_i = z$. 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 са величином улаза $n/2$, који се решава рекурзивно. Базни случај $n = 1$ решава се непосредним упоређивањем броја z са елементом.

Binarna_pretraga(X, n, z)
улаз: X - низ од n бројева уређених неопадајуће, z - број који се тражи
излаз: индекс i такав да је $X[i] = z$ или 0 ако такав индекс не постоји,
(претпоставка је да индекси низа иду од 1)
1 **return** *Nadji*$(z, X, 1, n)$

Nadji$(z, X, Levi, Desni)$
улаз: X - низ од n бројева уређених неопадајуће који се разматра
у границама од *Levi* до *Desni*, z - број који се тражи
излаз: индекс i такав да је $X[i] = z$ или 0 ако такав индекс не постоји

```
1  if Levi = Desni then
2    if X[Levi] = z then
3      return Levi
4    else return 0
5  else
6    Srednji ← ⌈(Levi + Desni)/2⌉
7    if z < X[Srednji] then
8      return Nadji(z, X, Levi, Srednji − 1)
9    else
10     return Nadji(z, X, Srednji, Desni)
```

Сложеност: После сваког упоређивања опсег могућих индекса се полови, па је потребан број упоређивања за проналажење задатог броја у низу величине n једнак $O(\log n)$. 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, овај програм није тешко 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 $z = 0$.

Бинарна претрага циклички уређеног низа

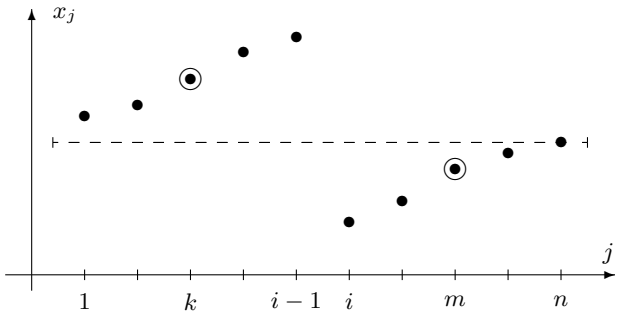
За низ $x_1, x_2, \ldots, x_n$ каже се да је **циклички уређен** ако важе неједнакости

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

где је x_i најмањи елемент 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 елемент 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 елемент 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 6.5. Према томе, једним упоређивањем може се елиминисати много елемената. Одговарајућим избором k и m, i се може одредити помоћу $O(\log n)$ упоређивања. Инваријанта главне петље алгоритма је услов да се i увек налази у (цикличном) интервалу $(Levi, Desni]$.



Слика 6.5: Циклички уређен низ — илустрација.

$Ciklicna\_Binarna\_Pretraga(X, n)$
улаз: X - низ од n различитих бројева уређених циклички
излаз: индекс минималног елемента у низу X
1 **return** $c\_Nadji(X, 1, n)$

$c\_Nadji(X, Levi, Desni)$
улаз: X - низ од n бројева уређених циклички који се разматра
у границама од $Levi$ до $Desni$
излаз: индекс минималног елемента у датом поднизу
1 **if** $Levi = Desni$ **then**
2 **return** $Levi$
3 **else**
4 $Srednji \leftarrow \lceil (Levi + Desni)/2 \rceil$
5 **if** $X[Srednji] < X[Desni]$ **then**
6 **return** $c\_Nadji(X, Levi, Srednji)$
7 **else**
8 **return** $c\_Nadji(X, Srednji + 1, Desni)$

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 $O(\log_2 j)$ 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 се 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 $i < \sqrt{n}$.

Проблем муцавог подниза

Принцип бинарне претраге може се искористити и у проблемима који на први поглед немају везе са бинарном претрагом. Нека су $A = a_1 a_2 \dots a_n$ и $B = b_1 b_2 \dots b_m$ два низа знакова из коначног алфабета и важи $m \leq n$. Каже се да је B подниз низа 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 се назива **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 низа A, онда је за свако i, $1 \leq i \leq j$, B^i подниз низа 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 низа 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 за које тврђење важи. Временска сложеност алгоритма је $O((n+m)\log(n/m)) = O(n\log(n/m))$.

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.

6.3.2 Истовремено тражење најмањег и највећег члана низа

Налажење највећег или најмањег елемента у низу је једноставно. Ако знамо највећи елемент у низу дужине $n-1$, онда треба да га још упоредимо са n-тим елементом да бисмо пронашли највећи елемент 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 $n-1$.

Размотримо сада наредни проблем.

Проблем: Пронаћи највећи и најмањи елемент 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 елемент (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 елемент са највећим и најмањим до тада. То су два упоређивања, што значи да ће број упоређивања поново бити $2n-3$, јер за први елемент није потребно ни једно упоређивање.

Алгоритам заснован на разлагању би радио на следећи начин: ако низ има више од два елемента, делимо га на две половине. Тачније ако радимо са низом $a_1, a_2, \ldots, a_n$ делимо га на два подниза $a_1, \ldots, a_{\lceil n/2 \rceil}$ и $a_{\lceil n/2 \rceil+1}, \ldots, a_n$. Након тога можемо рекурзивно решити ова два потпроблема. Како искомбиновати решења ова два потпроблема? Максимум полазног низа једнак је већој од вредности максимума два разматрана потпроблема, а минимум полазног низа мањој од вредности минимума два разматрана потпроблема. Ако је низ дужине један онда је једини елемент низа његов минимум и максимум, а ако је дужине два онда коришћењем једног поређења можемo решити проблем.

$MaksimumIMinumum(X, n, Max, Min)$
улаз: X - низ од n бројева
излаз: Max - максимални елемент низа, Min - минимални елемент низа
1 $MaxMin(X, 1, n, Max, Min)$

$MaxMin(X, Levi, Desni, Max, Min)$
улаз: X - низ од n бројева који се разматра у границама од $Levi$ до $Desni$
излаз: Max - максимални елемент низа, Min - минимални елемент низа

```
1   if Levi = Desni then
2     Max ← X[Levi]
3     Min ← X[Levi]
4   else if Levi = Desni − 1 then
5     if X[Levi] < X[Desni] then
6       Max ← X[Desni]
7       Min ← X[Levi]
8     else
9       Max ← X[Levi]
10      Min ← X[Desni]
11  else
12    Srednji ← ⌈(Levi + Desni)/2⌉
13    MaxMin(X, Levi, Srednji, Max, Min)
14    MaxMin(X, Srednji + 1, Desni, Max1, Min1)
15    Max ← max{Max, Max1}
16    Min ← min{Min, Min1}
```

Сложеност: Број поређења који се користи у овом приступу дат је рекурентном једначином:

$$T(n) = \begin{cases} 0, & n = 1 \\ 1, & n = 2 \\ 2T(\lfloor n/2 \rfloor) + 2, & n > 2 \end{cases}$$

Ако претпоставимо да је $n = 2^k$ важи:

$$T(2^k) = 2T(2^{k-1}) + 2$$

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

Ако израз $\frac{T(2^k)}{2^k}$ означимо са a_k, при чему важи $a_1 = \frac{T(2)}{2} = \frac{1}{2}$ добијамо једначину:

$$a_k = a_{k-1} + \frac{1}{2^{k-1}}$$

Сумирањем добијамо: $a_k = a_1 + 1 - 2^{1-k} = \frac{3}{2} - 2^{1-k}$. Према томе:

$$T(n) = T(2^k) = 2^k a_k = \frac{3}{2}2^k - 2 = \frac{3}{2}n - 2$$

Може се показати да ако n није степен двојке, тада важи $T(n) \leq \frac{3}{2}n - 2$.

6.3.3 Сортирање обједињавањем и сортирање раздвајањем (квик сорт)

Претходно разматрани алгоритми за сортирање обједињавањем и сортирање раздвајањем су засновани на стратегији решавања проблема разлагањем и нећемо их сада поново разматрати.

6.3.4 Проблем одређивања k-тог најмањег члана низа

Ранговске статистике низа S су k-ти најмањи бројеви r_k, $k = 1, 2, \ldots, n$. Све ранговске статистике могу се добити сортирањем низа S. Специјални случајеви су минимум r_1, максимум r_n и **медијана** $r_{\lceil n/2 \rceil}$.

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 елемент у S.

Овај проблем називамо проблем одређивања **ранговских статистика**, односно проблем **селекције**. Ако је k врло близу 1 или n, онда се k-ти најмањи елемент може одредити извршавањем алгоритма за налажење минимума (максимума) k пута. Овај приступ захтева око kn упоређивања. Ово је боље од сортирања низа, све док k не постане реда величине $\log n$. Међутим, постоји други алгоритам који ефикасно проналази k-ти најмањи елемент за произвољно k.

Идеја је применити разлагање на исти начин као и код алгоритма сортирања 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 елемент, а онда алгоритам применити рекурзивно *само на тај подниз*. Остали елементи могу се игнорисати.

$Selekcija(X, n, k)$
улаз: X - низ од n бројева и k - природни број
излаз: S - k-ти најмањи елемент низа X
1 **if** $k < 1$ **or** $k > n$ **then**
2 *print* "грешка"
3 **else**
4 $S \leftarrow Sel(X, 1, n, k)$
5 **return** S

$Sel(X, Levi, Desni, k)$
улаз: X - низ од n бројева који се разматра у границама од $Levi$ до $Desni$
k - природни број, $1 \leq k \leq Desni - Levi + 1$
излаз: S - k-ти најмањи елемент међу њима
1 **if** $Levi = Desni$ **then**
2 **return** $Levi$
3 **else**
4 $S \leftarrow Razdvajanje(X, Levi, Desni)$
5 **if** $S - Levi + 1 \geq k$ **then**
6 $S \leftarrow Sel(X, Levi, S, k)$
7 **else**
8 $S \leftarrow Sel(X, S + 1, Desni, k - (S - Levi + 1))$
9 **return** S

Сложеност: У просеку се очекује да сложеност алгоритма $T(n)$ буде линеарна, јер задовољава (грубо) рекурентну релацију $T(n) = T(n/2) + O(n)$. Као и код сортирања раздвајањем, лош избор пивота води квадратном алгоритму. Средња сложеност овог алгоритма може се оценити на 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 елемент. Претпоставимо да после раздвајања (које се састоји од $n - 1$ упоређивања) 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 елемент. За $i < k$ (односно $i > k$) долази се до рекурзивног позива алгоритма сложености $T(n-i, k-i)$ (односно $T(i-1, k)$) јер преостаје проналажење $(k - i)$-тог најмањег елемента од њих $n - i$ (односно k-тог најмањег елемента од њих $i - 1$). Према томе, диференцна једначина за $T(n, k)$ је

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

Сменом $j = n - i$ у првој суми и $j = i - 1$ у другој суми добијамо:

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

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

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

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

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.

6.3.5 Множење полинома

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 $n-1$ задата низовима својих коефицијената.

Проблем: Израчунати производ $R = P \cdot Q$, односно одредити коефицијенте тог полинома.

Природно је поћи од израза

$$PQ = \left(p_{n-1}x^{n-1} + \cdots + p_0\right)\left(q_{n-1}x^{n-1} + \cdots + q_0\right) =$$
$$p_{n-1}q_{n-1}x^{2n-2} + \cdots + \left(p_{n-1}q_{i+1} + p_{n-2}q_{i+2} + \cdots + p_{i+1}q_{n-1}\right)x^{n+i} + \cdots + p_0q_0.$$

Коефицијенти полинома PQ могу се израчунати директно из ове једнакости, 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. Приказаћемо сад алгоритам сложености $O(n^{log_2 3})$ заснован на разлагању.

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

$$Q_1 = q_0 + q_1 x + \cdots + q_{n/2-1} x^{n/2-1}, \quad Q_2 = q_{n/2} + q_{n/2+1} x + \cdots + q_{n-1} x^{n/2-1}.$$

Сада имамо

$$PQ = (P_1 + P_2 x^{n/2})(Q_1 + Q_2 x^{n/2}) = P_1 Q_1 + (P_1 Q_2 + P_2 Q_1) x^{n/2} + P_2 Q_2 x^n.$$

У изразу за PQ појављују се производи полинома степена $n/2-1$, који се могу израчунати индукцијом (рекурзивно). Комбиновањем 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 комбиновању тих производа. Према мастер теореми решење диференцне једначине је $T(n) = O(n^2)$, 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 $A + (B + C)x^{n/2} + Dx^n$. Може се приметити 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 $B + C = E - A - D$. Приметимо да је полином E производ два полинома степена $n/2-1$. 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 на необичан начин: добијени су од њих сабирањем коефицијената чији се индекси разликују за $n/2$. Овај неинтуитивни начин множења полинома знатно смањује број операција за велике вредности n.

Пример: 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 (и на линеарне полиноме). Уштеда је наравно много већа за велике n.

Множење бројева

Множење n-битних бројева u и v директно се своди на множење два полинома P и Q степена $n - 1$: коефицијенте полинома треба заменити бинарним цифрама ових бројева, x заменити са 2, израчунати производ $R = PQ$, после чега још остаје да се добијени производ "нормализује", тј. претвори у обичан бинарни број. Сложеност ове нормализације је $O(n)$, јер производ два n-битна броја има највише $2n$ бита.

Претпоставимо да су нам дата два n-битна броја u и v (једноставности ради претпоставићемо да је n степен двојке). Означимо са u_0 нижих $n/2$ битова броја u, а са u_1 виших $n/2$ битова, и аналогно за v_0 и v_1. Онда можемо записати бројеве u и v као:

$$u = 2^{n/2}u_1 + u_0$$

$$v = 2^{n/2}v_1 + v_0$$

Производ бројева u и v можемо записати као производ три броја дужине $n/2$:

$$uv = (2^{n/2}u_1 + u_0)(2^{n/2}v_1 + v_0) = 2^n u_1 v_1 + 2^{n/2}(u_1 v_0 + v_1 u_0) + u_0 v_0$$

$$= 2^n u_1 v_1 + 2^{n/2}((u_1 - u_0)(v_0 - v_1) + u_1 v_1 + u_0 v_0) + u_0 v_0$$

$$= u_1 v_1 (2^n + 2^{n/2}) + 2^{n/2}(u_1 - u_0)(v_0 - v_1) + u_0 v_0 (2^{n/2} + 1)$$

Овим се добија алгоритам чије време извршавања задовољава рекурентну једначину $T(n) = 3T(\frac{n}{2}) + O(n)$. Дакле сложеност изложеног алгоритма је $O(n^{\log_2 3})$.

6.3.6 Штрасенов алгоритам за множење матрица

Претпоставимо да је задатак помножити две квадратне матрице P и Q димензије n, односно израчунати $R = P \cdot Q$. На проблем множења матрица може се применити поступак разлагања, слично као на множење полинома. Због једноставности претпоставићемо да је димензија матрице n степен двојке. Нека је

$$P = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \quad Q = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \tag{6.1}$$

где су a, b, c, d, односно A, B, C, D $n/2 \times n/2$ матрице. Применом разлагања 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:

$$PQ = \begin{bmatrix} aA + bC & aB + bD \\ cA + dC & cB + dD \end{bmatrix},$$

Замењујући свако множење рекурзивним позивом, добијамо диференцну једначину $T(n) = 8T(n/2) + O(n^2)$ (комбиновање мањих решења састоји се од четири сабирања матрица реда $n/2$, укупно n^2 сабирања), 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. Производ PQ може се израчунати на следећи начин:

$$PQ = \begin{bmatrix} z_1 + z_4 & z_2 - z_3 + z_4 + z_5 \\ z_1 + z_3 + z_6 + z_7 & z_2 + z_6 \end{bmatrix},$$

при чему су са z_1, z_2, ..., 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 $z_7 = (d - b)(C - D)$. Ова чињеница може се непосредно проверити. На пример, збир $z_2 + z_6 = c(B + D) + (d - c)D = cB + dD$

Сложеност: У алгоритму се израчунава седам производа матрица два пута мање димензије и користи се константан број сабирања таквих матрица. Дакле, диференцна једначина за сложеност овог алгоритма је $T(n) = 7T(n/2) + O(n^2)$ и њено решење је $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$, што значи да је Штрасенов алгоритам асимптотски бржи од обичног множења матрица.

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 множења матрица за велике вредности n, а може се искористити и у другим проблемима са матрицама, као што су инверзија матрице и израчунавање детерминанте.

6.3.7 Налажење две најближе тачке

Претпоставимо да су задате локације n објеката и да је задатак проверити постоје ли међу њима два објекта, 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.

