Univerzitet u Banjoj Luci
Prirodno-matematički fakultet

Elementi teorije algoritama i struktura podataka

Ilija Lalović

Banja Luka, 2014. godine

Autor:
dr Ilija Lalović

Elementi teorije algoritama
i struktura podataka

Recezenti:
dr Nenad Mitić, Matematički fakultet Univerziteta u Beogradu
dr Djura Paunić, Prirodno-matematički fakultet Univerziteta u Novom Sadu

Urednik:
Autor

Tehnički urednik:
Autor

Izdavač:
Prirodno-matematički fakultet Univerziteta u Banjoj Luci

Za izdavača:
dr Rajko Gnjato

Glavni i odgovorni urednik:
dr Biljana Kukavica

Štampa:
Vilux d.o.o. Banja Luka

Za štampatiju:
Boro Luburić, dipl. graf. inž

Tiraž:
100

Predgovor

Knjiga je nastala iz predavanja koja sam, u okviru kursa Algoritmi i strukture podataka, držao studentima infromatike na studijskom programu matematika i informatika, Prirodno-matematičkog fakulteta u Banjoj Luci i studentima na Odsjeku za matematiku, računarstvo i fiziku na Filozofskom fakultetu u Istočnom Sarajevu. U okviru kursa sam nastojao dati uvod u osnovne definicije, rezultate i probleme iz teorije algoritama i struktura podataka. Podrazumijevana predznanja uključuju poznavanje matematike i informatike na nivou uvodnih kurseva prve godine studija.

Pošto u ovom izdanju nisam našao vremena za detaljniju obradu teksta, tekst često liči na koncept lekcija iz kojeg je nastao. Nadam se da ću taj nedostatak otkloniti u sljedećem izdanju. Studenti koji su slušali predavanja primijetiće da knjiga ne obuhvata neke od tema koje sam izlagao. Tako, na primjer, ovaj put nisam stigao prirediti za štampu Računarske metode u algebri, Učenje nad grupama permutacija, Sortiranje na mrežama i na povezanim oknima, Donja ograničenja složenosti algoritama, Algoritme na grafovima, Algoritme računarske geometrije i Algoritme teorije igara. Ova i druga predavanja koja nisu obuhvaćena ovom knjigom planiram publikovati u dodatnoj svesci.

U dodatku, na kraju knjige, dao sam kratke prikaze nekih tema studentskih seminarskih radova. Nadam se da ću tako ublažiti izostavljanje pomenutih sadržaja predavanja.

Zbog nedostatka zbirke zadataka iz teorije algoritama i struktura podataka na našem jeziku, u tekst smo stavili dosta primjera koji su potpuno implementirani u nekom od programskih jezika. Nadamo se da ćemo u sljedećem izdanju programski kod prebaciti u zbirku zadataka, a u knjizi ga zamijeniti potrebnim pseudokodom.

Kolege Dimitriju Čvokić, Darko Drakulić, Vladimir Janjić i Dragan Matić, su zajedno samnom držali pomenuti kurs i u kurs unosili razna poboljšanja. Kolega Drakulić je učestvovao u pisanju glava 7, 8 i 9, u okviru njegovog post-diplomskog studija. Kolege Karlo Bala i Marko Djukanović su pregledali neke od algoritama pretraživanja i priredili programe za rješavanje problema osam kraljica na šahovskoj tabli. Koleginica Ivana Prodanović, je u Latex-u otkucala dio koncepta preda-

vanja. Divni studenti su me učili puno stvari i inspirisali me da im vratim predajući. Ovom prilikom se svima najljepše zahvaljujem.

Posebnu zahvalnost dugujem Prof. dr Nenadu Mitiću i Prof. dr Djuri Pauniću koji su ljubazno pristali da obave težak posao recenziranja knjige i tako pomogli da se knjiga znatno poboljša.

Pokušao sam otkloniti najteže greške i izvinjavam se zbog grešaka koje su preostale u knjizi. Za sve greške je odgovornost isključivo moja. Koristite knjigu uz oprez!

Banja Luka, april 2014. g Ilija Lalović

Sadržaj

Glava 1

Analiza algoritama

U analizi algoritama smo realno zainteresovani samo za *red rasta* složenosti algoritma. Zanima nas kako će se algoritam ponašati u slučaju kada se dužina ulaza povećava i kako matematički evaluirati *funkciju koštanja algoritma* u odnosu na dužinu ulaza i njene elementarne operacije. Pokazuje se da je dovoljno odrediti *asimptotsko ponašanje* funkcije koštanja.

U teoriji algoritama i njihove složenosti govori se o *gornjem ograničenju* i o *donjem ograničenju*. Dato *gornje ograničenje* složenosti algoritma pokušavamo poboljšati, a za *donje ograničenje* pokušavamo dokazati da ne može postojati algoritam sa boljom složenošću.

1.1 Asimptotske oznake

Matematička veliko O notacija je korisna za analizu programa, pošto obuhvata asimptotski rast funkcija i ignoriše konstantne množitelje. Konstantni množitelji su ionako van naše kontrole kada prevedemo algoritme u programe.
Mi koristimo sljedeće definicije, mada postoje i alternativne:
Neka $f, g : \mathbb{N} \to \mathbb{N}$.

- $g(n) = O(f(n))$ akko $(\exists K \in \mathbb{R})$, tako da za dovoljno veliko $n_0 \in \mathbb{N}$ za svako $n \geqslant n_0$, $(n \in \mathbb{N})$, imamo $g(n) \leqslant K \cdot f(n)$

Primjer 1.1. *Za ilustraciju navodimo sljedeće jednostavne asimptotske odnose:*

(a) $(1000n^2 + 100n + 1000) = O(n^3)$

(b) $n^2 = O(n^3 - 100n^2 - 2n)$

(c) $n^i = O(poly(n))$*, gdje je* $deg(p) \geq i$*, a vodeći koeficijent* $p(n)$ *je pozitivan.*

Veliko O je zapravo $Omikron$, ali je dovoljno pisati "O". Relacija $f(n) = O(g(n))$ intuitivno znači da je funkcija $f(n)$ asimptotski manja ili jednaka funkciji $g(n)$. Veliko O daje asimptotsko gornje ograničenje.

Uvodimo i sljedeće definicije

- $f(n) = \Omega(g(n))$ akko $g(n) = O(f(n))$
- $g(n) = \theta(f(n))$ akko ($g(n) = O(f(n))$ i $f(n) = O(g(n))$
- $f(n) = o(g(n))$ akko $\frac{f(n)}{g(n)} \to 0,\ n \to \infty$
- $f(n) = \omega(g(n))$ akko $\frac{f(n)}{g(n)} \to \infty,\ n \to \infty$

Intuitivno $f(n) = \Omega(g(n))$ znači da je f asimptotski veća ili jednaka g. Veliko Ω daje donje ograničenje.

Intuitivno $g(n) = \theta(f(n))$ znači da je f asimptotski jednako g. Dakle, f je ograničeno odozgo i odozdo sa g. Veliko θ daje asimptotsku ekvivalentnost.

Zadatak 1.1. *Razmotriti sljedeće funkcije:*

$\sqrt{n},\ n,\ 2n,\ n\log n,\ n - n^3 + 7n^5,\ n^2 + \log n,\ n^2, n^3,$
$\log n,\ n^{1/3} + \log n,\ (\log n)^2,\ n!,\ \ln n,$
$\frac{n}{\log n},\ \log\log n,\ (1/3)^n,\ (3/2)^n.$

Grupisati ove funkcije tako da su $f(n)$ i $g(n)$ u istoj grupi akko $g(n) = O(f(n))$ i ispisati grupe u rastucećem poretku.