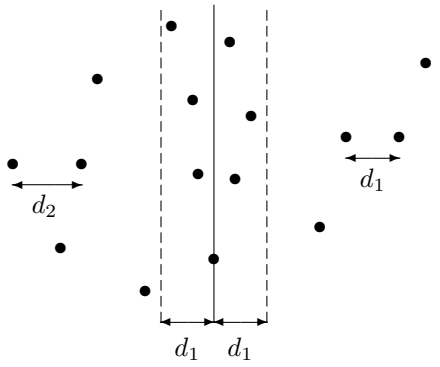
Директни приступ

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 пронаћи ефикаснији алгоритам за велике n.

Алгоритам заснован на разлагању

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. 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 пронађу 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 6.6 (ако више тачака лежи на правој поделе, тачке се могу на произвољан начин разделити између скупова). Начин поделе

изабран је тако да се максимално поједностави обједињавање решења мањих проблема. Сортирање треба извршити само једном.

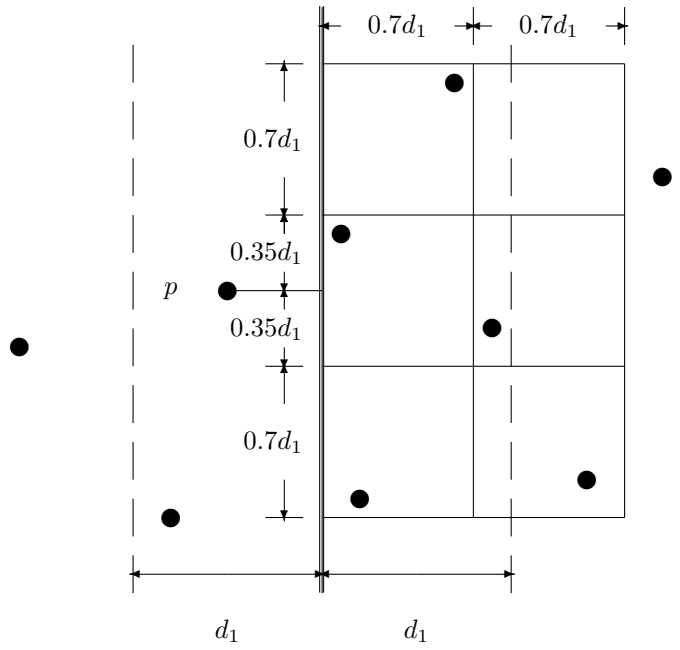


Слика 6.6: Проблем налажења две најближе тачке.

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. Најпре се скуп P дели на два подскупа P_1 и P_2 чије се величине разликују највише за један 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 6.6). 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 тој траци, па не можемо да себи приуштимо примену тривијалног алгоритма на њих.

Друго мање очигледно запажање је да је за произвољну тачку p у траци довољно проверити само њено растојање од највише 8 тачака из супротне траке: 4 које јој претходе, и 4 које су иза ње у сортираном редоследу по y-координатама. 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).

$Najblizi\_par1(p_1, p_2, \ldots, p_n)$
улаз: $p_1, p_2, \ldots, p_n$ - скуп од n тачака у равни
излаз: d - растојање између две најближе тачке скупа
1 Сортирати тачке према x координатама
2 {Ово сортирање извршава се само једном, на почетку}
3 Поделити скуп на два истобројна подскупа
4 Рекурзивно израчунати минимално растојање у оба подскупа
5 Нека је d мање од два минимална растојања
6 Елиминисати тачке које су на растојању већем од d од праве поделе



Слика 6.7: Оцена максималног броја тачака из супротног подскупа, блиских фиксираној тачки.

```
7   Сортирати преостале тачке према y координатама
8   За преостале тачке проверити њихова растојања од четири претходне
9   и четири наредне тачке из супротне траке
10  {довољно је проверити и само четири претходне тачке!}
11  if неко од ових растојања је мање од d then
12    ажурирати d
```

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 $T(n) < cn(\log_2 n - 1)^2 + cn \log_2 n =$

$cn(\log_2^2 n - \log_2 n + 1) < cn\log_2^2 n$. Сложеност $O(n\log^2 n)$ је асимптотски боља од квадратне, али видећемо да се може још мало побољшати.

Доказ коректности алгоритма: Остаје још да се докаже коректност изложеног алгоритма, односно чињенице да је довољно за сваку тачку проверити њено растојање од 4 пре ње и 4 после ње у супротној траци, сортирано по y-координатама. 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 6.7), 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 $0.7d_1$. Обзиром да важи $\sqrt{2}/2 > 0.7$, дијагонала ових квадрата мања је од d_1. Из ове чињенице следи да се у сваком од малих квадрата може 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 шест квадрата.

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. 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.

$Najblizi\_par2(p_1, p_2, \ldots, p_n)$
улаз: $p_1, p_2, \ldots, p_n$ - скуп од n тачака у равни
излаз: d - растојање између две најближе тачке скупа
1 Сортирати тачке према x координатама
2 {Ово сортирање извршава се само једном, на почетку}
3 Поделити скуп на два истобројна подскупа
4 Рекурзивно извршити следеће:
5 Израчунати минимално растојање у оба подскупа

```
6       Сортирати тачке у сваком делу према y координатама
7     Објединити два сортирана низа у један
8     {Обједињавање се мора извршити пре елиминације}
9     {Следећем нивоу рекурзије се мора испоручити сортиран комплетан скуп}
10    Нека је d мање од два минимална растојања
11    Елиминисати тачке које су на растојању већем од d од праве поделе
12    За преостале тачке проверити њихова растојања од четири претходне
13    и четири наредне тачке из супротне траке
14    {довољно је проверити и само четири претходне тачке!}
15    if  неко од ових растојања је мање од d  then
16      поправити d
```

6.4 Претрага са враћањем

У потрази за основним принципима конструкције алгоритама, **претрага са враћањем** или скраћено **претрага** (енгл. backtracking) представља једну од најопштијих техника. Многи проблеми који се тичу тражења скупа решења или који имају за циљ одређивање оптималног решења које задовољава неки скуп ограничења могу се решити формулисањем у виду проблема претраге са враћањем. Назив backtrack смислио је Д. Х. Лемер педесетих година прошлог века[1].

У великом броју примена метода претраге са враћањем, тражено решење се може изразити n-торком $(x_1, x_2, \ldots, x_n)$ где се x_i бира из неког коначног скупа S_i. Често се у проблему који се решава захтева да се нађе вектор (n-торка) за који се максимизује (или минимизује или задовољава) функција $P(x_1, x_2, \ldots, x_n)$. Некада се траже сви вектори који задовољавају P. На пример, сортирање низа целих бројева $A[1..n]$ је проблем чије се решење може представити n-торком x, где је x_i индекс i-тог најмањег елемента низа A. Функција P се у овом случају може задати као конјункција услова $A[x_i] \leq A[x_{i+1}]$ за $1 \leq i < n$. Скуп S_i је коначан и садржи целе бројеве од 1 до n. Иако сортирање није проблем који се обично решава методом претраге са враћањем, он представља један од познатих проблема чије се решење може формулисати у виду n-торке.

Ако са m_i означимо величину скупа S_i, онда постоји $m = m_1 \cdot m_2 \cdots m_n$ n-торки које представљају потенцијалне кандидате за решење. Алгоритам грубе силе би формирао све n-торке, за сваку од њих израчунао вредност функције P и вратио ону која постиже оптималну вредност. Алгоритам претраге са враћањем одликује својство да може да врати исти одговор са често доста мање од m покушаја. Основна идеја је да се пронађе и користи модификована функција $P(x_1, \ldots, x_i)$ (**ограничавајућа функција**) која за део вектора решења $(x_1, x_2, \ldots, x_i)$ може да установи да ли има икакву шансу за успех. Главна предност овог метода је следећа: ако је јасно да парцијални вектор $(x_1, x_2, \ldots, x_i)$ не може ни на који начин да буде део оптималног решења, онда се $m_{i+1} \cdots m_n$ могућих тест вектора у потпуности игнорише, односно прескаче.

Многи од проблема које решавамо методом претраге са враћањем захтевају да сва решења задовољавају сложени скуп ограничења. За сваки

[1] Овај део текста написан је на основу [4].

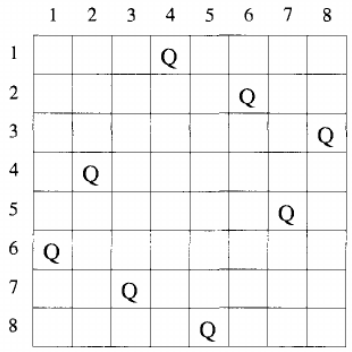
проблем ова ограничења се могу поделити у две категорије: експлицитна и имплицитна.

Експлицитна ограничења су правила која се односе на појединачне координате x_i, као што је на пример:

$l_i \leq x_i \leq u_i$ и x_i је целобројно, тј. $S_i = \{a \in Z : l_i \leq a \leq u_i\}$

Имплицитна ограничења су правила која се односе на две или више компоненти n-торке.

Пример 6.1 (Проблем 8 дама)**.** Класични комбинаторни проблем је поставити осам дама на 8×8 шаховској табли тако да се никоје две даме не нападају, тј. да никоје две не буду у истој врсти, колони нити дијагонали. Нумеришимо врсте и колоне шаховске табле бројевима од 1 до 8. Можемо, такође, и даме нумерисати бројевима од 1 до 8. С обзиром на то да две даме не могу бити у истој врсти, можемо без губитка општости претпоставити да се дама i поставља у врсту i. Сва решења проблема 8 дама се стога могу представити осморкама $(x_1, x_2, \ldots, x_8)$, при чему је са x_i означена колона у коју је смештена i-та дама. Експлицитна ограничења у овој формулацији су $S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $1 \leq i \leq 8$. Стога се простор решења састоји од 8^8 осморки. Имплицитна ограничења у овом случају су да никоја два x_i не смеју бити иста (тј. све даме морају бити у различитим колонама) и никоје две даме не смеју бити на истој дијагонали. Прво од ова два ограничења има за последицу да су сва решења *пермутације* осморке $(1, 2, 3, 4, 5, 6, 7, 8)$. Овим се простор решења смањује са 8^8 на 8! осморки. На слици 6.8 приказано је решење проблема коме одговара осморка $(4, 6, 8, 2, 7, 1, 3, 5)$.

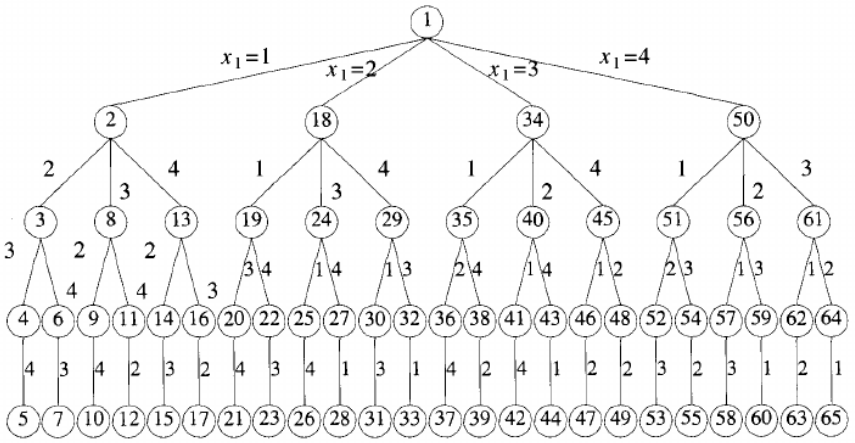


Слика 6.8: Једно решење проблема 8 дама

Алгоритми претраге са враћањем одређују решења проблема систематичном претрагом простора решења за дату инстанцу проблема. Ова претрага се може олакшати представљањем простора решења коренским стаблом. Понекад је могуће исти простор решења представити различитим коренским стаблима. Следећа два примера илуструју представљање просора решења стаблом.

Пример 6.2 (Проблем n дама)**.** Проблем n дама представља уопштење проблема 8 дама: потребно је поставити n дама на шаховској табли димензије $n \times n$ тако да се никоје две не нападају. Претходно разматрање показује

да се простор решења састоји од $n!$ пермутација n-торке $(1, 2, \ldots, n)$. На слици 6.9 приказано је једно могуће стабло за случај $n = 4$. Овакво стабло називамо **стаблом пермутација**. Гране су означене могућим вредностима за x_i. Гране које воде од чвора нивоа 0 до чворова нивоа 1 задају вредности за x_1. Тако на пример, лево подстабло одговара решењима код којих је $x_1 = 1$, његово лево подстабло решењима код којих је $x_1 = 1$ и $x_2 = 2$ и тако даље. Гране које воде од нивоа $i - 1$ до нивоа i су означене могућим вредностима за x_i. Простор могућих решења одговара свим путевима од корена до листа. Стабло на слици 6.9 има укупно $4! = 24$ листова. Стабло решења се може обилазити рецимо алгоритмом DFS; на слици 6.9 чворови стабла решења нумерисани су у складу са редоследом обиласка алгоритмом DFS.

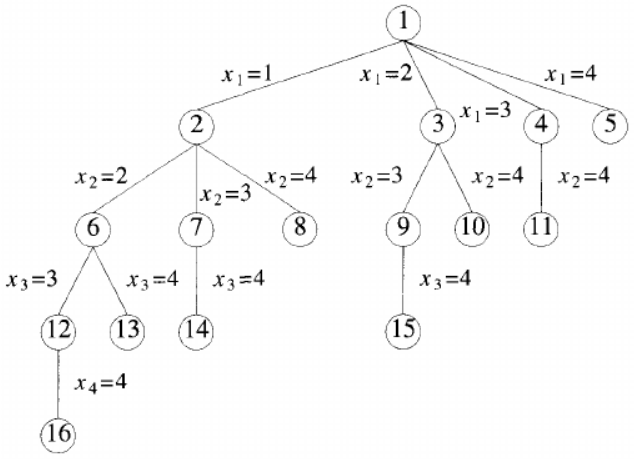


Слика 6.9: Стабло простора решења проблема 4 дама. Чворови су нумерисани као у алгоритму DFS

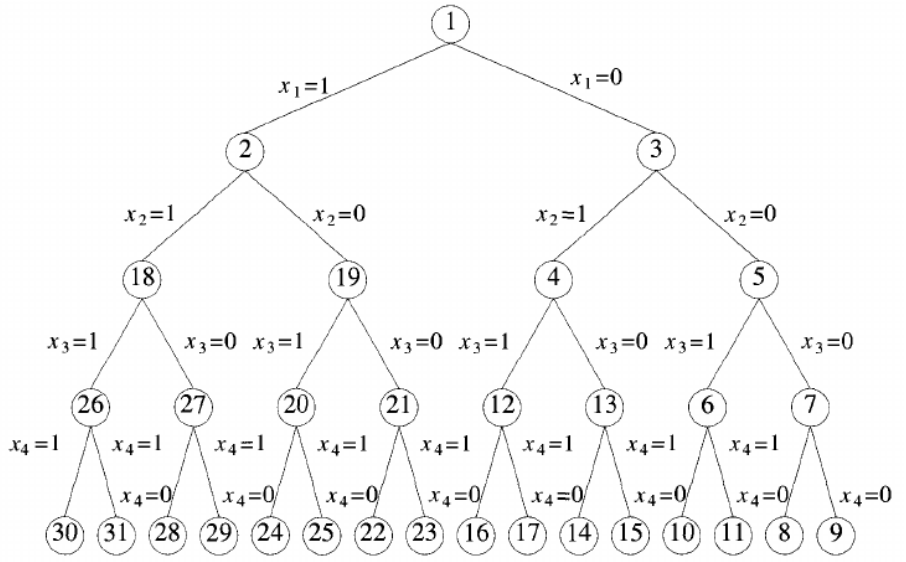
Пример 6.3 (Тражење подскупа бројева са задатим збиром)**.** За дати скуп позитивних бројева w_i, $1 \leq i \leq n$ и број m, потребно је пронаћи све подскупове бројева w_i чија је сума m. На пример, за $n = 4$ и $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$ и $m = 31$, тражени подскупови би били $(11, 13, 7)$ и $(24, 7)$. Уместо да представимо решење као вектор са компонентама w_i чија је сума m, можемо да представимо вектор решења задавањем индекса ових вредности w_i. У овој нотацији ова два решења бисмо записали као $(1, 2, 4)$ и $(3, 4)$. У општем случају сва решења су неке k-торке $(x_1, x_2, \ldots, x_k)$, $1 \leq k \leq n$ и различитим решењима могу одговарати торке различите дужине. Експлицитна ограничења би у овом случају била $x_j \in \{1, 2, \ldots, n\}$, а имплицитна ограничења да никоја два x_i нису иста, а сума w_{x_i} је мања или једнака m. Да бисмо избегли генерисање вишеструких инстанци истог решења, још једно имплицитно ограничење било би да је $x_i < x_{i+1}, 1 \leq i < k$.