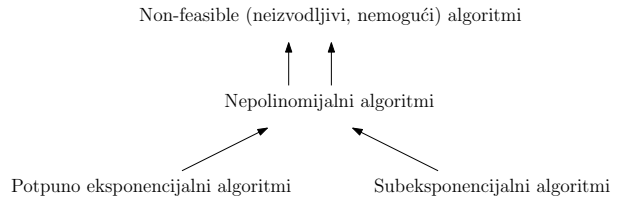
1.2 Hijerarhije složenosti algoritama

Prema Čerčovoj (A. Church) tezi, sve što je intuitivno izračunljivo, izračunljivo je i algoritamski. Postoji više ekvivalentnih formalnih definicija pojma algoritma, kao što su λ-računi, Turingove mašine, parcijalno rekurzivne funkcije, normalni algoritmi Markova. Programski jezici u kojima može biti implementirana Turingova (A. Turing) mašina, nazivaju se Turing kompletni jezici. Uobičajeni programski jezici, na primjer $Fortran$, $Algol\,68$, $Pascal$, C, $C++$, $Java$, $\#C$ su Turing kompletni. Iz Turing kompletnosti programskih jezika proizilaze različite njihove osobine. U Dodatku A, na primjeru jezika C i $C++$, naveli smo interesantnu osobinu da u Turing kompletnim jezicima možemo napisati program koji kao rezultat generiše vlastiti kod.

Vremenska složenost(kompleksnost) algoritma se odredjuje kao asimptotska složenost funkcije koja predstavlja "broj koraka" (jedinica vremena) algoritma u odnosu na dužinu ulaza (ne u odnosu na partikularni ulaz).
Prostorna složenost algoritma odredjuje se kao asimptotska složenost funkcije koja predstavlja prostorne resurse (memoriju) potrebnu algoritmu u odnosu na dužinu ulaza.

Razlikujemo worst case (najgori slučaj), best case (najbolji slučaj) i average case (statistički slučaj) složenosti algoritma. Ako prodajemo algoritam, onda saopštavamo najgori slučaj i garantujemo da će algoritam, u zavisnosti od ulaza, raditi onoliko vremena koliko je predvidjeno u najgorem slučaju, ili bolje od toga. Ako prodajemo algoritam i karakterišemo ga najboljim slučajem, onda to predstavlja varanje kupca, jer će algoritam samo za specijalne ulaze raditi onoliko vremena koliko je predvidjeno najboljim slučajem, a u ostalim slučajevima će trebati više vremena.

Prema složenosti algoritmi mogu biti polinomijalni i nepolinomijalni. Medju nepolinomijalnim algoritmima izdvajamo klase potpuno eksponencijalnih i subeksponencijalnih algoritama (slika 1.1). Algoritam čija je složenost zadata eksponencijalnom funkcijom (npr. $T(n) = O(2^n)$), je eksponencijalni algoritam. Algoritam čija je složenost zadata subeksponencijalnom funkcijom (npr. $T(n) = O(n^{\log\log\log n})$), je subeksponencijalni algoritam.



Slika 1.1: Dio hijerarhije neizvodljivih algoritama

Na slici 1.2 predstavljena je šema klasifikacije problema.

Vidimo da postoje i problemi za koje je dokazano da ne mogu biti riješeni algoritamski. Takav je na primjer *halting problem*, koji kaže da ne postoji algoritam koji na ulazu dobiva program P i ulaz In u program P, a na izlazu daje odgovor DA, ako P završava rad sa ulazom In, a odgovor NE, ako P ne završava rad sa ulazom In. Ako je problem algoritamski nerješiv, to ne znači da neki njegovi slučajevi ne mogu biti riješeni pogadjanjem. Klasa problema teških za rješavanje obuhvata nepolinomijalne algoritme. Klasa problema lakih za rješavanje, klasa P (od engleske riječi "polynomial") obuhvata probleme koji imaju polinomijalnu složenost. Nazivamo ih i izvodljivi(eng. Feasible) algoritmi.

Klasa NP (NP su početna slova engleske fraze "nondeterministic polynomial") obuhvata sve P-probleme, a takodje i probleme sa manje izvjesnim statusom: poznati su samo algoritmi eksponencijalne složenosti, ali nije dokazano da za njih ne postoje algoritmi polinomijalne složenosti. Klasu NP čini skup problema koji mogu biti riješeni pomoću nedeterminističkih algoritama polinomijalne složenosti. Pod **nedeterminističkim algoritmom** ovdje podrazumijevamo algoritme koji se sastoje od dvije etape: etape pogadjanja i etape verifikacije. U prvoj etapi jednostavno pogadjamo rješenje, a verifikacija se zasniva na determinističkoj provjeri, pomoću algoritma polinomijalne složenosti, da li to rješenje zadovoljava uslove problema.

Detaljnije razmatranje pokazuje da rješavanju problema nedeterminističkim al-

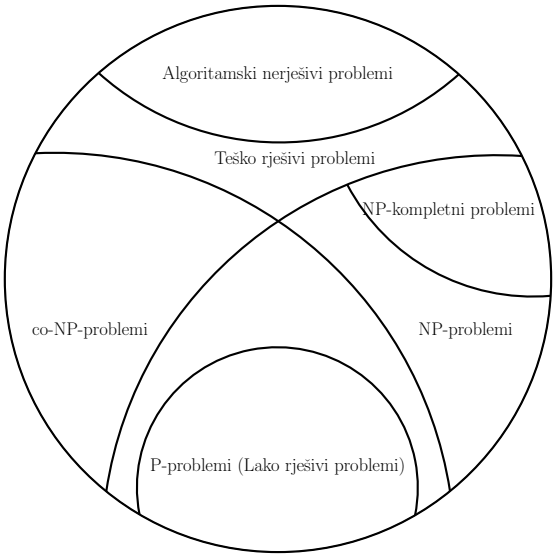
goritmima nedostaje odredjena simetrija u odnosu na rješavanje problema determinističkim algoritmima. Ako se problem X rješava pomoću polinomijalnog algoritma, tada se i problem $\neg X$ rješava pomoću polinomijalnog algoritma. Stvari stoje drugačije kada je u pitanju nedeterministički algoritam. U tom slučaju pomenuta simetrija ne postoji uvijek. Na slici 1.2 postoji klasa $co\text{-}NP$-problem, komplementarnih problemima tipa NP. Pri tome presjek klase NP sa klasom $co\text{-}NP$ nije prazan. Na primjer, *problem složenih brojeva*, u kojem pitamo je li prirodan broj složen, pripada objema klasama.

Moguće je da za rješavanje problema tipa NP postoje deterministički polinomojalni algoritmi, ali do danas problem "je li $P = NP$" nije riješen, iako usvajamo $P \neq NP$.

Ako bi jedan problem iz NP mogli riješiti deterministčkim polinomijalnim algoritmom, to bi i sve ostale probleme iz NP bilo moguće riješiti koristeći takve algoritme.

Postoji i tijesna povezanost izmedju NP-kompletnih problema i hipotezom $NP \neq co\text{-}NP$. Ako bi postojao NP-kompletan problem takav da bi njegova komplementarna verzija bila tipa NP, tada bi klase NP i $co\text{-}NP$ predstavljale istu klasu problema.

Ovdje nećemo razmatrati artimetičke i analitičke hijerarhije složenosti.



Slika 1.2: Klasifikacija problema uz pretpostavku $NP \neq P$

1.2.1 Poredjenje algoritama polinomijalne, logaritamske i eksponencijalne složenosti

Složenost algoritama zavisi od modela komputacije. Dimenzija problema je mjera ulaznih podataka. Npr. dimenzija problema množenja matrica može biti najveća dimenzija matrice faktora. Kada govorimo o vremenskoj složenosti mislimo na asimptotsku vremensku složenost, a kada govorimo o prostornoj složenosti mislimo na asimptotsku prostornu složenost. U sljedećoj tabeli dajemo poredjenje algoritama različite složenosti prema maksimalnoj dimenziji problema koji mogu riješiti u nekim vremenskim intervalima.

Algoritam	Vremenska složenost	Maksimalna dimenzija problema		
		1s	1min	1h
A1	n	10^6	$60 \cdot 10^6$	$3.6 \cdot 10^9$
A2	$n \log_2 n$	62 746	2 801 417	133 378 058
A3	n^2	1 000	7 745	60 000
A4	n^3	100	391	1 532
A5	2^n	19	25	30
A6	3^n	12	16	20
A7	10^n	6	7	9

Tabela 1.1: Poredjenje algoritama prema maksimalnoj dimenziji problema koji mogu riješiti u datom vremenskom intervalu

Algoritmi $A1 - A4$ su polinomijalni, a algoritmi $A5 - A7$ su eksponencijalni. Iz tabele se vidi prednost polinomijalnih algoritama.

U sljedećoj tabeli dajemo poredjenje vremena izvodjenja algoritama sa različitim vremenskim složenostima, na problemu iste dimenzije.

Složenost	Dimenzija problema n			
	10	20	30	40
n	$10 \cdot 10^{-6}$s	$20 \cdot 10^{-6}$s	$30 \cdot 10^{-6}$s	$40 \cdot 10^{-6}$s
$n \log_2 n$	$33,2 \cdot 10^{-6}$s	$86,4 \cdot 10^{-6}$s	$147,2 \cdot 10^{-6}$s	$21,3 \cdot 10^{-6}$s
n^2	$0,1 \cdot 10^{-3}$s	$0,4 \cdot 10^{-3}$s	$0,9 \cdot 10^{-3}$s	$1,6 \cdot 10^{-3}$s
n^3	$1 \cdot 10^{-3}$s	$8 \cdot 10^{-3}$s	$27 \cdot 10^{-3}$s	$64 \cdot 10^{-3}$s
2^n	0,001 s	1 s	17,9 min	12,7 dana
3^n	0,059 s	58,1 m	6,53 g	3 85500g
10^n	2,8 h	3171000 g	$3,17 \cdot 10^{16}$ g	$3,17 \cdot 10^{26}$ g

Tabela 1.2: Poredjenje vremena izvodjenja algoritama na raznim dimenzijama problema (s - sec, m - min, h - sati, g - godine)

Zaključujemo da povećanjem dimenzije problema eksponencijalni algoritmi mogu postati neupotrebljivi. Sa obzirom na podatak da se starost planete Zemlje ocjenjuje na $4,6 \cdot 10^9$ godina, posebno su interesantni rezultati u donjem dijelu tabele 1.2.

1.2.2 Uticaj povećanja brzine procesora na brzinu algoritma

Nakon analize rezultata prikazanih u tabelama 1.1 i 1.2 ostaje mogućnost da povećanje brzine procesora može popraviti kvantitativnu ocjenu složenosti eksponencijalnih algoritama u odnosu na složenost polinomijalnih algoritama. Ipak, kao što je pokazano u tabeli 1.3, algoritmi eksponencijalne složenosti ne mogu na taj način asimptotski doći u prednost nad algoritmima polinomijalne složenosti.

	Dimenzija max problema rješivog za 1 sat na računaru			
Složenost	Aktuelnom	10 × bržim	100 × bržim	1000 × bržim
n	n_1	$10n_1$	$100n_1$	$1000n_1$
$n \log_2 n$	n_2	$10n_2\, za\, veliko\, n_2$	$25n_2$	$140n_2$
n^2	n_3	$3,16n_3$	$10n_3$	$31,62n_3$
n^3	n_4	$2,15n_4$	$4,63n_4$	$10n_4$
2^n	n_5	$n_5 + 3,32$	$n_5 + 6,64$	$n_5 + 9,97$
3^n	n_6	$n_6 + 2,1$	$n_6 + 4,19$	$n_6 + 6,29$
10^n	n_7	$n_7 + 1$	$n_7 + 2$	$n_7 + 3$

Tabela 1.3: Poboljšanje brzine algoritma nakon povećanja brzine procesora

Pitanje: Može li eksponencijalni algoritam biti bolji od polinomijalnog?
Odgovor: Iz kvantitativnih ocjena u tabelama slijedi da asimptotski to nije moguće. Ali, ako su problemi koje rješavamo, ograničene dimenzije, onda je to moguće. Za ilustraciju razmotrimo algoritme date tabelom 1.4.

Algoritam	A1	A2	A3	A4	A5
Složenost	$100n$	$100n \log n$	$10n^2$	n^3	2^n

Tabela 1.4: Neki polinomijalni i eksponencijalni algoritmi na problemima ograničene dimenzije

Nije teško izračunati da:
A5 je najbolji za $2 \leqslant n \leqslant 9$;
A3 je najbolji za $10 \leqslant n \leqslant 58$;
A2 je najbolji za $59 \leqslant n \leqslant 1024$;
A1 je najbolji za $n > 1024$.

1.3 Najgori slučaj (Worst case), najbolji slučaj (best case) i srednji slučaj (average case) složenosti algoritama

Mi se najčešće koncentrišemo na nalaženje worst case (najgori slučaj) vremena izvodjenja algoritama. Za ovakav pristup imamo najmanje tri razloga:

(a) Worst case vrijeme izvodjenja je gornja granica vremena izvodjenja za svaki ulaz. Znanje te granice garantuje da algoritam neće nikada raditi duže, što nam je u praksi važno.

(b) Neki algoritmi često rade u worst case vremenu. Npr. kod traženja u bazi podataka neke partikularne informacije, algoritam pretraživanja će raditi u worst case vremenu, ako ta informacija ne postoji u bazi. Pri tome je traženje nepostojeće informacije čest slučaj.

(c) Average case (statistička složenost, matematičko očekivanje) je često loše kao i worst case. Npr. kod sortiranja umetanjem (insert sort) average case je $O(n^2)$, kao i worst case. Kod nekih drugih algoritama, na primjer Quick Sort-a to nije slučaj. Average case složenost ili očekivano vrijeme za jedan algoritam je $T(n) = \sum_{i=1}^{n} p_i T(i)$, gdje je p_i vjerovatnoća složenosti $T(i)$.

Primjer 1.2. *Razmotrimo tri vrste složenosti na primjeru insert sort-a skupa od n elemenata.*

1. *Worst case složenost se dešava ako elemenat koji umećemo, na svakom koraku moramo porediti sa svim do tada umetnutim elementima, da bi došao na ispravno mjesto. To znači da imamo nula, pa jedno, pa dva poredjenja i na kraju $n-1$ poredjenja. Odatle slijedi da je*

$$T(n) = 0 + 1 + ... + (n-1) = O(n^2).$$

2. *Best case složenost. Ako u svakom koraku element koji umećemo dolazi na svoje mjesto jednim poredjenjem, onda imamo*

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

3. *Da nadjemo average case složenost dokazaćemo lemu o inverzijama liste.*

Lema 1.1. *Prosječan broj inverzija u jednom nizu od n elemenata je $\frac{n(n-1)}{4}$.*

Dokaz. Posmatrajmo listu L i njen reverz L_r. Na primjer, $L = (8, 34, 64, 51, 32, 21)$, $L_r = (21, 32, 51, 64, 34, 8)$. Elementi (x, y) prave inverziju ako je $y > x$. Tačno u jednoj L ili L_r je (x, y) inverzija, a odatle slijedi da je broj inverzija $I = \frac{n(n-1)}{4}$. □

Teorema 1.1. *Average case složenost za insert sort je $T(n) = O(n^2)$.*

Dokaz. Svaki korak u sortiranju eliminiše samo jednu inverziju. □

1.4 Dizajn i analiza složenosti nekih karakterističnih algoritama

Opisaćemo nekoliko elementarnih algoritama i pokazati nalaženje vremenske složenosti tih algoritama.

1.4.1 Racionalizacija množenja velikog broja kompleksnih brojeva

Množenje kompleksnih brojeva (a u odnosu na polje realnih brojeva) može se posmatrati kao izračunavanje izraza $ac-bd$ i $ad+bc$ tako da je $(a,b)\cdot(c,d) = (ac-bd, ad+bc)$. U ovom slučaju imamo četiri množenja i dva sabiranja/oduzimanja $(+/-)$. Složenost algoritama je $T(n) = O(n^2)$. Do bržeg algoritma dolazimo ako za kompleksne brojeve (a,b) i (c,d) izračunamo:

1. $(a+b)\cdot(c+d) = ac+ad+bc+bd$;
2. $a\cdot c$;
3. $b\cdot d$.