Решења се могу представити на други начин, n-торком $(x_1, x_2, \ldots, x_n)$ тако да је $x_i \in \{0, 1\}, 1 \leq i \leq n$, при чему је $x_i = 0$ ако w_i није укључено у суму, а $x_i = 1$ ако јесте. Претходна решења бисмо сад записали као $(1, 1, 0, 1)$ и $(0, 0, 1, 1)$. На овај начин се сва решења могу изразити торкама

фиксне дужине. Закључујемо да је могуће да постоји више начина да се решења представе као торке које задовољавају нека ограничења. Може се показати да се у обе формулације проблема простор решења састоји од 2^n различитих торки.



Слика 6.10: Једна могућа огранизација простора решења за проблем суме подскупова. Бројеви уз чворове одговарају BFS поретку



Слика 6.11: Друго могуће стабло простора решења за проблем суме подскупова. Број листова стабла је 16, што је једнако броју чворова у претходном стаблу.

На сликама 6.10 и 6.11 приказана су стабла простора решења за сваку од ове две формулације за случај $n = 4$. Прва слика одговара променљивој

величини торке. Гране су означене тако да грана од чвора нивоа $i-1$ до чвора нивоа i представља вредност за x_i. У сваком чвору простор решења се дели на просторе делимичних решења. Простор решења се дефинише путевима од корена до *произвољног чвора* у стаблу, с обзиром на то да сваки овакав пут одговара подскупу који задовољава експлицитна ограничења. Могући путеви су () (празан пут од корена до њега самог), (1), (1, 2), (1, 2, 3), (1, 2, 3, 4), (1, 2, 4), (1, 3, 4), (2), (2, 3), итд. Друга слика одговара формулацији са фиксном величином торке. Гране од чвора нивоа $i-1$ до чвора нивоа i означене су вредностима x_i, које могу бити 0 или 1. Сви путеви од корена до *листа* дефинишу простор решења. Стабло на слици 6.11 има 2^4 листова који дају 16 могућих торки.

Сваки чвор у стаблу одговара **стању проблема**. **Прихватљива стања** су она стања проблема s за које пут од корена до чвора s дефинише торку у простору решења. У стаблу са слике 6.10 сви чворови су прихватљива стања, док у стаблу са слике 6.11 само листови представљају прихватљива стања. **Циљна стања** су она прихватљива стања s за које пут од корена до чвора s дефинише торку која је елемент скупа решења (тј. задовољава имплицитна ограничења) проблема. Стабло које одговара простору решења називамо **стаблом простора стања**.

У сваком унутрашњем чвору стабла у претходним примерима простор решења се дели у дисјунктне просторе делимичних решења. На пример, у чвору 1 стабла приказаног на слици 6.9 простор решења се дели у четири дисјунктна скупа. Подстабла са коренима у чворовима 2, 18, 34 и 50 представљају све елементе простора решења код којих је $x_1 = 1$, 2, 3 и 4, редом. У чвору 2 простор делимичних решења за $x_1 = 1$ се даље дели у три дисјунктна скупа.

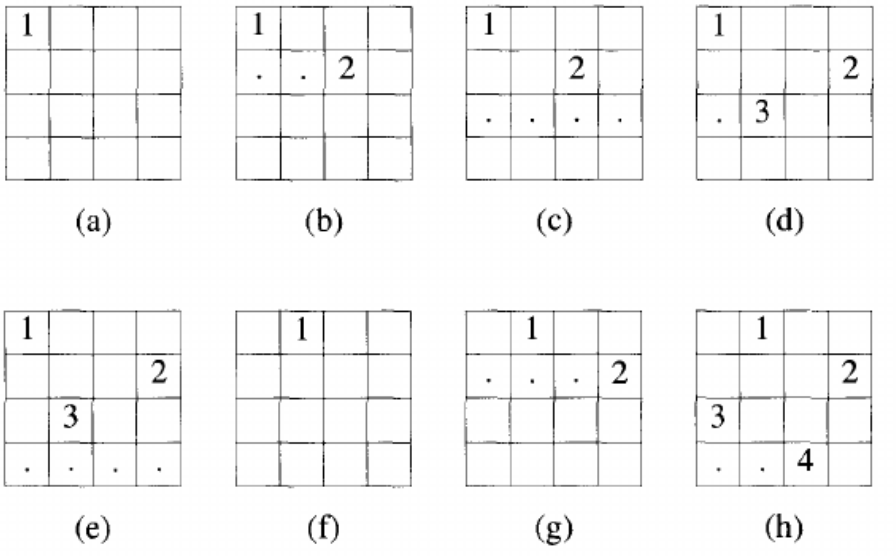
Стабла простора стања описана у примеру суме подскупа називају се **статичким стаблима**, јер њихова структура не зависи од инстанце проблема коју решавамо. За неке проблеме је корисно користити различите организације стабла за различите инстанце проблема. У том случају организација стабла се одређује динамички током претраге простора решења. Стабла простора стања која су зависна од инстанце проблема називају се **динамичким стаблима**. На пример, посматрајмо формулацију са фиксном величином торке у проблему суме подскупа. Коришћењем организације динамичког стабла, једна инстанца проблема за $n = 4$ се може решити коришћењем организације приказане на слици 6.11. Друга инстанца проблема за $n = 4$ може се решити тако што у нивоу 1 подела одговара изборима $x_2 = 1$ и $x_2 = 0$.

Када је једном осмишљено стабло простора стања, проблем се може решити систематичним генерисањем стања проблема, одређивањем која су од њих су прихватљива стања и коначно одређивањем која прихватљива стања су и циљна стања. Постоје два фундаментално различита начина за генерисање стања проблема. Оба приступа крећу од корена и генеришу остале чворове. Чвор који је генерисан и за кога нису генерисани сви синови називамо **живим (отвореним) чвором**. Живи чвор чији се синови тренутно генеришу, односно чвор чија се експанзија (проширивање) тренутно врши, називамо **Е-чвор**. **Мртав (затворени) чвор** је генерисани чвор који даље неће бити проширивани, односно чији су сви синови већ генерисани. У оба приступа одржавамо листу живих чворова.

- У складу са првим методом, чим се генерише нови син C текућег Е-чвора R, овај син постаје нови Е-чвор. R ће поново постати Е-чвор када се подстабло C потпуно истражи. Ово одговара генерисању стања проблема у дубину.
- У складу са другим методом Е-чвор остаје Е-чвор све док је жив, тј. док се не заврши са његовим проширивањем.

Оба метода поразумевају коришћење **ограничавајућих функција** за "убијање" (затварање отворених) живих чворова без генерисања њихових синова. Ограничавајуће функције користе се пажљиво, тако да се гарантује генерисање бар једног циљног чвора, односно генерисање свих чворова одговара ако је задатак да се нађу сва решења проблема. Генерисање чворова у дубину назива се **претрага са враћањем** (или само претрага). Методе генерисања стања код којих Е-чвор остаје Е-чвор све док не буде мртав води ка алгоритмима **гранања са одсецањем**, видети одељак 6.4.4.

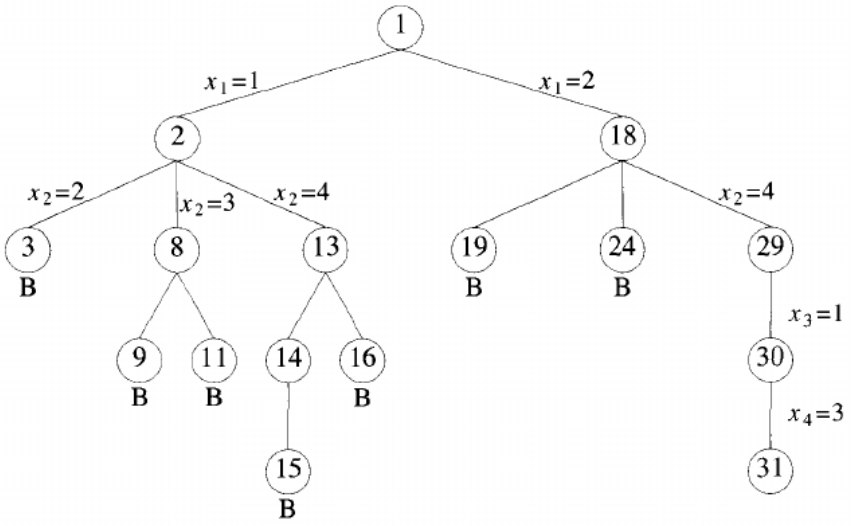
Чворови на слици 6.9 нумерисани су према редоследу генерисања који одговара претрази у дубину. Чворови на сликама 6.10 и 6.11 нумерисани су методом код кога Е-чвор остаје Е-чвор све дотле док не постане мртав чвор. Код стабла на слици 6.10 сваки нови чвор смешта се у ред. Када се заврши генерисање свих синова тренутног Е-чвора, наредни чвор са почетка реда постаје нови Е-чвор; ово одговара претрази стабла у ширину. Код стабла на слици 6.11 чворови се смештају у стек уместо у ред, а одговарајућа претрага зове се D-претрага.



Слика 6.12: Пример решења заснованог на претрази са враћањем за проблем 4 дама

Пример 6.4. Погледајмо како претрага са враћањем ради на примеру проблема 4 даме. Као ограничавајућу функцију узећемо очигледни критеријум да ако је $(x_1, x_2, \ldots x_i)$ пут до текућег Е-чвора, тада сваком сину

тог чвора до кога води грана означена са x_{i+1} одговара скуп дама на позицијама $(1, x_1), (2, x_2), \ldots, (i+1, x_{i+1})$ при чему се никоје две даме не нападају. Почињемо са кореном као јединим живим чвором. Он постаје Е-чвор и пут је (). Генеришемо једног сина. Претпоставимо да се синови генеришу у растућем редоследу. Стога се генерише чвор 2 са слике 6.9 и пут је сада (1). Ово одговара постављању даме 1 у колону 1. Чвор 2 постаје Е-чвор. Генерише се чвор 3 и одмах затим убија. Следећи чвор који се генерише је чвор 8 и пут постаје (1,3). Сада чвор 8 постаје Е-чвор. Ипак, и он се убија јер сви његови синови представљају конфигурације које не воде циљном чвору. Враћамо се у чвор 2 и генеришемо новог сина 13. Пут је сада (1,4). На слици 6.12 приказане су конфигурације шаховске табле како претрага напредује. Тачке одговарају постављањима даме која су покушана, а нису дозвољена због напада неке друге даме. На слици 6.12(b) друга дама се поставља у колоне 1 и 2 и најзад се коначно смешта у колону 3. На слици 6.12(c) алгоритам покушава са све четири колоне и није у могућности да смести наредну даму на таблу. Сада се морамо вратити уназад. На слици 6.12(d) помера се друга дама у колону 4, а трећа дама у колону 2. Наредни дијаграми показују преостале кораке алгоритма све док се не пронађе решење.



Слика 6.13: Део стабла са слике 6.9

На слици 6.13 приказан је део стабла са слике 6.9 који се генерише. Чворови су нумерисани према редоследу којим су генерисани. Чвор који се убија као резултат ограничавајуће функције има слово B испод себе. Приметимо да стабло са слике 6.9 има 31 чвор за разлику од новог стабла које има 16 чворова.

Посматрајмо сада методу претраге са враћањем у општем случају. Претпоставимо да је задатак пронаћи све циљне чворове, а не само један. Нека је $(x_1, x_2, \ldots, x_i), 1 \leq i \leq n$, пут од корена до чвора у стаблу простора стања. Означимо са $T(x_1, x_2, \ldots, x_i)$ скуп свих могућих вредности за x_{i+1}

тако да је $(x_1, x_2, \ldots, x_{i+1})$ такође пут до стања проблема. Специјално, $T(x_1, x_2, \ldots, x_n) = \emptyset$. Претпостављамо да је B_{i+1} одговарајућа логичка ограничавајућа функција такве да ако је $B_{i+1}(x_1, x_2, \ldots, x_{i+1})$ нетачно за пут $(x_1, \ldots, x_{i+1})$ од корена до стања проблема, онда пут не може бити продужен тако да досегне неки од циљних чворова. Природно је описати алгоритам претраге са враћањем на рекурзиван начин.

```
Backtrack(k)
X - вектор решења, n - димензија вектора – глобалне променљиве
улаз: k - индекс компоненте X чија се вредност бира у текућем позиву
1 for X[k] ∈ T(X[1..k − 1]) do
2   if B_k(X[1..k]) then
3     if X[1..k] је пут до циљног чвора then
4       print(X[1..k])
5     if k < n then
6       Backtrack(k + 1)
```

Овај алгоритам позивамо са $Backtrack(1)$. Приметимо да се овим алгоритмом штампају сва решења и претпоставља се да торке различитих дужина могу да сачињавају решење. Ако је потребно само једно решење, може се додати флег као параметар као индикатор прве појаве успеха.

Приказаћемо и итеративну верзију овог уопштеног алгоритма. Приметимо да се елементи генеришу методом у дубину.

```
Backtrack_it(X, n)
улаз: X - вектор решења, n - димензија вектора
излаз: одштампане све n-торке X које задовољавају услове
1 k ← 1
2 while k ≠ 0 do
3   if постоји неко X[k] ∈ T(X[1..k − 1]) које није обрађено и
       B_k(X[1..k]) then
4     if X[1..k] представља пут до циљног чвора then
5       print(X[1..k])
6     k ← k + 1
7   else k ← k − 1
```

6.4.1 Проблем n дама

За проблем n дама, приказаћемо алгоритам за генерисање свих конфигурација које одговарају решењима. Коду претходи код алгоритма за проверу да ли је дозвољено поставити k-ту даму на позицију i у k-тој врсти.

```
Smesti(k, i)
улаз: X - глобални низ чији су елементи X[1..k − 1] већ постављени,
      k - индекс врсте, i - индекс колоне
излаз: true ако се дама може поставити у k-тој врсти и i-тој колони;
       иначе false
1 for j ← 1 to k − 1 do
2   if X[j] = i {две даме у истој колони}
      or |X[j] − i| = |j − k| { две даме на истој дијагонали } then
3     return false
4 return true
```

$NDama(k)$
X - вектор решења, n - димензија вектора – глобалне промењиве
улаз: X - низ у који смештамо индексе колона решења n дама,
k - индекс врсте до које се стигло, n - димензија шаховске табле
излаз: штампају се сва могућа постављања n дама на таблу димензије $n \times n$
са постављених $k-1$ првих дама

```
1 for i ← 1 to n do
2   if Smesti(k, i) then
3     X[k] ← i
4     if k = n then
5       print(X[1..n])
6     else
7       NDama(k + 1)
```

Списак свих решења добија се позивом $NDama(1)$. Због симетрије, довољно је за прву врсту разматрати само колоне од 1 до $\lceil n/2 \rceil$.

6.4.2 Збир подскупа

Претпоставимо да имамо на располагању n различитих позитивних бројева и да нам је циљ да пронађемо све подскупове ових бројева чији је збир једнак m. У одељку 6.3 показано је како се овај проблем може формулисати коришћењем торки фиксне или променљиве величине. Размотрићемо сад решење засновано на претрази коришћењем стратегије са торкама фиксне величине. У том случају елемент x_i вектора решења има вредност нула или један у зависности од тога да ли је предмет w_i укључен у подскуп или не. Синови сваког чвора са слике 6.11 се једноставно генеришу. За чвор нивоа $i-1$ леви син одговара случају $x_i = 1$, а десни случају $x_i = 0$. Природно је да се за ограничавајућу функцију користи функција:

$$B_k(x_1, \ldots, x_k) = true \text{ акко } \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m \text{ и } \sum_{i=1}^{k} w_i x_i + min_{k+1 \leq j \leq n} w_j \leq m$$

Јасно је да $x_1, \ldots, x_k$ не могу бити на путу ка циљном чвору ако овај услов није задовољен. Ограничавајућа функција се може поједноставити ако се тежине w_i претходно поређају тако да чине неопадајући низ. У овом случају $x_1, \ldots, x_k$ не може водити ка циљном чвору ако је

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} > m$$

Стога користимо функцију ограничавања:

$$B_k(x_1, \ldots, x_k) = true \text{ акко } \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m \text{ и } \sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$$

У усавршеној верзији алгоритма који следи избегава се да се стално рачунају ове две суме, тиме што се користе њихове претходне вредности.