Proizvod dobijamo po šemi $(2-3, 1-2-3)$. Dakle, kada od druge veze oduzmemo treću, dobivamo član $ac-bd$, a kada od prve veze oduzmemo drugu i treću vezu, dobivamo član $ad+bc$. Ukupno imamo tri množenja i pet sabiranja/oduzimanja. Može se pokazati da je dobijeni algoritam brži. Takodje se može dokazati da su tri množenja neophodna i dovoljna. Intuitivno, pošto je u većini domena računanja množenje skuplja operacija od sabiranja/oduzimanja, smanjenjem broja množenja povećavamo brzinu algoritma.

1.4.2 Hornerova šema. Kompleksnost izračunavanja vrijednosti polinoma

Posmatramo izračunavanje vrijednosti polinoma

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$$

u tački X. Ulaz se zadaje koeficijentima, stepenom polinoma i tačkom u kojoj se traži vrijednost, respektivno:

$$a_0,\ a_1,\ a_2, ..., a_{n-1},\ a_n,\ X.$$

Na izlazu treba dati vrijednost polinoma $p(X)$.
Pravilo Hornera. Prema pravilu Hornera polinome možemo predstaviti u sljedećem obliku:

1. $a_1 x + a_0$, za $n = 1$
2. $(a_2 x + a_1)x + a_0$, za $n = 2$

3. $((a_3x + a_2)x + a_1)x + a_0$, za $n = 3$, itd.

Uopšte

$$a_0 + x[a_1 + x[a_2 + \ldots + a_n x] \ldots]$$

Za izračunavanje $p(x)$ po Hornerovoj šemi, trebamo n operacija množenja i n operacija sabiranja. Može se pokazati da u modelu sekvencijalnih programa (to jest, programa bez petlji) Hornerova šema daje optimalan algoritam (u odnosu na broj operacija sabiranja i mnozenja) za izračunavanje vrijednosti polinoma u datoj tački X. Koristeći preprocesiranje koeficijenata može se dobiti algoritam koji evaluira polinome bolje nego Hornerova šema.

Dokazi optimalnosti broja operacija za množenje kompleksnih brojeva i za evaluaciju polinoma po Hornerovoj šemi zahtijevaju detaljnu analizu računanja nad poljima brojeva. Množenje kompleksnih brojeva i evaluacija polinoma obično se predstave preko množenja matrica, pa se onda metodama linearne algebre izvode donja ograničenja za broj potrebnih operacija. Interesantno je da je za množenje matrica nad nekomutativnim prstenom potrebno manje operacija nego za množenje nad komutativnim prstenom ili poljem.

1.4.3 Elementarne metode sortiranja: bubble sort (sortiranje mjehurićima), selection sort (sortiranje izabiranjem) i insert sort (sortiranje umetanjem)

Neka je niz koji treba sortirati sastavljen od $a[i] \in S,\ i = 1, 2, ..., n$, pri čemu je S linearno uredjen skup. Sortiramo u rastućem poretku.

Bubble sort (sortiranje mjehurićima)

Poredimo

$$a[j] \text{ sa } a[j+1], j = 1, 2, ..., n-1, i = 1, 2, ..., n-1.$$

Ukoliko je

$$a[j] > a[j+1]$$

tada elementima razmejnjujemo mjesta. Na taj način će u prvom prolazu najveći element doći na zadnje mjesto. U drugom prolazu će drugi po veličini element doći na pretposljednje mjesto itd. U i-tom prolazu dovodimo i-ti element na odgovarajuće mjesto. Nakon nekog broja prolaza, najviše $n - 1$, svi će elementi biti sortirani. Napominjemo da uvodjenjem indikatora(zastavice) možemo uciniti da algoritam prekine rad čim su brojevi sortirani. Naime, čim u nekom prolazu nema izmjena postavljamo indikator na 1, a onda izlazimo iz petlje, jer je provjera indikatora postavljena u uslovu petlje.

Implementaciju sortiranja mjehurićima dajemo u sljedećem $C++$ programu.

```
template <classT>
void sort(T*a, int n)
{
for(int i=1;i<n;i++)
    for(int j=1;j<=n-i;j++)
        if(a[j-1]<a[j]) swap(a[j-1],a[j]);
//invarijanta: i-ti po velicini elementi
//su na ispravnom mjestu
}
```

Worst case složenost:

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

je broj poredjenja koje trebe napraviti u najgorem slučaju. Tu je $n-1$ broj poredjenja da se u najgorem slučaju najveći elemenat dovede na zadnje mjesto, $n-2$ broj poredjenja da se drugi po veličini elemenat dovede na predzadnje mjesto, i tako dalje. Ako uzmemo da se jedno poredjenje izvodi u jedinici vremena, onda govorimo o vremenskoj složenosti algoritma.

Insertion sort (sortiranje umetanjem)

Na prvo mjesto dovodimo proizvoljan element niza $a[]$. Iz ostatka niza uzimamo proizvoljno elemenat i dovodimo ga na prvo mejsto niza ako je manji od elementa koji tamo već stoji, ili na drugo mjesto, u suprotnom. Zatim iz ostatka niza uzimamo neki element i poredimo ga sa drugim i prvim elementom, i umetnemo ga na mjesto tako da prva tri elementa budu uredjeni u rastućem poretku, i tako dalje, dok ne uredimo svih n elemenata niza. Postupak je sličan slaganju karata u igri remi.

U sljedećem programu dajemo implementaciju sorta umetanjem u $C{+}{+}$ jeziku.

```
template <classT>
void sort( T *a, int n)
{
for(int i=1;i<n;i++)
{
    T temp=a[i];
    for(int j=i;j>0 && a[j-i]>temp;j--)
        a[j]=a[j-1];
        a[j]=temp;
//Invarijanta: a[0]<=a[1] <=...<=a[i]
}
}
```

Worst case kompleksnost je

$$T(n) = 1 + 2 + ... + (n-1) = O(n^2),$$

jer za dvočlani niz imamo jedno poredjenje, za tročlani dva poredjenja... i za n-ti član, završni niz, imamo $n-1$ poredjenja. Uopšte, i-ti element se dovodi na pravo mjesto u i-tom nizu, koristeći u najgorem slučaju $(i-1)$ poredjenja.

Selection sort (sortiranje izabiranjem)

U prvom krugu se na prvo mjesto niza $a[\,]$ dovodi najmanji element niza. U drugom krugu se na drugo mjesto dovodi najmanji element iz ostatka niza, itd.

Sljedeći program predstavlja implementaciju sorta izborom u $C++$ jeziku.

```
template <classT>
void sort( T *a, int n)
{for(int i=0;i<n-1;i++)
    {int min=i
                for(int j=i+1;j<n;j++)
                   if(a[j]<a[min])min =j;
//Invarijanta: a[min]<=a[j] za i<=j<n
swap(a[min], a[i]);
       }
}
```

Worst case složenost se dobiva po obrascu

$$T(n) = (n-1) + (n-2) + ... + 1 = O(n^2),$$

gdje je $(n-1)$ broj poredjenja da se od n brojeva izabere najmanji, $(n-2)$ broj poredjenja da se iz $n-1$ brojeva izabere najmanji, $\ldots$, 1 broj poredjenja da se u nizu od dva preostala elementa izabere manji.

1.4.4 Algoritam za nalaženje drugog elementa u nizu i razvoj algoritama efektivnog sortiranja

Dajemo primjer za nalaženje prvog i drugog po veličini elementa niza u $n + \log_2 n - const$ koraka. Sa slike 1.3 vidimo da prvi element biramo u $n-1$ koraka (poredjenja), jer u svakom čvoru drveta imamo rezultat nekog poredjenja, pa je broj potrebnih poredjenja jednak broju čvorova drveta.

Kandidati za drugi po veličini element su oni koji su izgubili od najvećeg kandidata. Na svakom nivou drveta je po jedan takav element, pa je zbog toga njihov broj jednak visini drveta, dakle $\log_2 n$. Na slici 1.3 smo takve brojeve označili pravougaonikom.