$SumaPodskupova(w, n, m, k, s, r)$
улаз: w - нерастући низ бројева дужине n,
m - тражена сума елемената подскупа
k - индекс елемента низа x који се евентуално укључује у збир
$s = \sum_{j=1}^{k-1} w[j]x[j]$, $r = \sum_{j=k}^{n} w[j]$
излаз: штампају се сви могући подскупови са сумом m
1 {генеришемо левог сина; приметимо: $s + w[k] \leq m$ јер је B_{k-1} тачно}
2 $x[k] \leftarrow 1$
3 **if** $s + w[k] = m$ **then**
4 *print* $x[1..k]$ {пронађено је једно од решења}
5 **else if** $s + w[k] + w[k+1] \leq m$ **then**
6 $SumaPodskupova(w, n, m, k+1, s + w[k], r - w[k])$
7 {генеришемо десног сина и рачунамо B_k користећи претпоставку да је B_{k-1} тачно}
8 **if** $s + w[k+1] \leq m$ **and** $s + r - w[k] \geq m$ **then**
9 $x[k] \leftarrow 0$
10 $SumaPodskupova(w, n, m, k+1, s, r - w[k])$

Алгоритам се позива са $SumaPodskupova(w, n, m, 1, 0, \sum_{i=1}^{n} w_i)$. Интересантно је да алгоритам не користи експлицитно тест $k > n$ за прекид рекурзије. Овај тест није неопходан јер на старту алгоритма важи $s < m$ и $s + r \geq m$. Стога је $r > 0$, тј. сума r садржи бар један сабирак, па је $k \leq n$. Такође ваља приметити да у наредби **else if** из $s + w_k < m$ и $s + r \geq m$ следи да је $r \neq w_k$ и стога $k + 1 \leq n$. Приметимо, такође, да ако важи $s + w_k = m$, онда $x_{k+1}, \ldots, x_n$ морају бити једнаки 0. У оквиру **else if** наредбе не тестирамо $\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$, јер већ знамо да је $s + r \geq m$ и $x_k = 1$.

6.4.3 3-бојење графа

Нека је $G = (V, E)$ неусмерени граф. **Исправно бојење** (или само бојење) графа G је такво придруживање боја свим чворовима, да су суседним чворовима увек придружене различите боје.

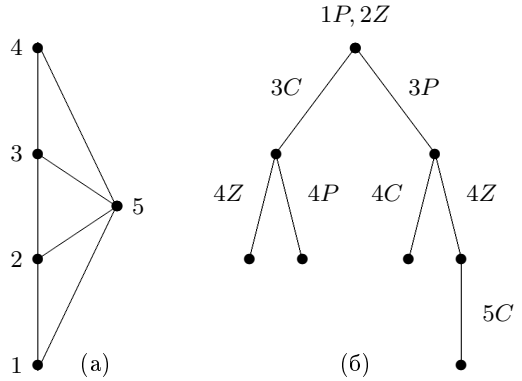
Проблем 3-бојење графа: Дат је неусмерени граф $G = (V, E)$. Установити да ли се G може обојити са три боје.

У случају проблема 3-бојења, 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 претраге са враћањем стабла простора стања (стабла варијанти), који је добра илустрација претраге и гранања са одсецањем. Да бисмо разликовали чворове графа и стабла, чворове стабла звaћемо *теменима*.

Корен стабла одговара почетном стању проблема, а свака грана одговара некој одлуци о вредности неког параметра. Означимо три боје са C (црвено),

P (плаво) и Z (зелено). На почетку се могу изабрати нека два суседна чвора v и w, и обојити рецимо са P и Z. 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 стабла варијанти. Стабло се конструише у току самог обиласка. У сваком темену t стабла бира се следећи чвор u графа који треба обојити, 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 Z), онда постоје две могућности за бојење u, P и C, па се корену додају два 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 тренутку када 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 неком темену на нижем нивоу) и покушавамо са другим синовима. Пример графа, и стабла које одговара решавању проблема 3-бојење на том графу, приказан је на слици 6.14. Приметимо да када се у овом примеру 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.



Слика 6.14: Пример примене претраге на 3-бојење графа.

Алгоритам за обилазак стабла варијанти може се формулисати индукцијом по броју обојених чворова.

Индуктивна хипотеза: Умемо да завршимо 3-бојење графа који има мање од k необојених чворова, или да за такав граф установимо да се не може обојити са три боје.

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.

```
3 – bojenje(G, U)
улаз: G = (V, E) - неусмерени граф,
      U - скуп обојених чворова (на почетку празан)
излаз: придруживање једне од три боја сваком чвору
1 if U = V then
2    print "бојење је завршено"
3    прекид свих рекурзивних позива
4 else
5    изабрати неки чвор v ∈ V, v ∉ U
6    for C ← 1 to 3 do
7       if  ниједан сусед v није обојен бојом C then
8          нека је U_1 скуп добијен од скупа U додавањем чвора v обојеног бојом C
9          3-бојење(G, U_1)
```

Није тешко наћи пример графа и редослед његових чворова, такав да се при бојењу графа добија стабло са експоненцијалним бројем темена. 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 примену одговарајуће хеуристике. *Хеуристика* је део алгоритма који не гарантује да ће се до решења доћи ефикасније, али се претпоставља на основу искуства или неке анализе да је претрага на тај начин ефикаснија. У овом примеру хеуристика би могла да буде да се у сваком кораку за бојење бира чвор који има највише необојених суседа или рецимо чвор чији су суседи већ обојени највећим бројем различитих боја.

6.4.4 Бојење графа минималним бројем боја

Посматрајмо општи проблем бојења графа: циљ је пронаћи најмањи број боја потребан за бојење задатог графа, а не установити да ли је граф могуће обојити са три боје. Може се формирати стабло слично као за 3-бојење, али број синова сваког темена 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 да побољша следеће опажање. 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 (извршити одсецање), јер је већ познато боље решење. Према томе, k служи као **граница** за претрагу.

У општем случају у сваком темену стабла варијанти 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 (вршимо одсецање). Важан начин да се алгоритам гранања са одсецањем учини ефикасним је израчунавање добрих доњих граница, односно избор добре ограничавајуће функције; ако се тражи минимум, онда се траже добре горње границе. Други важан елемент је налажење доброг редоследа обиласка, који омогућује брзо проналажење добрих решења, а тиме и раније напуштање неперспективних подстабала.

6.4.5 (0,1) проблем ранца

Подсетимо се проблема ранца. Нека је дат низ вредности артикала $V = [v_1, v_2, \ldots v_n]$, низ тежина артикала $W = [w_1, w_2, \ldots, w_n]$ и капацитет ранца M. Све ове вредности су позитивни цели бројеви. Потребно је одредити вектор $(x_1, ..., x_n), x_i \in \{0, 1\}$ тако да важи $\sum w_i x_i \leq M$, а да вредност суме $\sum v_i x_i$ буде максимална. Кажемо да n-торка $[x_1, x_2, \ldots, x_n]$ нула и јединица представља **прихватљиво решење** ако је $\sum w_i x_i \leq M$.

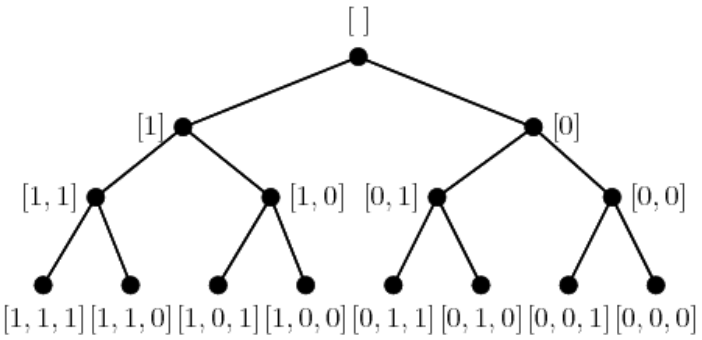
Једноставан алгоритам за решавање овог проблема састоји се од испробавања свих 2^n могућих n-торки нула и јединица (видети [3]). То можемо урадити тако што ћемо прво изабрати вредност за x_1, затим за x_2 итд. Претрага са враћањем представља једноставан метод за генерисање свих могућих n-торки. Након што се свака n-торка генерише, проверава се да ли је прихватљива. Ако јесте прихватљива, онда се њена вредност пореди са тренутно најбољим решењем. Најбоље текуће решење се ажурира кад год се нађе боље прихватљиво решење.

Означимо са $X = [x_1, x_2, \ldots, x_n]$ текућу комплетирану n-торку која се конструише, а са $V\_tekuce = \sum_{i=1}^{n} w_i x_i$ њену вредност. Нека је $OptX$ тренутно најбоље решење, а $OptV$ вредност тог решења. У наставку следи код једноставног рекурзивног алгоритма за решавање овог проблема.

```
Ranac1(l)
V - вектор вредности артикала, W - вектор тежина артикала,
M - капацитет ранца (глобалне променљиве)
улаз: l - редни број координате коју тренутно постављамо
излаз: OptV - вредност оптималног решења,
       OptX -вредност торке за коју се постиже оптимална вредност
1  if l = n + 1 then
2    if Σ_{i=1}^{n} w_i x_i ≤ M then
3      V_tekuce ← Σ_{i=1}^{n} v_i x_i
4      if V_tekuce > OptV then
5        OptV ← V_tekuce
6        OptX ← [x_1, x_2, ..., x_n]
7  else
8    x_l ← 1
9    Ranac1(l + 1)
10   x_l ← 0
```

11 $Ranac1(l+1)$

Пре позива алгоритма треба поставити вредност $OptV$ на нула, а онда позвати алгоритам $Ranac1(1)$ да би се решила инстанца проблема. Рекурзивни позиви имплицитно граде стабло простора стања. За $n = 3$ ово стабло је приказано на слици 6.15.



Слика 6.15: Стабло простора стања за $n = 3$.

Предложени алгоритам генерише 2^n бинарних n-торки. Потребно је $O(n)$ корака да се свако решење провери, те је стога асимптотски време извршавања овог алгоритма $O(n2^n)$. Приметимо да нису све n-торке нула и јединица прихватљиве, па би једноставна модификација овог алгоритма могла то да узме у обзир. За проблем ранца приметимо да мора да важи:

$$\sum_{i=1}^{l-1} w_i x_i \leq M$$

за свако парцијално решење $[x_1, x_2, \ldots, x_{l-1}]$. Другим речима, можемо проверити парцијална решења да видимо да ли је задовољен критеријум прихватљивости. Ако је $l \leq n$ уводимо ознаку:

$$W\_tekuce = \sum_{i=1}^{l-1} w_i x_i$$

У измењеном алгоритму проверава се прихватљивост пре него што се направи рекурзивни позив. Овај алгоритам се позива са $l = 1$ и $W\_tekuce = 0$.

$Ranac2(l, W\_tekuce)$
V - вектор вредности артикала, W - вектор тежина артикала,
M - капацитет ранца, X - вектор решења (глобалне променљиве)
улаз: l - редни број координате коју тренутно постављамо
W_tekuce - текућа тежина ранца
излаз: $OptV$ - вредност оптималног решења,
$OptX$ -вредност торке за коју се постиже оптимална вредност
1 **if** $l = n + 1$ **then**

```
2    if ∑_{i=1}^{n} v_i x_i > OptV then
3       OptV ← ∑_{i=1}^{n} v_i x_i
4       OptX ← [x_1, x_2, ..., x_n]
5  else
6    if W_tekuce + w_l ≤ M then
7       x_l ← 1
8       Ranac2(l + 1, W_tekuce + w_l)
9       x_l ← 0
10      Ranac2(l + 1, W_tekuce)
11   else
12      x_l ← 0
13      Ranac2(l + 1, W_tekuce)
```

Кад год је $W\_tekuce + w_l > M$ имамо одсецање. То значи да се један од два рекурзивна позива (онај који одговара $x_l = 1$) не изводи. Одсецање се може урадити у још неким ситуацијама, коришћењем ограничавајућих функција. Означимо са $vrednost(X)$ вредност произвољног прихватљивог решења X. За свако парцијално прихватљиво решење $X = [x_1, \ldots, x_{l-1}]$ означимо са $P(X)$ максималну вредност произвољног решења које је потомак X у стаблу простора стања. Другим речима $P(X)$ означава максималну вредност $vrednost(X')$, где је $X' = [x'_1, \ldots, x'_n]$ такво да је $x_i = x'_i$ за $1 \leq i \leq l-1$. У принципу $P(X)$ би се могло израчунати проласком кроз подстабло са кореном X. Да бисмо ово избегли, користићемо ограничавајућу функцију. Ограничавајућа функција је реална функција коју означавамо са B дефинисана на скупу стања у стаблу за коју важи: за свако прихватљиво решење X, $B(X) \geq P(X)$. Другим речима, $B(X)$ је горња граница за вредност доступног решења које је потомак парцијалног низа X. Када се зна нека ограничавајућа функција, та функција се може искористити за одсецање у стаблу стања. Претпоставимо да смо у некој фази претраге и да имамо текуће парцијално решење $X = [x_1, \ldots, x_{l-1}]$ и да је $OptV$ текућа оптимална вредност. Увек када важи $B(X) \leq OptV$, важи и $P(X) \leq B(X) \leq OptV$, односно ниједан потомак не може поправити текућу оптималну вредност. Стога можемо одсећи цело текуће подстабло.

Показаћемо сада како дефинисати ефикасну ограничавајућу функцију за проблем ранца коришћењем разломљеног проблема ранца, чија је формулација проблема иста као за $(0,1)$ проблем ранца, само уместо услова $x_i \in \{0,1\}$ имамо услов $0 \leq x_i \leq 1$. Подсетимо се да се разломљени проблем ранца може ефикасно решити похлепним алгоритмом у којем артикле посматрамо у нерастућем редоследу односа вредности према тежини.

Нека $Razlomljeni(k; v_1, \ldots, v_k, w_1, \ldots, w_k, M)$ означава оптимално решење разломљеног проблема ранца за произвољну инстанцу међу онима које садрже неке од првих k предмета (оне предмете којима одговара јединица на одговарајућој координати у вектору X). Овај алгоритам користимо за дефинисање ограничавајуће функције на следећи начин: за прихватљиво парцијално решење $X = [x_1, \ldots, x_{l-1}]$ (0,1) проблема ранца, дефинишемо:

$$B(x) = \sum_{i=1}^{l-1} v_i x_i + Razlomljeni(n-l+1, v_l, \ldots, v_n, w_l, \ldots, w_n, M - \sum_{i=1}^{l-1} w_i x_i)$$

$$= W\_tekuce + Razlomljeni(n-l+1, v_l, \ldots, v_n, w_l, \ldots, w_n, M - W\_tekuce)$$

Овде је $B(X)$ једнако суми већ добијених вредности за објекте $1, \ldots, l-1$ и вредности преосталих $n-l+1$ објеката, коришћењем преосталог капацитета $M - W\_tekuce$, али дозвољавањем нецелобројних вредности за x_i. На овај начин можемо добити већу вредност, стога је $B(X) \geq V(X)$, где је са $V(X)$ означена вредност решења X и B је заиста ограничавајућа функција. С обзиром на то да ју је лако израчунати, корисна је за одсецање.

Вратимо се сада на решавање (0,1) проблема ранца. Пре него што започнемо алгоритам претраге, корисно је сортирати објекте у опадајућем редоследу односа вредности и тежине. Стога за објекте $l, ..., n$ за које се израчунава ограничавајућа функција важи да су већ дати у одговарајућем редоследу, тј. важи $\frac{v_1}{w_1} \geq \ldots \geq \frac{v_n}{w_n}$.