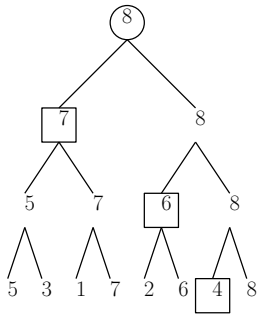
Dakle, prvi i drugi element niza možemo izabrati u $(n-1)+(\log_2 n-1)$ koraka.

Na osnovu opisanog postupka pokušaćemo razviti ideju za sortiranje niza u vremenu koje je bolje od $O(n^2)$, u najgorem slučaju. Nadjemo najveći element niza i stavimo ga prvo mjesto u novom nizu. Nadjemo drugi po veličini element u nizu i stavimo ga na drugo mjesto, itd. U najgorem slučaju, ovaj algoritam ima složenost

$$T(n) = n + (n-1) + ... + 1 = O(n^2).$$

Medjutim, ako algoritam poboljšamo tako da na svakom koraku pamtimo ko je sve izgubio od prethodno izabranih elementa, kao u gore opisanoj metodi za izbor drugog po veličini elementa, dobićemo složenost

$$T(n) = (n-1) + \log_2 n + \ldots + \log_2 n = O(n \log n)$$



Slika 1.3: Pronalaženje prvog i drugog po veličini elementa niza

u najgorem slučaju.

U ovom algoritmu smo poboljšali vrijeme izvodjenja (smanjili broj poredjenja elemenata) na račun memorije za pamćenje elemenata.

1.5 Metoda programiranja podijeli i vladaj. Dizajn i analiza složenosti rekurzivnih algoritama

Rekurzivno programiranje zasniva se na mogućnosti da metoda može pozvati samu sebe za manje vrijednosti argumenata. Da nebi došlo do protivrječnosti, pri dizajniranju rekurzivnog programa moraju se zadovoljiti tri osnovna pravila.

- Rekurzija ima *osnovni slučaj,* koji uključujemo da bi program mogao završiti izvodjenje.
- Rekurzivni pozivi pozivaju podproblem, koji u nekom smislu mora biti *manji* od problema u metodi koja poziva, tako da rekurzivni pozivi konvergiraju ka *osnovnom slučaju*.
- Pozvani podproblemi se ne smiju preklapati.

Prije ocjenjivanja složenosti rekurzivnih algoritama vodimo računa o nekoliko koraka, koje navodimo i komentarišemo u sljedećem spisku. Ovi koraci nisu nezavisni i ne izvode se linearno, već se sa svakog koraka možemo vratiti na neki od prethodnih koraka u cilju poboljšanja algoritma.

1. Definisati model komputacije. Ako računamo u formalizmima kao što su rekurzivne funkcije, mašine Turinga, $\lambda-$računi, algoritmi Markova ili RAM (Random Access Machine), a pri tome radimo nad istim domenom, tada će se ocjene uglavnom razlikovati u konstantama, što se ne odražava na formule složenosti u O simbolici. Ako model računanja biramo izmedju determinističkih, nedeterminnističkih, alternirajućih, paralelnih, stohastičkih ili

neuniformnih modela, onda se ocjene mogu kvantitativno razlikovati. Takodje, nad različitim domenima, na primjer, polje, komutativni prsten, nekomutativni prsten, možemo dobiti kvantitativno različite ocjene složenosti.

2. Definisati formate podataka ulaza i izlaza. Formati podataka su uslovljeni domenom računanja.

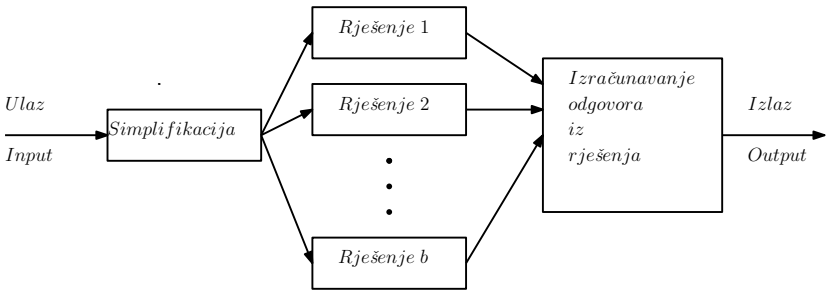
3. Definisati COST funkciju. Koriste se analitičke tehnike koje su identične tehnikama za analizu vremena izvodjenja programa. U oba slučaja brojimo koliko će puta biti izvedene neke osnovne operacije. Osnovne operacije izdvajamo nakon što napravimo *cost model*. *Cost model* daje mogućnost da smanjimo vremensku složenost na račun prostorne složenosti i obrnuto.

4. Pridružiti dužine ulazima

Funkcija složenosti problema se odnosi na dužinu inputa i pri tome ne na partikularan input. Pokušavamo naći gornju i donju granicu problema. Ako problem ima gornju granicu $O(f(n))$ to znači da postoji algoritam A za problem i $O(f(n))$ je složenost od A.
Intuitivno, algoritam je jedan potpuno specifikovan niz koraka koji uvijek daje korektan odgovor. Sa druge strane, heuristika dozvoljava na nekim inputima greške, ali na većini inputa ne.

1.5.1 Podijeli i vladaj tehnika (ili rekurzivna tehnika) rješavanja problema.

Na slici 1.4 ilustrujemo rješavanje problema tehnikom "podijeli i vladaj" (" divide and conquer"). Abstrahujući od nebitnih detalja izvršimo uprošćavanje ulaza u problem. Zatim problem podijelimo nad podprobleme $P_1, P_2, \ldots, P_b$ i nadjemo rekurzivno njihova rješenja $R_1, R_2, \ldots, R_b$, respektivno. Sljedeća etapa je računanje odgovora iz tih rješenja i davanje izlaza.



Slika 1.4: Divide and conquer tehnika

Svaki od b problema dobija $\frac{n}{c}$ dužinu inputa od dužine ulaza n. Analiza kompleksnosti takvog algoritma daje sljedeću rekurentnu formulu

$$T(n) = \begin{cases} bT(\frac{n}{c}) + n^d, & n > 1 \\ 1, & n = 1 \end{cases} \quad (1.1)$$

Prema slici 1.4 jasno je kako dobivamo član $bT(\frac{n}{c})$. Član n^d dobivamo iz zahtjeva da izračunavanje odgovora iz rješenja bude polinomijalno.

Rješenje dobivene rekurentne jednačine tražićemo u obliku $T(n) = O(n^{h(a,b,c)})$.

Prema jednačini 1.1 dobivamo

$T(n) = n^d + b[(\frac{n}{c})^d + b[(\frac{n}{c^2})^d + b \cdot T(\frac{n}{c^3})]]$.

Uslov zaustavljanja razlaganja nalazimo iz $\frac{n}{c^i} = 1$, odakle je $i = \log_c n$.

Slijedi da je $T(n) = n^d + b(\frac{n}{c})^d + b^2(\frac{n}{c^2})^d + b^3(\frac{n}{c^3}) + ... + b^i(\frac{n}{c^i})^d = n^d[1 + \frac{b}{c^d} + (\frac{b}{c^d})^2 + (\frac{b}{c^d})^3 + ... + (\frac{b}{c^d})^i]$

Mi razlikujemo izmedju $b?cd(\Leftrightarrow \log_c b?d)$, pri čemu je $? \in \{=, <, >\}$. Sumirajući odgovarajući red, nalazimo sljedeća tri slučaja:

1. $\log_c b = d \Rightarrow T(n) = O(n^d \log_c n)$
2. $\log_c b < d \Rightarrow T(n) = O(n^d)$
3. $\log_c b > d \Rightarrow T(n) = O(n^d(\frac{b}{c^d})^{\log_c n}) = O(n^d \frac{b^{\log_c n}}{(c^{\log_c n})^d}) = O(n^{\log_c b})$, jer je $b^{\log_c n} = b^{\log_c b \log_b n}$

Izraz $\log_c b$ nazivamo eksponent rekurzije, a d nazivamo direktni eksponent.