$Ranac3(l, W\_tekuce)$
V - вектор вредности артикала, W - вектор тежина артикала,
M - капацитет ранца, X - вектор решења (глобалне променљиве)
улаз: l - редни број координате коју тренутно постављамо
W_tekuce - текућа тежина ранца
излаз: $OptV$ - вредност оптималног решења,
$OptX$ -вредност торке за коју се постиже оптимална вредност

```
1  if l = n + 1 then
2    if Σ_{i=1}^{n} v_i x_i > OptV then
3      OptV ← Σ_{i=1}^{n} v_i x_i
4      OptX ← [x_1, x_2, ..., x_n]
5  else
6    B ← Σ_{i=1}^{l-1} v_i x_i + Razlomljeni(n − l + 1, v_l, ..., v_n, w_l, ..., w_n, M − W_tekuce)
7    if B ≤ OptV then
8      return {одсецање јер нема бољег решења}
9    if W_tekuce + w_l ≤ M then
10     x_l ← 1
11     Ranac3(l + 1, W_tekuce + w_l)
12   if B ≤ OptV then
13     return
14   x_l ← 0
15   Ranac3(l + 1, W_tekuce)
```

Важно је напоменути да се услов одсецања $B \leq OptV$ проверава пре сваког рекурзивног позива. Ово се ради зато што се вредност $OptV$ може повећати како алгоритам напредује, те стога проверавамо да ли можемо вршити одсецање приликом сваке припреме избора вредности за x_l.

Пример 6.5 (Пример решавања (0,1) проблема ранца)**.** Претпоставимо да на располагању имамо пет објеката тежина редом 11, 12, 8, 7, 9, вредности 23, 24, 15, 13, 16 и капацитет ранца $M = 26$. Приметимо да су објекти

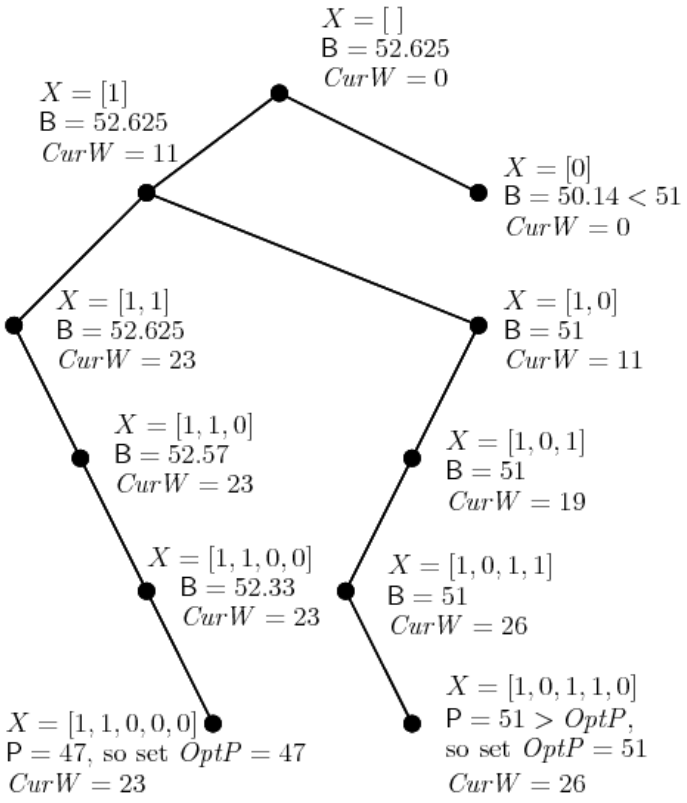
већ поређани у опадајућем редоследу вредности по јединици тежине. На слици 6.16 приказано је стабло простора стања које се обилази у редоследу као што је дато у алгоритму `Ranac3`. У сваком чвору бележимо текуће вредности за X, $B(X)$ и W_tekuce.

Приметимо да се одсецање коришћењем ограничавајуће функције у овом примеру догађа у три групе темена:

1. У тренутку када је $X = [1, 0]$ и $[1, 0, 1]$, најбоља већ пронађена вредност је $OptV = 51$. С обзиром на то да је $B(X) = 51$, други рекурзивни позив се не изводи. Стога се подстабла са кореном $[1, 0, 0]$ и $[1, 0, 1, 0]$ одсецају.

2. Када је $X = [0]$, имамо да важи $OptV = 51$ и $B(X) = 50.14 < 51$, стога се не изводе рекурзивни позиви. Подстабло са кореном $[0]$ се одсеца.

3. Такође, одсецање због премашеног капацитета се јавља у чворовима $[1, 1]$, $[1, 1, 0]$, $[1, 1, 0, 0]$ и $[1, 0, 1, 1]$. У сваком од ових чворова први рекурзивни позив за $x_l = 1$ се не изводи.



Слика 6.16: Стабло простора стања за `Ranac3`.

Глава 7

Генерисање комбинаторних објеката

Алгоритми за генерисање комбинаторних објеката (као што су пермутације, варијације, комбинације) одувек су привлачили пажњу. Нама су интересантни као компонента алгоритама заснованих на грубој сили, односно као примери алгоритама претраге (претраге са враћањем). Проблем који разматрамо је генерисање свих објеката датог типа – на пример, свих пермутација реда n, односно комбинација, партиција и варијација.

7.1 Увод

У вези са сложеношћу ових алгоритама може се разматрати сложеност генерисања, односно сложеност листања свих објеката. На пример, оптимални алгоритам за генерисање свих пермутација реда n био би сложености $O(n!)$, а алгоритам за њихово листање сложености $O(n \cdot n!)$, јер је испис сваке пермутације дужине n сложености $O(n)$. Ако је P број инстанци комбинаторног објекта, а N просечна величина инстанце, онда се каже да алгоритам листа све инстанце за асимптотски оптимално време ако је сложеност алгоритма $O(N \cdot P)$.

Међу низовима $a = a_1, a_2, \ldots, a_p$ и $b = b_1, b_2, \ldots, b_q$ елемената из уређеног скупа **лексикографски поредак** се дефинише на следећи начин: a претходи b (односно $a < b$) ако и само ако постоји индекс i такав да је $a_j = b_j$ за $j < i$ и важи $i = p+1 \leq q$ или $a_i < b_i$. На пример, $11 < 112 < 221$ (овде је $i = 3$, односно $i = 1$). Овај поредак се користи за утврђивање редоследа речи у речнику. На пример, лексикографски редослед подскупова скупа $\{1, 2, 3\}$ представљених као скупова био би: $\emptyset$, $\{1\}$, $\{1, 2\}$, $\{1, 2, 3\}$, $\{1, 3\}$, $\{2\}$, $\{2, 3\}$, $\{3\}$. У бинарној нотацији редослед је нешто другачији: 000, 001, 010, 011, 100, 101, 110, 111, што одговара подскуповима $\emptyset$, $\{3\}$, $\{2\}$, $\{2, 3\}$, $\{1\}$, $\{1, 3\}$, $\{1, 2\}$, $\{1, 2, 3\}$. Као што се види, лексикографски редослед објеката зависи од начина њиховог представљања – различитим нотацијама могу да одговарају различити редоследи истих објеката.

Алгоритми за генерисање комбинаторних објеката могу да буду рекурзивни или итеративни. Итеративни алгоритми обично омогућавају бољу

контролу над начином генерисања наредног објекта, полазећи од текућег. Ми ћемо овде разматрати итеративне алгоритме.

Скоро сви алгоритми за генерисање се заснивају на једној од следеће три идеје:

- коришћење неког начина *нумерисања* (придруживања узастопних целих бројева објектима). На пример, бинарним тројкама одговарају цели бројеви $0, 1, 2, \ldots, 7$, па се тројке могу генерисати тако што се редом за $i = 0, 1, \ldots, 7$ формира троцифрени бинарни запис броја i

- *лексикографско ажурирање*; проналази се најдеснији елемент инстанце који треба "ажурирати" или преместити на нову позицију. Тако се, на пример, наредна бинарна тројка у низу тројки $000, 001, \ldots, 111$ добија тако што се најдеснија нула замени јединицом, па се иза ње допишу нуле

- *правило најмање промене*; прелаз између узастопних објеката врши се помоћу најмањег броја измена; примери примене овог правила су:

  - генерисање у облику Грејовог кода, када су промене теоријски најмање могуће. На пример, у низу бинарних тројки $000, 001, 011, 010, 110, 111, 101, 100$ сваке две узастопне тројке разликују се на тачно једној позицији
  - транспозиције, када се наредна инстанца добија заменом пара елемената (не обавезно суседних). На пример, у низу пермутација $123, 132, 231, 213, 312, 321$ свака пермутација се од претходне добија једном транспозицијом.
  - замена пара суседних елемената. На пример, у низу пермутација $123, 132, 312, 321, 231, 213$ свака пермутација се од претходне добија једном заменом суседних елемената.

И алгоритми који су засновани на нумерисању објеката се држе лексикографског поретка. Према томе, алгоритми за генерисање комбинаторних објеката могу да се поделе на оне који се држе лексикографског редоследа и оне који се држе правила најмање промене. Оба типа алгоритама имају своје предности, па избор зависи од примене.

Многи проблеми захтевају у оквиру решења потпуну претрагу варијанти. Типични примери таквих проблема су проблем 8 дама, тражење пута кроз лавиринт, избор предмета којима би се попунио ранац датог капацитета. За неке од проблема не зна се алгоритам полиномијалне сложености, па за њихово решавање преостаје неки облик потпуне претраге. Пошто је обично број решења које треба проверити експоненцијална функција од величине улаза, потребно је користити неку стратегију систематичне претраге, да би се повећала ефикасност потпуне претраге. Једна таква стратегија је претрага (са враћањем), поступак који ради са парцијалним решењима проблема. Решење се проширује у веће парцијално решење ако то може да доведе до комплетног решења – та фаза се зове *фаза проширивања*. Ако проширивање текућег решења није могуће или је достигнуто комплетно решење а тражи се и наредно решење, онда се врши повратак на краће парцијално решење и покушава се поново – ова фаза назива се *фазом*

скраћивања. Претрага је обично повезана са лексикографским поретком инстанци. Општи облик ове стратегије претраге може се описати следећим кодом:

```
1 иницијализација
2 repeat
3   if текуће парцијално решење се може проширити then
4     проширивање
5   else
6     скраћивање
7   if текуће решење је прихватљиво then
8     одштампај га
9 until претрага је завршена
```

7.2 Подскупови и партиције

Без смањења општости може се претпоставити да је задатак излистати све подскупове скупа $\{1, 2, \ldots, n\}$. Сваки подскуп може се представити као низ $x_1, x_2, \ldots, x_r$, $1 \leq r \leq n$, $1 \leq x_1 < x_2 < \ldots < x_r \leq n$. Подскуп са тачно m елемената је (m, n)-**подскуп**. На пример, за $n = 5$ лексикографски редослед подскупова је следећи:

```
1   12   123   1234   12345
                1235
          124   1245
          125
     13   134   1345
          135
     14   145
     15
2   23   234   2345
          235
     24   245
     25
3   34   345
     35
4   45
5
```

У фази проширења алгоритам штампа подскупове у истом реду слева удесно. Ако је последњи елемент у подскупу n, алгоритам прелази у нови ред – то је фаза скраћивања.

```
Podskupovi(n)
1 r ← 0; x_r ← 0 {због погодности се користи x_0}
2 repeat
3   if x_r < n then
4     x_{r+1} ← x_r + 1; r ← r + 1 { проширивање }
5   else
6     r ← r − 1; x_r ← x_r + 1 { скраћивање }
7   одштампати x_1, x_2, ... x_r
8 until x_1 = n
```

За генерисање (m, n)-подскупова треба променити инструкцију **if** на следећи начин:

```
if x_r < n and r < m then
  x_r ← x_r + 1; r ← r + 1 { проширивање }
elseif x_r < n then
  x_r ← x_r + 1 { прескок }
else
  r ← r − 1; x_r ← x_r + 1 { скраћивање }
```

Уведена фаза "прескок" се користи када се са једног подскупа прелази на подскуп у неком каснијем реду, прескачући подскупове са више од m елемената. На пример, за $m = 3, n = 5$ резултат су прве три колоне претходне табеле.

Размотримо сада алгоритам за генерисање варијација. Свака (m, n)-**варијација** елемената скупа $\{1, 2, \ldots, n\}$ може се представити низом $z_1, z_2, \ldots, z_m$, где је $1 \leq z_i \leq n$. Наредна варијација која следи после $z_1, z_2, \ldots, z_m$ може се одредити тако што се пронађе највећи индекс t такав да је $z_t < n$, што значи да се z_t може повећати; индекс t је *преломна тачка* (енг. turning point). Вредност z_t повећава се за један, а нове вредности z_i за $i > t$ су 1. На пример, после (4,3) варијације 2133 следи четворка 2211 и овде је преломна тачка 2: најдеснији елеменат различит од 3 је 1, елемент са индексом 2). Алгоритам се може описати следећим кодом:

```
Varijacije(m, n)
1 for i ← 0 to m do z_i ← 1
2 repeat
3   одштампати z_1 z_2 ... z_m
4   t ← m
5   while z_t = n do t ← t − 1
6   z_t ← z_t + 1
7   for i ← t + 1 to m do z_i ← 1
8 until t = 0
```

Подскупови се могу излистати и у облику бинарних низова, при чему свака јединица одговара елементу подскупа. На пример, подскуп $\{1, 3, 4\}$ за $n = 5$ може се представити низом 11010. Према томе, подскупови одговарају целим бројевима записаним бинарно. Једноставни рекурзивни алгоритам

за генерисање (редоследом који је супротан од лексикографског) свих бинарних n-торки може се описати следећим кодом:

Binarni_stringovi(n)
1 **if** $n = 0$ **then** одштампати $c_1, c_2, \ldots, c_n$
2 **else**
3 $\quad c_n \leftarrow 0$; *Binarni_stringovi*$(n-1)$
4 $\quad c_n \leftarrow 1$; *Binarni_stringovi*$(n-1)$

Задати природан број n може се на више начина представити у облику збира природних бројева (својих делова): $n = x_1 + x_2 + \ldots + x_m$. Оваква представа зове се **партиција** ако редослед сабирака није битан. На пример, број 5 има седам партиција:

$$5, 4+1, 3+2, 3+1+1, 2+2+1, 2+1+1+1, 1+1+1+1+1$$

Ако је редослед сабирака битан, онда се представе броја n у облику збира зову **целобројне композиције**. На пример, композиције броја 5 су:

$$5, 4+1, 1+4, 3+2, 2+3, 3+1+1, 1+3+1, 1+1+3, 2+2+1, 2+1+2, 1+2+2,$$

$$2+1+1+1, 1+2+1+1, 1+1+2+1, 1+1+1+2, 1+1+1+1+1$$

Композиције броја n у m делова су представе броја n у облику збира тачно m сабирака. Те композиције се могу представити у облику $n = x_1 + x_2 + \ldots + x_m$, где је $x_1 > 0, \ldots, x_m > 0$. Показаћемо сада каква веза постоји између целобројних композиција и комбинација или подскупова, у зависности да ли је фиксиран број сабирака.

Посматрајмо композицију $n = x_1 + \ldots + x_m$, где је m фиксирано или није фиксирано. Нека је $y_1, \ldots, y_m$ низ са i-тим чланом: $y_i = x_1 + \ldots + x_i$, $1 \leq i \leq m$. Очигледно је $y_m = n$. Низ $y_1, y_2, \ldots, y_{m-1}$ је подскуп скупа $\{1, 2, \ldots, n-1\}$.

- Ако број делова m није фиксиран, онда композиције броја n у произвољан број делова одговарају подскуповима скупа $\{1, 2, \ldots, n-1\}$ и њихов број је 2^{n-1}.
- Ако је број делова m фиксиран, онда су низови $y_1, \ldots, y_{m-1}$ комбинације $m-1$ од $n-1$ елемената из скупа $\{1, 2, \ldots, n-1\}$ и њихов број је $\binom{n-1}{m-1}$. Сваки низ $x_1, \ldots, x_m$ се лако може добити од низа $y_1, \ldots, y_m$, јер је $x_i = y_i - y_{i-1}$ (може се сматрати да је $y_0 = 0$). На пример, $(3,5)$-композицијама $3+1+1, 1+3+1, 1+1+3, 2+2+1, 2+1+2, 1+2+2$ (низовима x_1,x_2,x_3) одговарају низови $3, 4, 5$; $1, 4, 5$; $1, 2, 5$; $2, 4, 5$; $2, 3, 5$; $1, 3, 5$ (низови y_1,y_2,y_3), који сви имају 5 као последњи елемент, а чији префикси дужине два чине свих шест (2,4)-комбинација.

Према томе, за генерисање композиција броја n може се искористити алгоритам за генерисање подскупова или алгоритам за генерисање комбинација.

7.3 Комбинације

Произвољна (m,n)-**комбинација** елемената скупа $\{1,2,\ldots,n\}$ може се представити низом $z_1, z_2, \ldots, z_m$, где је $1 \leq z_1 < z_2 < \ldots < z_m \leq n$. Из ових неједнакости следи да је $z_{m-1} \leq n-1, z_{m-2} \leq n-2, z_{m-3} \leq n-3$, односно уопште $z_i \leq n-m+i$ за $1 \leq i \leq m$. Број (m,n)-комбинација је $\binom{n}{m} = \frac{n!}{m!(n-m)!}$. Размотримо сада алгоритам за генерисање комбинација лексикографским редоследом. Да би се одредила наредна комбинација после комбинације $z_1, z_2, \ldots z_m$ проналази се највећи индекс t такав да је $z_t < n-m+t$, односно највећи индекс t такав да се z_t може повећати; индекс t називамо **преломна тачка**. Пошто се z_t повећа за један, иза њега следе елементи $z_t+1, z_t+2, \ldots$, односно за i-ти елемент, $i \geq t$ је $z_t + i - t + 1$.

На пример, низ (4,6)-комбинација са одговарајућим вредностима t приказана је у следећој табели:

1234	1235	1236	1245	1246	1256	1345	1346	1356	1456	2345	2346	2356	2456	3456
t=4	4	3	4	3	2	4	3	2	1	4	3	2	1	0

Једноставније је преломну тачку одредити на основу претходне преломне тачке. Могућа су два случаја: или је z_t од своје максималне вредности $n-m+t$ мање за бар 2, или је мање за тачно 1.

Случај $z_t < n-m+t-1$ Тада се z_t повећава за 1 и не достиже своју максималну вредност, а елементи који следе иза њега су z_t+1, $z_t+2, \ldots$, и ниједан од њих не достиже своју максималну вредност, па је преломна тачка t за наредну комбинацију једнака m. Такав пример је комбинација 1256 из горње табеле: $t=2$, z_t-3 је за 2 мање од своје максималне вредности 4. У наредној комбинацији 1345 сви елементи су мањи од своје максималне вредности, па је преломна тачка $t=4$.

Случај $z_t = n-m+t-1$

Тада се z_t повећава за 1 и достиже своју максималну вредност, а елементи који следе иза њега су $z_t+1, z_t+2, \ldots$ (они не мењају своје вредности!). Сви они, као и z_t, достигли су своју максималну вредност, па је преломна тачка t за наредну комбинацију једнака $t-1$. Такав пример је комбинација 1356 из горње табеле: $t=2, z_t=3$ је за 1 мање од своје максималне вредности 4, па је наредна комбинација 1456 (елементи после z_t, 4, 5 и 6 нису променили своје вредности!). Нова преломна тачка је за 1 мања од претходне, односно 1.

Према томе, алгоритам се може приказати следећим кодом.

```
Kombinacije(m, n)
1   z_0 ← 1; t ← m
2   for i ← 1 to m do z_i ← i
3   repeat
4     одштампати z_1 z_2 ... z_m
5     z_t ← z_t + 1
6     if z_t = n − m + t then
7       t ← t − 1
8     else
9       for i ← t + 1 to m do
10        z_i ← z_t + i − t
11      t ← m
12  until t = 0
```

Запажамо да се преломна тачка увек одређује после само једне провере.

7.4 Пермутације

Низ $p_1, p_2, \ldots, p_n$ различитих елемената је пермутација скупа $\{1, 2, \ldots, n\}$ ако и само ако је $\{p_1, p_2, \ldots, p_n\} = \{1, 2, \ldots, n\}$. На пример, шест пермутација скупа $\{1, 2, 3\}$ су $123, 132, 213, 231, 312, 321$. На основу општег метода, пермутације се лексикографским редоследом могу генерисати на следећи начин: пермутација која следи иза $x_1, x_2, \ldots x_n$ одређује се тако што се пронађе највећи индекс i такав да је $x_i < x_{i+1}$ (који и у овом случају зовемо преломна тачка; ако такав индекс не постоји, онда наредна пермутација не постоји). Затим се проналази најмањи елемент x_j, $j > i$, такав да је $x_j > x_i$; елементи x_i и x_j замењују места. После тога се инвертује редослед елемената $x_{i+1}, \ldots, x_n$ (који су пре тога чинили опадајући низ). На пример, за пермутацију $3, 9, 4, 8, 7, 6, 5, 2, 1$ преломна тачка је $x_3 = 4$, па се x_3 замењује са $x_7 = 5$ (то је најмањи елемент иза $x_3 = 4$ који је већи од њега) и инвертује се редослед елемената $8, 7, 6, 4, 2, 1$; наредна пермутација је $3, 9, 5, 1, 2, 4, 6, 7, 8$. Алгоритам се може описати наредним кодом:

```
Permutacije(n)
1  for i ← 0 to n do z_i ← i { због удобности се користи z_0 = 0}
2  while i ≠ 0 do
3    одштампати z_1 z_2 ... z_n
4    i ← n − 1
5    while z_i ≥ z_{i+1} do i ← i − 1
6    j ← n
7    while z_i ≥ z_j do j ← j − 1
8    заменити z_i и z_j
9    обрнути редослед елемената z_{i+1}, z_{i+2}, ..., z_n
```

7.5 Задаци за вежбу

1. Претпоставимо да у проблему одабира активности, уместо да увек бирамо активност која се најраније завршава, бирамо активност која

најкасније почиње, а која је компатибилна са претходном одабраним активностима. Доказати да и оваква похлепна стратегија даје оптимално решење.

2. Не даје свака похлепна стратегија у проблему одабира активности оптимално решење. Дати пример да наредне стратегије не дају оптимално решење:

   - приступ којим се бира најкраћа активност која је компатибилна са претходно одабраним активностима,
   - приступ којим се бира активност која се преклапа са најмање преосталих активности,
   - приступ којим се бира активност која прва почиње.

3. Претпоставимо да имамо на располагању скуп активности које треба распоредити унутар великог броја сала за предавање, при чему се свака од активности може одржати у било којој од сала. Желимо да распоредимо све активности коришћењем што мањег броја сала. Дати ефикасни похлепни алгоритам за утврђивање која активност треба да се одржи у којој сали. Овај проблем се назива и *проблем бојења интервалног графа*. Чворови интервалног графа су активности – интервали, а гранама су повезане некомпатибилне активности. Циљ је одредити најмањи могући број боја којим је могуће обојити све чворове графа, тако да никоја два суседна чвора немају исту боју.

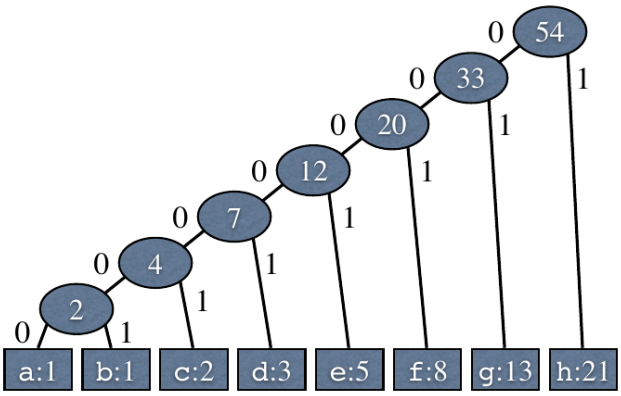
4. Нека је дат скуп активности. Конструисати алгоритам за распоређивање активности којим се минимизује просек времена завршетка свих активности. Свака активност се мора извршавати у континуитету, тј. једном када активност a_i крене са радом, ради непрекидно током времена p_i. Доказати да предложени алгоритам минимизује просечно време завршетка и оценити временску сложеност предложеног алгоритма.

   Размотримо измењену верзију проблема у којој свака од активности може да крене тек након неког момента r_i и притом је могуће неку активност на неко време суспендовати (произвољан број пута) и наставити са њеним радом касније. Конструисати алгоритам за распоређивање активности којим се минимизује просек времена завршетка свих активности у овом сценарију и оценити временску сложеност овог алгоритма.

5. Пун резервоар аутомобила омогућава прелазак X километара. Како прећи дати пут са што мање заустављања ради сипања бензина, ако знамо где се на путу налазе пумпе?

6. Описати ефикасни алгоритам који за дати скуп $\{x_1, x_2, \ldots, x_n\}$ од n тачака на реалној правој, одређује најмањи број затворених интервала јединичне дужине који садрже све тачке.

7. Који је оптимални Хафманов код за наредни скуп фреквенција, које одговарају првим елементима Фибоначијевог низа?



Слика 7.1: Кодно стабло за дате фреквенције

$$a:1,\ b:1,\ c:2,\ d:3,\ e:5,\ f:8,\ g:13,\ h:21$$

Како гласи одговор у општем случају, када су фреквенције првих n Фибоначијевих бројева?

Нека је F_n n-ти члан Фибоначијевог низа, $F_1 = F_2 = 1$. Индукцијом се може показати да важи $\sum_{i=1}^{n} F_i = F_{n+2} - 1$.

База индукције: $F_1 + F_2 = 1 + 1 = 2 = 3 - 1 = F_4 - 1$

Индуктивни корак: претпоставимо да тврђење важи за све позитивне бројеве мање од n. Онда је:

$$\sum_{i=1}^{n} F_i = F_1 + F_2 + \sum_{i=3}^{n}(F_{i-1} + F_{i-2}) = 2 + \sum_{i=3}^{n} F_{i-1} + \sum_{i=3}^{n} F_{i-2}$$

$$= F_{n+1} + F_n - 1 = F_{n+2} - 1$$

Стога ће, приликом конструкције Хафмановог кода, текућа сума фреквенција увек бити мања од свих преосталих листова осим најмањег, те ће најмањи лист увек бити комбинован са текућем сумом. Стога ће кôд за i-то слово бити $10^{i-1}, i = 1, 2, \ldots, n$.

8. Посматрајмо проблем враћања кусура од n динара коришћењем најмањег броја новчаница. Претпоставимо да су расположиве новчанице у вредностима које су степен броја c, тј. $c^0, c^1, \ldots, c^k$ за неке целе бројеве $c > 1$ и $k \geq 1$. Показати да похлепни приступ увек даје оптимално решење.

Похлепни алгоритам за тражење кусура од n динара одговара представљању броја n у систему са основом c: тражи се новчаница c^j тако да важи: $j = \max\{0 \leq i \leq k : c^i \leq n\}$ која је део решења а онда рекурзивно решење за кусур вредности $n - c^j$. Докажимо прво наредну лему:

Лема: За $i = 0, 1, , \ldots, k$ нека је са a_i означен број новчаница вредности c^i које се користе у оптималном решењу. Тада за $i = 0, 1, \ldots, k-1$ важи $a_i < c$.

Доказ: Ако је $a_i \geq c$ за неко $0 \leq i < k$, онда можемо поправити решење коришћењем једне више новчанице од c^{i+1} и c мање новчаница од c^i. На овај начин вредност кусура остаје неизмењена, а смањен је број новчаница за $c - 1 > 0$.

Да бисмо показали да је похлепно решење оптимално, показаћемо да свако друго решење није оптимално. Нека је $j = \max\{0 \leq i \leq k : c^i \leq n\}$; тада похлепно решење користи бар једну новчаницу вредности c^j. Посматрајмо непохлепно решење које не користи новчанице вредности c^j. Нека непохлепно решење користи a_i новчаница вредности c^i за $i = 0, \ldots, j-1$. Тада је $\sum_{i=0}^{j-1} a_i c^i = n$. С обзиром на то да је $n \geq c^j$ важи $\sum_{i=0}^{j-1} a_i c^i \geq c^j$. Претпоставимо сад да је непохлепно решење оптимално. Према горњој леми $a_i \leq c-1$ за $i = 0, \ldots, j-1$. Стога је:

$$\sum_{i=0}^{j-1} a_i c^i \leq \sum_{i=0}^{j-1} (c-1)c^i = (c-1)\sum_{i=0}^{j-1} c^i = (c-1)\frac{c^j - 1}{c-1} = c^j - 1 < c^j$$

што је у контрадикцији са претходном претпоставком. Стога непохлепно решење није оптимално.

9. Нека су дата два скупа A и B од којих сваки садржи n позитивних целих бројева. Дозвољено је преуредити редослед елемената у скуповима на произвољан начин. Након преуређивања нека је a_i i-ти елемент скупа A, а b_i i-ти елемент скупа B. Укупна добит једнака је $\prod_{i=1}^{n} a_i^{b_i}$. Конструисати алгоритам којим се максимизује укупна добит и утврдити његово време извршавања.

10. Конструисати алгоритам тернарне претраге који испитује да ли се нека вредност налази у низу бројева уређених неопадајуће на следећи начин: пореди се вредност која се тражи са елементом низа на позицији $n/3$, уколико је потребно пореди се даље са елементом на позицији $2n/3$ и на овај начин полазни проблем своди на проблем три пута мање димензије. Одредити сложеност овог алгоритма.

```
Ternarna_pretraga(X, n, z)
улаз: X - низ од n бројева уређених неопадајуће, z - број који се тражи
излаз: индекс i такав да је X[i] = z или 0 ако такав индекс не постоји
1 return Nadji_t(z, X, 1, n)
```

```
Nadji_t(z, X, Levi, Desni)
улаз: X - низ од n бројева уређених неопадајуће који се разматра
        у границама од Levi до Desni, z - број који се тражи
излаз: индекс i такав да је X[i] = z или 0 ако такав индекс не постоји
1 if Levi = Desni then
2    if X[Levi] = z then
```

```
3       return Levi
4     else return 0
5   else
6     S_1 ← ⌈Levi + (Desni − Levi)/3⌉
7     S_2 ← ⌈Levi + (Desni − Levi) ∗ 2/3⌉
8     if z < X[S_1] then
9       return Nadji_t(z, X, Levi, S_1 − 1)
10    else if z < X[S_2]
11      return Nadji_t(z, X, S_1, S_2 − 1)
12    else
13      return Nadji_t(z, X, S_2, Desni)
```

Сложеност алгоритма се добија решавањем рекурентне једначине:

$$T(n) = T(n/3) + 2$$

11. Изменити алгоритам бинарне претраге тако да не дели улаз на два дела приближно једнаке величине, већ на два дела чије су приближно величине 1/3 и 2/3 улаза. Колика би била сложеност овог алгоритма?

12. Шта би се десило ако бисмо у алгоритму за истовремено одређивање минимума и максимума изоставили редове 4–10? Да ли би се и даље израчунавала коректна вредност минимума и максимума низа?

13. Нека су X и Y два скупа целих бројева од n елемената, сваки сортиран у неопадајућем редоследу. Конструисати алгоритам сложености $O(\log n)$ којим се проналази медијана скупа $X \cup Y$.

14. Показати како се могу помножити два комплексна броја $a+bi$ и $c+di$ коришћењем само три множења реалних бројева. Алгоритам као улаз прима вредности a, b, c и d, а као излаз треба да дâ реалну компоненту $ac - bd$ и имагинарну компоненту $ad + bc$.

15. Колико брзо се може помножити матрица димензије $kn \times n$ матрицом димензије $n \times kn$, коришћењем Штрасеновог алгоритма као примитиве? Шта ако се редослед матрица измени?

16. Претпоставимо да смо дошли до варијанте Штрасеновог алгоритма заснованог на чињеници да се матрице димензије 3×3 могу помножити коришћењем само m множења уместо уобичајених 27. Колико мало мора да буде m да би нови алгоритам био бржи од Штрасеновог за довољно велико n?

17. Проблем максималне суме суседних елемената низа је проблем у коме за дати низ $A[1..n]$ природних бројева, треба пронаћи вредности i и j, $1 \leq i, j \leq n$ тако да је вредност суме $\sum_{k=i}^{j} A[k]$ максимална могућа. Конструисати алгоритам заснован на разлагању који решава овај проблем у времену $O(n \log n)$.

18. Алгоритам *NDama* се може учинити још ефикаснијим ако се функција $Smesti(k, i)$ преформулише тако да или враћа наредну легитимну

колону у коју је могуће сместити k-ту даму или информацију да то није могуће. Преформулисати обе функције тако да имплементирају ову стратегију.

19. На шаховсој табли димензија $n \times n$ скакач је постављен на произвољно поље са координатама (x, y). Проблем који разматрамо је како одредити $n^2 - 1$ потеза скакача тако да је свако поље табле посећено тачно једном, ако такав низ потеза постоји. Конструисати алгоритам који решава овај проблем.

20. Нека је дат скуп бројева $w = \{5, 7, 10, 12, 15, 18, 20\}$ и вредност $m = 35$. Пронаћи све могуће подскупове скупа w чији је збир једнак m. Нацртати део стабла простора стања који се генерише.