Zadatak 1.2. *Ponoviti gore navedenu analizu ako je ulaz podijeljen na jedan problem sa dužinom $\frac{n}{3}$ i jedan problem sa dužinom $\frac{2n}{3}$. Prelaznu složenost uzeti n^d.*

Formule $1, 2,$ i 3 ćemo zvati **Glavni argument rekurzije**. Pošto ove formule predstavljaju rješenje rekurentne jednačine 1.1, lako ih možemo primijeniti u specijalnim slučajevima rekurzije. Primjene ilustrujemo primjerima koji slijede.

1.5.2 Binarno pretraživanje: nalaženje zadatog elementa u uredjenom nizu brojeva za vrijeme $O(\log n)$

Dat je sortiran niz od n brojeva. Treba naći clan niza jednak zadatom elementu x. Dajemo sljedeći algoritam.

Biramo u nizu clan $a_{\frac{n}{2}}$.

Ako je $a_{\frac{n}{2}} = x$, tada Stop.

Ako je $a_{\frac{n}{2}} < x$, ostaje da se pretraži samo desna polovina,

a ako je $a_{\frac{n}{2}} > x$, ostaje da se pretraži samo lijeva polovina.

Složenost u najgorem slučaju je $T(n) = 1 \cdot T(\frac{n}{2}) + n^0$, jer razmatramo samo jedan podproblem (lijevi ili desni), a razmatrani podproblem dobiva ulaz $n/2$. Na kraju dolazimo do traženog elementa ili do odgovora da elemenat ne postoji u nizu, u n^0 koraka.

Dakle, na osnovu formule 1.1 imamo $d = 0, b = 1, c = 2$, što daje $\log_c b = \log_2 1 = 0 = d$, pa treba koristiti formulu 1. Glavnog argumenta rekurzije iz 1.5.1.

Prema toj formuli slijedi da je složenost algoritma binarnog pretraživanja $T(n) = O(\log n)$.

Implementacija algoritma binarnog pretraživanja u C++

(a) Rekurzivna procedura

Dajemo rekurzivnu proceduru za rješavanje problema. Ako je x u nizu $a[]$ tada će procedura vratiti njegovu lokaciju, a inače će biti vraćeno -1.

```
int find(float *a, int start, int stop, float x)
{
    if(start>stop) return -1;
    int mid=(start+stop)/2;
    if(x==a[mid] return mid);
    if(x<a[mid] return find(a,start,mid-1,x);
    if(x>a[mid] return find(a,mid+1,stop,x);
}

Dajemo testni program za proceduru find
int main()
{float a[]={11.1, 22.2,33.3,44.4,55.5,66.6,77.7,88.8};
    int k=find(a,0,8,66.6);
    cout<<"k=" <<k<<end;
    k=find(a,0,8,50);
    cout<<"k=" <<k<<end;}
```

(b) Iterativna procedura

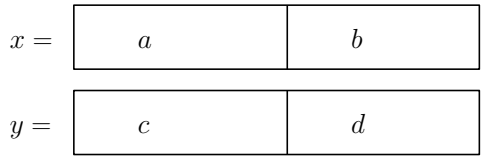
Dajemo iterativnu proceduru koja daje isti efekat kao i gore navedena rekurzivna procedura.

```
template <class T>
int find (constT*a, int begin, int end, const T & target)
{int lo=begin; int hi=end-1;
while(lo<=hi){
    int mid=(lo+hi)/2;//Na\'{c}i sredinu.
    if(a[mid]==target)return mid; //Nadjen!
    if(a[mid]>target)hi=mid-1;//Tra\v{z}i u lijevoj polovini.
    else lo=mid+1l //Tra\v{z}i u desnoj polovini.
}
return end;
}
```

Testni program bi se napravio slično kao u slučaju (a).

1.5.3 Efektivno množenje dugih cijelih brojeva u vremenu $O(n^{1.59})$

Razmotrimo množenje dva n-bitna binarna broja. Tradicionalni metod treba $T(n) = O(n^2)$ operacija nad bitovima. U metodu koji izlažemo dovoljno je $T(n) = O(n^{1.59})$ operacija nad bitovima. Razbijemo x i y na dva jednaka dijela kao na slici 1.5.



Slika 1.5: Razbijanje brojeva na po dva dijela

Ako svaki od tih dijelova posmatramo kao $\frac{n}{2}$-cifreni binarni broj, onda možemo predstaviti množenje x i y u obliku $xy = (a \cdot 2^{\frac{n}{2}} + b)(c \cdot 2^{\frac{n}{2}} + d) = ac \cdot 2^n + (ad + bc) \cdot 2^{\frac{n}{2}} + bd$.

Odavde slijedi da proizvod možemo izračunati pomoću programa

```
{
        u = (a+b)*(c+d);
        v = a*c;
        w = b*d;
        z = v*2^n + (u–v–w)*2^{n/2} + w;
}
```

Prema formuli 1.1 za složenost programa slijedi da je $b = 3$, $c = 2$ i $d = 1$, jer sada imamo 3 množenja binarnih brojeva dužine $n/2$, a onda rješenje sastavljamo u n^1 koraka, pa je $T(n) = 3T(\frac{n}{2}) + n$. Pošto je $\log_c b = \log_2 3 > 1$, koristimo slučaj 3. Glavnog argumenta rekurzije i za vremensku složenost množenja dva binarna broja dobivamo $T(n) = O(n^{1.59})$. Može se pokazati da nad proizvoljnim poljem treba najmanje tri operacije množenja da bi se sračunao proizvod dva duga broja. Najbolji poznati algoritam ima složenost $O(n \log n \log\log n)$.

1.5.4 Efektivno množenje matrica u vremenu $O(n^{2.81})$

Neka su A i B dvije $(n \times n)$ matrice. Bez umanjenja opštosti možemo smatrati da je n stepen broja 2, zbog toga što dopunjavanje matrice nulama neće kvalitativno uticati na funkciju složenosti.

Svaku matricu A i B možemo razbiti na četiri matrice i preko njih izraziti proizvod matrica A i B:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{12} & C_{22} \end{bmatrix}$$

gde je

$$C_{11} = A_{11}B_{11} + A_{12}B_{21},\ C_{12} = A_{11}B_{12} + A_{12}B_{22},$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21},\ C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Ako se C_{ij} izračunava pomoću m množenja i a sabiranja/oduzimanja, tada rekurzivno primenjujući gornji algoritam dobivamo $T(n) \leq mT(\frac{n}{2}) + \frac{a \cdot n^2}{4}, n > 2$, gde je n stepen dvojke.

Običan metod množenja matrica ima složenost $T(n) = O(n^3)$. Naš rekurzivni metod ima 8 mnozenja $(\cdot)$ i 4 sabiranja $(+)$, a za sastavljanje rješenja treba n^2 operacija. To daje $T(n) = 8T(\frac{n}{2}) + n^2$. Pošto je $\log_c b = \log_2 8 = 3 > d = 2$, prema slučaju 3 Glavnog argumenta rekurzije slijedi da je $T(n) = O(n^{\log_c b}) = O(n^3)$. Dakle, ako na ovaj način primijenimo tehniku podijeli i vladaj, onda nećemo ostvariti napredak u odnosu na uobičajeni način množenja matrica.

V. Štrasen (vidi [29]) je napravio svojevrsno iznenadjenje kada je našao metod množenja matrica nad proizvoljnim prstenom u kojem je dovoljno sedam množenja, pa je poboljšao vremensku složenost množenja matrica.

Na početku se formiraju proizvodi blokova matrica.

$m_1 = (A_{12} - A_{22})(B_{21} + B_{22})$
$m_2 = (A_{11} + A_{22})(B_{11} + B_{22})$
$m_3 = (A_{11} - A_{21})(B_{11} + B_{12})$
$m_4 = (A_{11} + A_{12})B_{22}$
$m_5 = A_{11}(B_{12} - B_{22})$
$m_6 = A_{22}(B_{21} - B_{11})$
$m_7 = (A_{21} + A_{22})B_{11}$

Zatim se izračuna

$C_{11} = m_1 + m_2 - m_4 + m_6,$
$C_{12} = m_4 + m_5,$
$C_{21} = m_6 + m_7,$
$C_{22} = m_2 - m_3 + m_5 - m_7.$

Prema formuli 1.1 imamo $T(n) = 7T(\frac{n}{2}) + 18n^2$. Prema Glavnom argumentu rekurzije dobivamo $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$, što je bolje od klasičnog algoritma.