21. За $m = 35$ извршава се алгоритам *SumaPodskupova* на скуповима:

    а) $w = \{5, 7, 10, 12, 15, 18, 20\}$,

    б) $w = \{20, 18, 15, 12, 10, 7, 5\}$ и

    ц) $w = \{15, 7, 20, 5, 18, 10, 12\}$.

    Да ли постоје приметне разлике у броју прегледаних чворова у стаблу простора стања?

22. Написати алгоритам претраге са враћањем за проблем суме подскупова коришћењем стабла простора стања које одговара формулацији са променљивом дужином торке.

23. Навести пример фамилије графова за коју трајање бојења са три боје алгоритмом претраге расте експоненцијално са бројем чворова n, а бојење не постоји.

Глава 8

Тражење речи у тексту

Нека су $S = s_1 s_2 \dots s_n$ и $P = p_1 p_2 \dots p_m$ две ниске знакова из коначне азбуке. За прву од њих рећи ћемо да је **текст**, а за другу да је **реч** (не у обичном смислу — ова реч може да садржи и размаке, односно било које знакове). Подниска ниске S је ниска $S[i..j]$ од узастопних знакова S, $i \leq j$.

Проблем. За дати текст $S = s_1 s_2 \dots s_n$ и реч $P = p_1 p_2 \dots p_m$ установити да ли постоји подниска ниске S једнака P, а ако постоји, пронаћи прву такву подниску, односно најмањи индекс k такав да је $S[k..k+m-1] = P$

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

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

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

У другом случају реч се "помера" за један знак удесно, односно наставља се са провером једнакости p_1 са s_{k+1}. Овај једноставан алгоритам за тражење речи у тексту описује следећи код.

$Nadji\_rec(S, n, P, m)$
улаз: S - текст дужине n и P - реч дужине m
излаз: $Start$ - индекс почетка прве подниске ниске S једнаке P,
ако таква постоји, односно 0 у противном
1 $Start \leftarrow 0$

```
2   i ← 1 {индекс слова у речи}
3   j ← 1 {индекс слова у тексту}
4   while i ≤ m and j ≤ n do  {i, j су индекси у низовима P, односно S}
5     if P[i] = S[j] then
6       i ← i + 1 {слагање: напредовање и у S и у P}
7       j ← j + 1
8     else
9       j ← j − i + 2 {неслагање: P се помера за 1 удесно}
10      i ← 1
11  if i > m then
12    Start ← j − i + 1 {пронађена је подниска текста једнака речи}
13  return Start
```

Број упоређивања знакова мањи је од mn, па је сложеност овог алгоритма у најгорем случају $O(mn)$.

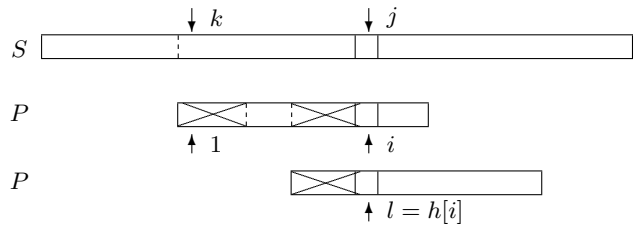
Нерационалност овог алгоритма последица је враћања уназад у тексту после наиласка на неслагање. Претпоставимо да је до неслагања дошло на позицији i у речи, тј. за неки индекс j у тексту је $p_i \neq s_j$ и $P[1..i-1] = S[j-i+1..j-1]$ (видети слику 8.1). Поставља се питање: колико најмање треба реч P померити удесно, односно које последње слово p_l речи треба поставити наспрам слова s_j текста, тако да се део речи $P[1..l-1]$, и даље слаже са делом текста $S[j-l+1..j-1]$? Из претходног слагања речи са текстом следи да је $P[i-l+1..i-1] = S[j-l+1..j-1]$, па $P[1..l-1] = S[j-l+1..j-1]$ повлачи $P[1..l-1] = P[i-l+1..i-1]$. Закључује се да је тражени индекс l за један већи од дужине највећег префикса низа $P[1..i-1]$ једнаког суфиксу низа $P[1..i-1]$, односно l је за један веће од *дужине периода* дела речи $P[1..i-1]$. Индекс l зависи само од P и i. Могуће је дакле најпре обрадити реч, тако да се у посебан низ h на позицију $h[i]$ смести израчунати индекс l, $i = 1, 2, \ldots, m$.

Претпоставимо да ни за једно $0 < l < i$ није испуњен услов $P[1..l-1] = P[i-l+1..i-1]$, па је тај услов испуњен само за $l = 0$:

- ако је $i > 1$, онда је $h[i] = 1 < i$
- ако је $i = 1$, онда је се због $h[i] < i$ (реч се мора померити удесно) мора узети да је $h[i] = 0$, односно у алгоритму КМР се p_1 ставља наспрам s_{j+1}, што одговара стављању p_0 наспрам s_j.

Према томе, под претпоставком да је за дату реч P израчунат низ h, побољшани алгоритам за тражење речи P у тексту S одвија се на следећи начин (видети слику 8.1). Нека је приликом упоређивања P са неком подниском S $p_i \neq s_j$ прво неслагање. Тада се очитава вредност $l = h[i]$ и реч 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 исти начин као да је $p_l \neq s_j$ прво неслагање после низа слагања: очитава се вредност $h[l]$ и реч се помера даље удесно за $l - h[l]$, итд. Уколико је пак $l = 0$, онда се напредује у тексту, тј. p_1 се поставља наспрам s_{j+1}. Описани алгоритам, који је добио име КМР по својим ауторима Кнуту, Морису и Прату (Knuth, Moris, Pratt), може се описати следећим кодом.

$KMP(S, n, P, m)$



Слика 8.1: Идеја на којој се заснива алгоритам КМР.

улаз: S - текст дужине n и P - реч дужине m
{претпоставља се да је низ h израчунат }
излаз: $Start$ - индекс почетка прве подниске ниске S једнаке P,
ако таква постоји, односно 0 у противном

```
1   Start ← 0
2   i ← 1 {индекс слова у речи}
3   j ← 1 {индекс слова у тексту}
4   while i ≤ m and j ≤ n do  {i, j су индекси у низовима P, односно S}
5     while i > 0 and P[i] ≠ S[j] do
6       i ← h[i] {померање речи удесно за i − h[i] }
7     i ← i + 1 {напредовање и у S и у P}
8     j ← j + 1
9   if i > m then
10    Start ← j − i + 1 {пронађена је подниска текста једнака речи}
11  return Start
```

Алгоритам најпре учитава реч и формира табелу h (према алгоритму који ће бити изложен касније), која одређује колико знакова треба померити реч удесно у случају неслагања. Затим се прелази на тражење речи P у тексту S. Пошто нема враћања, улаз се може учитавати знак по знак. У унутрашњој **while** петљи се, после откривања неслагања $p_i \neq s_j$, реч P помера удесно за $i - h[i]$ позиција, тј. индекс слова у речи уместо i добија нову вредност $h[i]$. Овај корак се понавља све док не буде $i = 0$ или $p_i = s_j$ (што значи да је $P[1..i] = S[j-i+1..j]$ за тренутну вредност i). У оба случаја се у спољашњој **while** петљи индекси у речи P и тексту S повећавају за један.

Сложеност. Линија $i \leftarrow h[i]$ у унутрашњој петљи алгоритма КМР (која *смањује* i бар за 1) не може се извршити више пута од линије $j \leftarrow j + 1$; $i \leftarrow i + 1$ која *повећава* i за 1) у спољашњој петљи. Међутим, иста та линија повећава j за 1, и само та линија мења j, па је број њених извршавања $O(n)$. Према томе, у унутрашњој петљи се реч P помера удесно највише $O(n)$ пута, па је временска сложеност овог алгоритма $O(n)$. Приметимо да временска сложеност алгоритма КМР не зависи од величине алфабета.

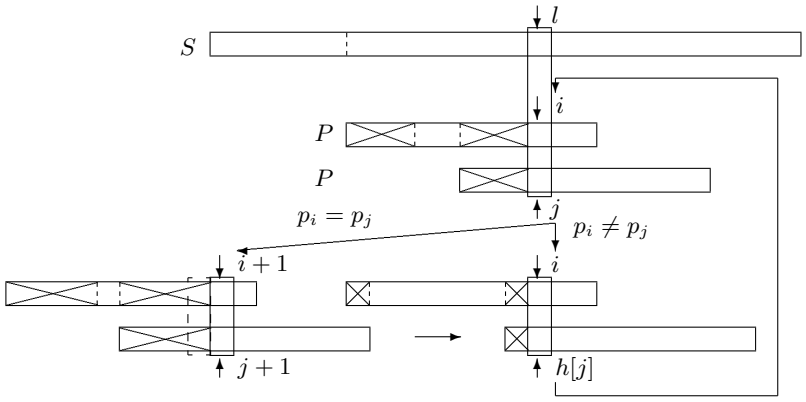
Размотримо сада како се ефективно могу израчунати елементи низа h. Непосредно се види да је $h[1] = 0$ и $h[2] = 1$. Следеће разматрање показује како се може одредити $h[i+1]$ ако се зна $h[1..i]$. Као што је речено, $j = h[i]$ је највећи индекс j за који је $P[1..j-1] = P[i-j+1..i-1]$. Другим речима, j је за 1 веће од дужине периода дела речи $P[1..i-1]$. Потребно

је одредити $h[i+1]$, што је за 1 веће од дужине периода дела речи $P[1..i]$. Ако је $P[i] = P[j]$, онда је дужина периода $P[1..i]$ за један већа од дужине периода $P[1..i-1]$ (јер важи $P[1..j] = P[i-j+1..i]$), па је $h[i+1] = j+1$. У противном, ако је $P[i] \neq P[j]$, онда период $P[1..i]$ може само да продужи неки краћи период $P[1..i-1]$, који мора бити период $P[1..j-1]$, па се $j = h[i]$ мора заменити са $h[j]$, да би се наставило на исти начин (сем ако се добије $j = 0$). Овим је одређен начин израчунавања чланова низа h. Алгоритам *Pomaci* за израчунавање низа h, који је практично идентичан са алгоритмом КМР, описан је следећим кодом.

Pomaci(P, m)
улаз: P - реч дужине m
излаз: h - низ дужине m

```
1   h[1] ← 0 {неслагање на првом знаку у КМР : померање удесно}
2   h[2] ← 1 {неслагање на другом знаку у КМР : померање речи удесно за 1}
3   for i ← 2 to m do  {израчунавање h[i+1]}
4     j ← h[i] {повећање индекса j за 1}
5     { инваријанта петље: на овом месту је P[i-j+1..i-1] = P[1..j-1] }
6     while j > 0 and P[i] ≠ P[j] do
7       j ← h[j] {смањивање индекса j}
8     h[i+1] ← j+1 {дужина периода је продужена за 1}
```

На слици 8.2 је шематски приказан поступак у унутрашњој петљи алгоритма.



Слика 8.2: Унутрашња петља и израчунавање једног елемента табеле h у алгоритму *Pomaci*.

Сложеност. Број извршавања линије $j \leftarrow h[j]$ у унутрашњој петљи, која *смањује* j бар за 1, мањи је или једнак од броја извршавања линије $j \leftarrow h[i]$ ($m-1$ пута) у спољашњој **for** петљи, која за један *повећава* j, јер је j увек позитивно. Према томе, израчунавање h обавља се за време $O(m)$.

Пример. Нека је реч $P = abaaba$. Алгоритам *Pomaci* даје следећи низ h:

i	$P[i]$	$h[i]$	i	$P[i]$	j	$P[j]$	=?	j	$P[j]$	=?
1	a	0								
2	b	1	2	b	1	a	$\neq$	0		
3	a	1	3	a	1	a	$=$			
4	a	2	4	a	2	b	$\neq$	1	a	$=$
5	b	2	5	b	2	b	$=$			
6	a	3								

Применимо алгоритам КМР да бисмо пронашли ову реч у тексту

$$S = abaabcabaaba.$$

После слагања првих 5 знакова, долази се до неслагања на знаку c у S. У том тренутку је $i = j = 6$:

$$\begin{array}{cccccccccccc} & & & & & & j=6 & & & & & \\ & & & & & & \downarrow & & & & & \\ a & b & a & a & b & & c & a & b & a & a & b & a \\ a & b & a & a & b & & a & & & & & & \\ & & & & & & \uparrow & & & & & \\ & & & & & & i=6 & & & & & \end{array}$$

Прва итерација у унутрашњој петљи алгоритма КМР поставља $i = h[6] = 3$, што одговара померању речи за три слова удесно; тиме се $P[3]$ поставља наспрам $S[6]$:

$$\begin{array}{cccccccccccc} & & & & & & j=6 & & & & & \\ & & & & & & \downarrow & & & & & \\ a & b & a & a & b & & c & a & b & a & a & b & a \\ & & & a & b & & a & a & b & a & & & \\ & & & & & & \uparrow & & & & & \\ & & & & & & i=3 & & & & & \end{array}$$

Пошто је и даље $p_i \neq s_j$, i добија нову вредност $h[3] = 1$.

$$\begin{array}{cccccccccccc} & & & & & & j=6 & & & & & \\ & & & & & & \downarrow & & & & & \\ a & b & a & a & b & & c & a & b & a & a & b & a \\ & & & & & & a & b & a & a & b & a & \\ & & & & & & \uparrow & & & & & \\ & & & & & & i=1 & & & & & \end{array}$$

Због поновног неслагања $i = 1$ са замењује са $h[1] = 0$, односно реч P помера се удесно за једну позицију:

$$\begin{array}{cccccccccccc} & & & & & & j=1 & & & & & \\ & & & & & & \downarrow & & & & & \\ a & b & a & a & b & c & a & b & a & a & b & a & \\ & & & & & & a & b & a & a & b & a & \\ & & & & & & \uparrow & & & & & \\ & & & & & & i=1 & & & & & \end{array}$$

На овом месту алгоритам КМР проналази реч у тексту.

Други ефикасан алгоритам за налажење речи у тексту развили су Бојер и Мур 1977. (Boyer, Moore). Приказаћемо га укратко. Разлика између овог и алгоритма КМР је у томе што се реч 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 реч помери удесно. Ако је, на пример, s_m ='Z', а знак 'Z' се не појављује у речи, онда се реч 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 реч помера $m-i$ позиција удесно. Одлука о томе колико реч треба померити удесно постаје компликованија ако је претходно дошло до неколико слагања знакова речи са одговарајућим знацима текста; при томе се узима у обзир и евентуални суфикс пронађеног дела речи једнак префиксу речи, као код алгоритма КМР. 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 речи за m позиција удесно.

8.1 Задаци за вежбу

1. Нека су сви знаци у речи P међусобно различити. Како се може побољшати ефикасност првог алгоритма $Nadji\_rec$ за тражење речи P у тексту S?

2. Израчунати низ h ако је $P = ababbabbababbababbabb$.

3. Уцртати у правоугаону табелу 6×12 сва упоређивања која се извршавају при тражењу речи $P = abaaba$ у тексту $S = abaabcabaaba$, и то првим алгоритмом, односно алгоритмом КМР.

4. Показати да се тражење речи P у тексту S своди на одређивање низа h за реч PS добијену надовезивањем P и S.

Глава 9

Алгоритми нумеричке оптимизације

9.1 Израчунавање вредности полинома

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. Проблем се може решити индуктивним приступом, односно свођењем решења полазног проблема на решење мањег проблема, који се затим решава рекурзивно; претпоставка да се мањи проблем може решити рекурзивно зове се *индуктивна хипотеза*. 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. Испоставља се 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 (појачану) 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 $2n$ множења 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 коефицијент, a_{n-1} је $(n-2)$-и коефицијент, 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. Алгоритам се може описати следећим кодом.

Vrednost_polinoma(a, x)
улаз: $a = a_0, a_1, \ldots, a_n$ - коефицијенти полинома и x - реалан број
излаз: P - вредност полинома у тачки x
1 $P \leftarrow a_n$
2 **for** $i \leftarrow 1$ **to** n **do**
3 $\quad P \leftarrow x \cdot P + a_{n-i}$
4 **return** P

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.

9.2 Лагранжов интерполациони полином

Вредности функција се обично апроксимирају вредностима полинома. Уобичајено је да се за ту сврху користе полиноми најмањег степена који

пролазе кроз све тачке из задатог скупа тачака. Кроз две тачке може се на јединствени начин провући права — полином првог степена. Општије, кроз $n+1$ задатих тачака (x_0, y_0), (x_1, y_1), $\ldots (x_n, y_n)$, може се на јединствени начин провући полином степена n — **Лагранжов интерполациони полином**.

Као и у примеру за $n = 3$, и овде се види да је вредност i-тог производа у овој суми једнака 1 за $x = x_i$, односно 0 ако је $x = x_j, j \neq i$.

Пример Нека је $n = 3$ и нека је потребно пронаћи коефицијенте полинома трећег степена који пролази кроз задате тачке (x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3). Посматрајмо полиноме трећег степена

$$\begin{aligned} A(x) &= \frac{(x-x_1)(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)} \\ B(x) &= \frac{(x-x_0)(x-x_2)(x-x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)} \\ C(x) &= \frac{(x-x_0)(x-x_1)(x-x_3)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)} \\ D(x) &= \frac{(x-x_0)(x-x_1)(x-x_2)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)} \end{aligned}$$

Непосредно се види да су вредности ових полинома у тачкама x_0, x_1, x_2, x_3 следеће:

полином / тачка	A	B	C	D
x_0	1	0	0	0
x_1	0	1	0	0
x_2	0	0	1	0
x_3	0	0	0	1

Због тога је вредност полинома трећег степена

$$P(X) = y_0 A(x) + y_1 B(x) + y_2 C(x) + y_3 D(x)$$

у тачкама редом x_0, x_1, x_2, x_3 једнака y_0, y_1, y_2, y_3, односно $P(x)$ је тражени (јединствени, што нећемо доказивати) полином трећег степена који пролази кроз задате тачке (x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3).

Овај закључак се непосредно уопштава. Јединствени полином степена n који пролази кроз задатих $n+1$ тачака (x_0, y_0), (x_1, y_1),$\ldots (x_n, y_n)$ је **Лагранжов интерполациони полином**,

$$L_n(x) = \sum_{i=0}^{n} y_i \prod_{j \in \{0,1,\ldots,n\} \setminus \{i\}} \frac{x - x_j}{x_i - x_j}.$$

Пример, наставак. Потребно је одредити коефицијенте полинома трећег степена који пролази кроз тачке (x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3) задате следећом табелом.

i	0	1	2	3
x_i	0	1	-1	-2
y_i	-3	3	-13	-33

Тражени полином је

$$\begin{aligned} L_3(x) &= -3\frac{(x-1)(x-(-1))(x-(-2))}{(0-1)(0-(-1))(0-(-2))} \\ &\quad +3\frac{(x-0)(x-(-1))(x-(-2))}{(1-0)(1-(-1))(1-(-2))} \\ &\quad -13\frac{(x-0)(x-1)(x-(-2))}{((-1)-0)((-1)-1)((-1)-(-2))} \\ &\quad -33\frac{(x-0)(x-1)(x-(-1))}{((-2)-0)((-2)-1)((-2)-(-1))} \\ &= x^3-2x^2+7x-3. \end{aligned}$$

9.3 Приближно решавање једначина

Потребно је приближно (са задатом тачношћу) решити једначину $f(x)=0$. При томе се претпоставља да се зна интервал $[a,b]$ такав да на његовим крајевима функција $f(x)$ има вредности различитих знакова, да $f(x)$ има први извод на том интервалу и да је функција $f(x)$ монотона на том интервалу. Под овим претпоставкама $f(x)$ има јединствену нулу $x^*\in[a,b]$, тј. $f(x^*)=0$. Ако је задат број $\epsilon>0$, потребно је одредити приближно решење једначине $f(x)=0$, односно такав број $\hat{x}$ да је $|x^*-\hat{x}|<\epsilon$. Постоји више поступака за приближно решавање једначина које се заснивају на израчунавању вредности функције у низу тачака $x_0=a$, $x_1=b$, x_2, $\ldots$, x_n, са циљем да се пронађе тачка $\hat{x}$ довољно близу нуле x^*.

9.3.1 Метод половљења интервала

Код **метода половљења интервала** вредност функције израчунава се најпре у тачки $x_2=(x_0+x_1)/2$. На крајевима тачно једног од подинтервала $[x_0,x_2]$, $[x_2,x_1]$ функција има вредности различитих знакова, па се са тражењем нуле у том подинтервалу наставља на исти начин као у полазном интервалу $[x_0,x_1]$. Тиме је тачно два пута смањена дужина интервала у коме се налази нула функције. После израчунавања вредности функције у тачки x_n, $n>1$, долази се до интервала дужине $(x_1-x_0)/2^{n-1}$. Ако се за n изабере вредност $\lceil\log_2\frac{x_1-x_0}{\epsilon}\rceil$, и ако се узме да је $\hat{x}$ средина тог интервала, односно $\hat{x}=(x_{n-1}+x_n)/2$, онда је $|\hat{x}-x^*|<(x_1-x_0)/2^n\leq\epsilon$. Алгоритам се може описати следећим кодом.

Metod_polovljenja_intervala(f,x_0,x_1,ϵ)
улаз: f - функција, $x_0<x_1$ такви да је $f(x_0)f(x_1)<0$, ϵ - граница грешке
излаз: $\hat{x}$ - приближна вредност решења, тј. $|\hat{x}-x^*|\leq\epsilon$

```
1   a ← x_0
2   b ← x_1
3   n ← ⌈log_2 (x_1−x_0)/ε⌉
4   for i ← 2 to n do
5       {Инваријанта петље: x_{n−1} ∈ {a,b}, f(a)f(b) ≤ 0, b − a = (x_1 − x_0)/2^{i−2}}
6       a ← min{x_{i−2}, x_{i−1}}
7       b ← max{x_{i−2}, x_{i−1}}
8       x_n ← (a + b)/2
```

```
9      if f(a)f(x_n) < 0 then
10       b ← x_n
11     else
12       a ← x_n
13   return x̂ = (a + b)/2
```

Коректност алгоритма следи из чињенице да је инваријанта петље $x_{n-1} \in \{a, b\}$, $f(a)f(b) \leq 0$, $b - a = (x_1 - x_0)/2^{n-2}$.

Пример. Ако је $f(x) = x^2 - 2$, једначина $f(x) = 0$ има корен на интервалу $[1, 2]$, јер је $f(1) = -1 < 0$ и $f(2) = 2 > 0$. У том интервалу је функција монотоно растућа, јер је њен извод $f'(x) = 2x$ позитиван. Решење ове једначине на 11 децимала је 1.41421356237. У наредној табели приказано је решавање ове једначине методом половљења интервала.

i	x_i	a	b	$(a+b)/2$	$f(a)$	$f(b)$	$f(x_i)$
0	1						
1	2	1	2	1.5	−1	2	0.25
2	1.5000000	1.0000000	1.5000000	1.2500000	−1.0000000	0.2500000	−0.4375000
3	1.2500000	1.2500000	1.5000000	1.3750000	−0.4375000	0.2500000	−0.1093750
4	1.3750000	1.3750000	1.5000000	1.4375000	−0.1093750	0.2500000	0.0664063
5	1.4375000	1.3750000	1.4375000	1.4062500	−0.1093750	0.0664063	−0.0224609
6	1.4062500	1.4062500	1.4375000	1.4218750	−0.0224609	0.0664063	0.0217285
7	1.4218750	1.4062500	1.4218750	1.4140625	−0.0224609	0.0217285	−0.0004272
8	1.4140625	1.4140625	1.4218750	1.4179688	−0.0004272	0.0217285	0.0106354
9	1.4179688	1.4140625	1.4179688	1.4160156	−0.0004272	0.0106354	0.0051003
10	1.4160156	1.4140625	1.4160156	1.4150391	−0.0004272	0.0051003	0.0023355
11	1.4150391	1.4140625	1.4150391	1.4145508	−0.0004272	0.0023355	0.0009539
12	1.4145508	1.4140625	1.4145508	1.4143066	−0.0004272	0.0009539	0.0002633
13	1.4143066	1.4140625	1.4143066	1.4141846	−0.0004272	0.0002633	−0.0000820
14	1.4141846	1.4141846	1.4143066	1.4142456	−0.0000820	0.0002633	0.0000906
15	1.4142456	1.4141846	1.4142456	1.4142151	−0.0000820	0.0000906	0.0000043
16	1.4142151	1.4141846	1.4142151	1.4141998	−0.0000820	0.0000043	−0.0000388
17	1.4141998	1.4141998	1.4142151	1.4142075	−0.0000388	0.0000043	−0.0000173
18	1.4142075	1.4142075	1.4142151	1.4142113	−0.0000173	0.0000043	−0.0000065
19	1.4142113	1.4142113	1.4142151	1.4142132	−0.0000065	0.0000043	−0.0000011
20	1.4142132	1.4142132	1.4142151	1.4142141	−0.0000011	0.0000043	0.0000016

Запажа се да се број тачних цифара резултата ($\sqrt{2} = 1.4142135...$) повећава за око један после свака три израчунавања вредности функције.

9.3.2 Метод лажног положаја, метод сечице

Код **метода лажног положаја** (regula–falsi) претпоставља се да је функција $f(x)$ монотона на интервалу чији су крајеви x_0 и x_F, и да је $f(x_0)f(x_F) < 0$. Због једноставности претпоставићемо да на том интервалу $f(x)$ има први и други извод и да други извод не мења знак на интервалу (тј. функција је на интервалу или конкавна или конвексна). Претпоставимо да је $x_F > x_0$ ако је $f'(x)f''(x) > 0$ на интервалу, односно да је $x_F < x_0$ у противном.

Пример. Нека је, као у претходном примеру, $f(x) = x^2 - 2$. Једначина $f(x) = 0$ има корен на интервалу $[1, 2]$, јер је $f(1) = -1 < 0$ и $f(2) = 2 > 0$. Поред тога, $f'(x) = 2x > 0$ и $f''(x) = 2 > 0$, па се може узети да је $x_0 = 1$ и $x_F = 2$.

Метод лажног положаја на основу тачака x_n и x_F израчунава наредну тачку x_{n+1} као x-координату пресека праве (сечице) кроја пролази кроз тачке $(x_n, f(x_n))$, $(x_F, f(x_F))$ и x-осе. Једначина те праве је

$$\frac{y - f(x_n)}{x - x_n} = \frac{f(x_F) - f(x_n)}{x_F - x_n}$$

x-координата пресека ове праве са x-осом x_{n+1} добија се као вредност за x, ако се стави да је $y = 0$:

$$x_{n+1} = x_n - \frac{f(x_n)}{f(x_F) - f(x_n)}(x_F - x_n), \quad n = 0, 1, \ldots$$

Наставак примера. Нека је $x_F = 2$; тада је $f(x_F) = x_F^2 - 2 = 2$. У наредној табели приказано је решавање ове једначине методом лажног положаја.

i	x_i	$f(x_i)$
0	1	−1
1	1.33333333333333	−0.22222222222222
2	1.40000000000000	−0.04000000000000
3	1.41176470588235	−0.00692041522491
4	1.41379310344828	−0.00118906064209
5	1.41414141414141	−0.00020406081012
6	1.41420118343195	−0.00003501277966
7	1.41421143847487	−0.00000600728684
8	1.41421319796954	−0.00000103068876
9	1.41421349985132	−0.00000017683827
10	1.41421355164605	−0.00000003034065
11	1.41421356053263	−0.00000000520563

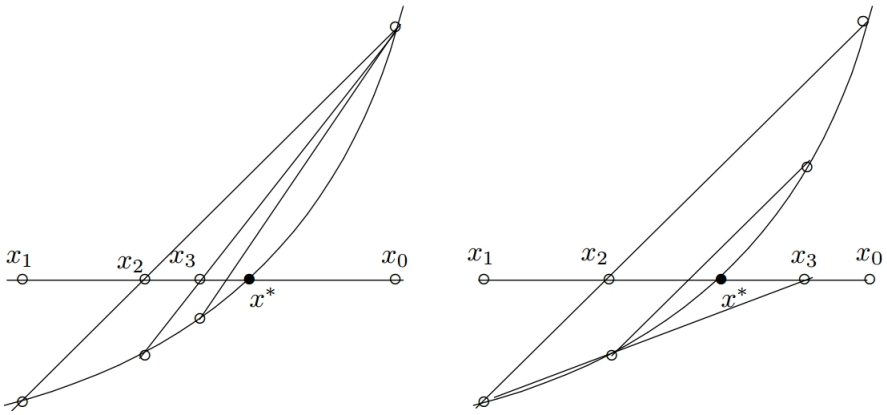
На сличан начин, **метод сечице** на основу тачака x_{n-1} и x_n израчунава наредну тачку x_{n+1} као x-координату пресека праве кроја пролази кроз тачке $(x_{n-1}, f(x_{n-1}))$, $(x_n, f(x_n))$, и x-осе. Једначина те праве је

$$\frac{y - f(x_{n-1})}{x - x_{n-1}} = \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

x-координата пресека ове праве са x-осом x_{n+1} добија се тако што се овде стави да је $y = 0$:

$$x_{n+1} = x_n - \frac{f(x_n)}{f(x_{n-1}) - f(x_n)}(x_{n-1} - x_n), \quad n = 0, 1, \ldots$$

Наредна слика је илустрација ова два метода. Запажа се да овај метод не гарантује да је решење једначине унутар интервала (x_n, x_{n+1}) за свако n.



Наставак примера. Нека је $x_0 = 1$, $x_1 = 2$; тада је $f(x_0) = -1$ и $f(x_1) = 2$. У наредној табели приказано је решавање ове једначине методом сечице.

i	x_i	$f(x_i)$
0	1	−1
1	2	2
2	1.33333333333333	−0.22222222222222
3	1.40000000000000	−0.04000000000000
4	1.41463414634146	0.00118976799524
5	1.41421143847487	−0.00000600728684
6	1.41421356205732	−0.00000000089315
7	1.41421356237310	0.00000000000000
8	1.41421356237310	0.00000000000000

Запажа се да број тачних цифара резултата расте линеарно са бројем израчунавања вредности функције, знатно брже него код метода половљења интервала.

9.3.3 Метод тангенте — Њутнов метод

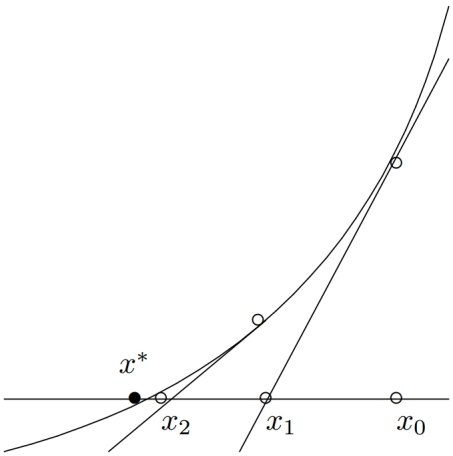
Идеја на којој се заснива **метод тангенте**, односно **Њутнов метод** је да ако је x_n близу x^*, онда је тачка пресека тангенте на криву $y = f(x)$ у тачки x_n још ближе решењу x^*. Једначина тангенте на криву $y = f(x)$ у тачки x_n је

$$y - f(x_n) = f'(x_n)(x - x_n).$$

Стављајући овде $y = 0$ добија се x-координата пресека ове праве са x-осом x_{n+1}:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n = 1, 2, \ldots$$

Наредна слика је илустрација метода тангенте.



Наставак примера. Нека је $x_0 = 1$, $x_1 = 2$; тада је $f(x_0) = -1$ и $f(x_1) = 2$. У наредној табели приказано је решавање ове једначине методом сечице.

i	x_i	$f(x_i)$
0	1	-1
1	1.5000000	0.2500000
2	1.41666666666667	0.00694444444444
3	1.41421568627451	0.00000600730488
4	1.41421356237469	0.00000000000451
5	1.41421356237310	0.00000000000000
6	1.41421356237310	0.00000000000000

Запажа се да се број тачних цифара резултата повећава много брже са бројем израчунавања вредности функције, него код претходних метода (отприлике са квадратом броја израчунавања).

Литература

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, Second Edition, MIT Press, 2002.

[2] М. Живковић, Алгоритми, Математички факултет, 2000.

[3] D. L. Kreher, D. R. Stinson, Combinatorial algorithms: generation, enumeration, and search, CRC press, 1999.

[4] E. Horowitz, S. Sahni, S. Rajasekaran, Computer Algorithms, Computer Science Press, 1997.

[5] I. Parberry, W. Gasarch, Problems on Algorithms, Second Edition, 2002.

[6] I. Stojmenović, Generating All and Random Instances of a Combinatorial Object, Handbook of Applied Algorithms: Solving Scientific, Engineering and Practical Problems, John Wiley & Sons, Inc., 2008.