Napomena. Neki pokušavaju popraviti algoritam tako što startuju sa 3×3 blokovima matrica. Tada je $T(n) = xT(\frac{n}{3}) + O(n^2)$ i treba naci x(tj. broj množenja) tako da bude $\log_3 x < 2.81$. Za $x = 27$ jednostavno je $f(n) = O(n^{\log_3 27}) = O(n^3)$. Za $x = 24$ je nadjeno $f(n) = O(n^{\log_3 24}) \geq 2.81$, što je lošije od algoritma Strassena.

D. Coopersmith i Sh. Winograd (vidi [8]) su $1972.g.$ našli algoritam koji množi matrice $n \times n$ u vremenu $O(n^{2,375})$. Do $2010.\,g.$ njihov algoritam je bio najbrži poznati. U $2010.\,g.$ A. Stothers (vidi [28]) je dao algoritam koji radi u vremenu $O(n^{2.374})$, a $2011.\,g.$ V. Williams (vidi [30]) je popravila granicu algoritma na $O(n^{2.373})$. U praksi se ipak najčešće koristi Štrasenov algoritam, jer su ostali algoritmi teški za implementaciju.

Zadatak 1.3. *Koristeći metode linearne algebre pokušajte rekonstruisati način na koji je Strassen došao do gore navedenih sedam formula za množenje 2×2 matrica.*

Zadatak 1.4. *Dokazati*

(a) Proizvod dvije $n \times n$ kvadratne matrice zahtijeva najmanje $O(n^2)$ koraka.

(b) Šest množenja nije dovoljno za nalaženje proizvoda dvije 2×2 matrice, nad proizvoljnim prstenom, Štrasenovim postupkom. [Ako bi šest množenja bilo dovoljno, onda bi kompleksnost množenja matrica, nad proizvoljnim prstenom, bila $T(n) = O(n^{\log_2 6}) = O(n^{2.59})$].

1.5.5 Složenost drugih operacija sa matricama

U ovom dijelu ćemo vidjeti da neke operacije nad matricama, kao što su LUP dekompozicija, inverzija matrica, računanje determinante, mogu da se svedu na množenje matrica , a onda se mogu izvesti isto tako brzo kao i množenje matrica. Množenje matrica je svodljivo na inverziju matrica, pa svaki napredak u poboljšanju algoritama u jednoj oblasti, povlači odgovarajuće poboljšanje u drugoj oblasti. Na kraju odjeljka ćemo pokazati da postoje algoritmi za množenje Booleovih matrica kojima je složenost manja od $O(n^3)$.

LUP dekompozicija matrica
Iz linearne algebre je poznato da nesingularna matrica A može da se piše u obliku $A = LUP$. Pri tome je L donja trouglasta matrica, U je gornja trouglasta matrica, sa jedinicama na dijagonali, a P je permutaciona matrica. Kažemo da matrica A ima LUP dekompoziciju.
Tvrdnja. LUP dekompozicija nesingularne, $n \times n$ matrice može, da se izvede u vremenu $O(n^{2.81})$.
Dokaz. Za dokaz vidjeti knjigu [9]. □

Rješavanje sistema linearnih jednačina
Efektivne metode rješavanja sistema linearnih jednačina koriste metod LUP. Ako je sistem zadat jednačinom $Ax = b$ i ako je $A = LUP$, tada možemo pisati $LU(Px) = b$. Stavljajući $Px = y$, $Uy = z$ i $Lz = b$, nalazimo da rješavanje zahtijeva $O(n^2)$ vrijeme. Kada je nadjeno z, onda rješavanje $Uy = z$ zahtijeva $O(n^2)$ vrijeme, a rješavanje $Px = y$ još $O(n)$. Zajedno sa vremenom $O(n^{2.81})$, potrebnim za nalaženje LUP dekompozicije, to daje ukupno vrijeme $T(n) = O(n^{2.81})$ potrebno za rješavanje sistema linearnih jednačina.

Invertovanje matrica
Prvo razmotrimo invertovanje trougaone matrice

$$A = \begin{bmatrix} B & C \\ 0 & D \end{bmatrix}.$$

Tvrdnja.

$$A^{-1} = \begin{bmatrix} B^{-1} & -B^{-1}CD^{-1} \\ 0 & D^{-1} \end{bmatrix}.$$

Dokaz.

$$AA^{-1} = \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix}.\square$$

Složenost je $T(n) = 2T(\frac{n}{2}) + \Theta(n^{2.81})$, pa konačno dobivamo $T(n) = O(n^{2.81})$.

U slučaju proizvoljne, nesingularne matrice imamo $A = LUP \Rightarrow A^{-1} = P^{-1}U^{-1}L^{-1}$. Za nalaženje P^{-1}, U^{-1} i L^{-1} složenost je $O(n^2)$, $O(n^{2.81})$ i $O(n^{2.81})$ respektivno. Odatle slijedi da je ukupna složenost $T(n) = O(n^{2.81})$.

Tvrdnja Matrično množenje nije teže od invertovanja.

Dokaz. Neka su date matrice A i B, tipa $n \times n$. Želimo naći matricu AB. Lako je provjeriti da je

$$\begin{bmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix}^{-1} = \begin{bmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{bmatrix}.$$

Odavde vidimo da se proizvod matrica tipa $n \times n$ može dobiti invertovanjem matrice tipa $3n \times 3n$. To dokazuje našu tvrdnju. $\square$

Determinante

Pošto je $det(A) = det(L)det(U)det(P)$, onda je složenost $O(n^{2.81})$ za LUP dekompoziciju, plus složenost za postprocesiranje $O(n)$. Ukupna složenost je $T(n) = O(n^{2.81})$.

Zadatak 1.5. *Posmatrajmo A, B kao matrice reda $n \times n$ sa $k-$bitnim cijelim elementima. Šta je najveći, prolazni rezultat koji će biti memorisan negdje u rekurziji u algoritmu Štrasena? (Naći dobru granicu). Naći najbolju granicu koju možete.*

Množenje Booleovih matrica

Štrasenov algoritam zahtijeva operacije sabiranja $(+)$, oduzimanja $(-)$ i množenja $(\cdot)$. Standardni algoritam može se primijeniti i tamo gdje operacija oduzimanja $(-)$ nije definisana, na primjer sa Booleovim operacijama.

Ako posmatramo Booleovu strukturu kao semi-prsten (sa $1 + 1 = 1$), onda se može pokazati da je množenje dvije Booleove matrice ekvivalentno računanju tranzitivnog zatvaranja grafa. Na žalost, zatvoreni semi-prsten nad $\{0, 1\}$ nije prsten, pa se Štrasenov algoritam ne može direktno primijeniti na množenje dvije Booleove matrice.

Nad $\{0, 1\}$ možemo napraviti polje $GF(2)$, sa jedinstvenim inverznim elementom, sa $0 + 0 = 0$, $0 + 1 = 1 + 0 = 1$ i $1 + 1 = 0$. Strukturu $GF(2)$ ne možemo koristiti direktno pri množenju matrica. Kako onda izvesti množenje Booleovih matrica $A = [a_{ij}]$, $B = [b_{ij}]$, $a_{ij}, b_{ij} \in \{0, 1\}$, $i, j \in \{1, \ldots, n\}$?

Množenje

$$C = A \cdot B = [c_{ij}], c_{ij} = \bigvee_{k=1}^{n} (a_{ik} \wedge b_{kj})$$

se može izvesti na uobičajeni način sa $O(n^3)$ operacija $\vee$ i $\wedge$. Ne može se direktno primijeniti Štrasenov algoritam jer nemamo strukturu prstena. Problem se može

prevazići tako što uložimo Booleovu strukturu u cijele brojeve $\mod (n+1)$, primijenimo Štrasenov algoritam za množenje AB kao cijelih, a onda u rezultatu zamijenimo svaki ne-nula cijeli broj jedinicom, a nule nulama. Lako je provjeriti da je dobiveni rezultat korektan. Ako je $n+1$ prost broj, tada cijeli po modulu $n+1$ čine polje. Inače, čine prsten.

Najbolja poznata vremenska složenost algoritma je

$$T(n) = O(n^{2.81}) \cdot O(\log n \cdot \log\log n \cdot \log\log\log n),$$

a odatle

$$T(n) = O(n^{2.81} \cdot \log n \cdot \log\log n \cdot \log\log\log n),$$

jer je najbolja poznata složenost za množenje $k-$bitnih brojeva

$$O(k \cdot \log k \cdot \log\log k).$$

Umjesto modula $n+1$, može se koristiti modul po $q = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 > n$, a onda koristimo modul posebno za svaki od ovih manjih.

Složenost najboljeg poznatog deterministčkog algoritma je

$$O(n^{2.81} \cdot \log n \cdot \log\log\log n \cdot \log\log\log\log n).$$

Sada ćemo probati izračunati $A \cdot B = C$, nad $\wedge$, $\vee$, tako što ćemo staviti

$$C = \bigvee_i [A \cdot (B \wedge R_i)],$$

pri čemu se proizvod $(\cdot)$ računa $\mod 2$ (nad $GF(2)$), a R_i je slučajna Booleova matrica u kojoj nule ostaju nule, a jedinice postaju nule sa vjerovatnoćom $\frac{1}{2}$.. Može se dokazati tvrdnja da je sa velikom vjerovatnoćom rezultat C korektan.

Zadatak 1.6. *Analizirati vjerovatnoću da je C korektna matrica kao funkciju od n i broja slučajnih matrica.* .

Najbolji poznati stohastički algoritam za množenje dvije Booleove matrice reda $n \times n$ ima složenost $T(n) = O(n^{2.81} \log n)$.

Osnovne pojmove o primjeni stohastičkih modela u teoriji algoritama opisali smo u dodatku A.2.

Zadatak 1.7. *Opravdati broj ponavljanja $O(\log n)$ za dobivanje vjerovatnoće ispod željene konstante.*

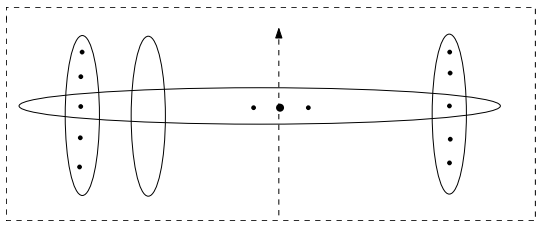
1.6 Nalaženje k-tog elementa u linearnom vremenu

Linearnim pretraživanjem nalazimo minimum ili maksimum niza u vremenu $O(n)$. Može se pokazati da je minimalan broj potrebnih poredjenja za nalaženje minimuma ili maksimuma niza jednak $\lceil \frac{3}{2}n \rceil - 2$. Opšti problem nalaženja $k-$tog

po veličini elementa niza smo riješili ranije u vremenu $n + k \log_2 n$. Ako $k-$ti elemenat niza S tražimo tako da prethodno sortiramo niz S, onda nam u najgorem slučaju treba $n \log n$ poredjenja. Problem nalaženja $k-$tog elementa niza značajno se razlikuje od problema sortiranja, pa se može očekivati bolji algoritam. U radu Blum M. et al. [5], prikazan je algoritam koji teorijski ima složenost $5n$, ako se broje samo poredjenja. U radu Schonhage A. et al. [25], autori su konstruisali algoritam koji ima složenost $3n$. Oba algoritma se neuobičajeno komplikovano zapisuju u jezicima programiranja. Zbog toga koristimo algoritam koji na svakom koraku odredjuje takav elemenat e koji omogućuje da se na svakom koraku dužina niza smanji najmanje za $n/4$ i da se dobije $T(n) = O(n)$ u najgorem slučaju. Ideju tog algoritma prikazujemo sljedećim pseudokodom i slikom (1.6).

int function $IZBOR(k,\ S)$:

1. **if** $\mid S \mid < 50$ then $\{$
2. urediti S;
3. **return** $k-$ti najmanji element iz S $\}$; **else**
4. $\{$ Razbiti S na $\lfloor \mid S \mid /5 \rfloor$ nizova po 5 elemenata u svakom nizu
5. pri čemu će ostati najviše 4 neiskorištena elementa niza;
6. urediti svaki petoelementni niz;
7. neka je M niz medijana tih petoelementnih nizova;
8. $m = IZBOR(\lceil \mid M \mid /2 \rceil, M)$;
9. Neka su S_1, S_2, S_3 - skupovi elemenata manjih, jednakih, većih od m, respektivno;
10. **if** $\mid S_1 \mid \geq k$ **then return** $IZBOR(k, S_1)$; **else**
11. **if** $(\mid S_1 \mid + \mid S_2 \mid \geq k)$ then **return** m; **else**
12. **return** $IZBOR(k- \mid S_1 \mid - \mid S_2 \mid, S_3)$.



Slika 1.6: Nalaženje k-tog elementa u linearnom vremenu

Iz analize navedenog koda zaključujemo

$$T(n) = \begin{cases} cn, & n \leqslant 49 \\ T(\frac{n}{5}) + T(\frac{3n}{4}) + cn, & n \geqslant 50 \end{cases}$$

$T(\frac{n}{5})$ trebamo za $IZBOR(\lceil \frac{\lfloor M \rfloor}{2} \rceil, M)$. Zatim u najgorem slučaju ne treba razmatrati $\frac{n}{4}$ elemenata u donjoj lijevoj četvrtini, ili u gornjoj desnoj četvrtini, pa trebamo $T(\frac{3n}{4})$. Za manipulisanje konačnim skupovima, po pet elemenata, trebamo linearan broj koraka cn.

Teorema 1.2. *Vrijeme rada algoritma u najgorem slučaju je* $T(n) \leqslant 20n$.

Dokaz. Dokazujemo indukcijom po n. Osnovni korak je trivijalan. Ako $(\forall k)(k < n \implies T(k) \leqslant 20k)$, tada je $T(n) \leqslant T(\frac{n}{5}) + T(\frac{3n}{4}) + n \leqslant 20 \cdot \frac{n}{5} + 20 \cdot \frac{3n}{4} + n = 20n$.□

Dajemo i detaljniji dokaz navedene ocjene složenosti. Označimo pesimističku složenost tog algoritma sa $T(n)$.

Složenost uredjivanja petorki je linearna funkcija od n, jer je složenost sortiranja jedne petorke konstantna. Označimo tu složenost sa dn, $d > 0$.

Pri rekurzivnom pozivu razmatramo svaki peti elemenat niza, pa je složenost toga koraka $T(\lfloor \frac{n}{5} \rfloor)$.

Nakon nalaženja medijane, prema slici (1.6), zaključujemo da su $3\lfloor \lfloor \frac{n}{5} \rfloor / 2 \rfloor$ elemenata niza manji od medijane i isto toliko elemenata niza veći od medijane.

Ako je $n \geq 90$, tada važe sljedeće nejednakosti

$(6n \geq 5n + 90) \Rightarrow (n \geq \frac{5}{6}n + 15) \Rightarrow (\frac{n}{5} \geq \frac{n}{6} + 3) \Rightarrow (\lfloor \frac{n}{5} \rfloor \geq \lfloor \frac{n}{6} + 1 \rfloor + 2) \Rightarrow (\lfloor \frac{n}{5} \rfloor \geq \frac{n}{6} + 2) \Rightarrow (\lfloor \frac{n}{5} \rfloor / 2 \geq \frac{n}{12} + 1) \Rightarrow (\lfloor \lfloor \frac{n}{5} \rfloor / 2 \rfloor \geq \lfloor \frac{n}{12} + 1 \rfloor) \Rightarrow (3\lfloor \lfloor \frac{n}{5} \rfloor / 2 \rfloor \geq \frac{n}{4})$.

To znači da je za $n \geq 90$ broj elemenata manjih ili jednakih medijani, najmanje $\frac{n}{4}$, a isto tako i broj elemenata koji su veći ili jednaki medijani. Složenost manipulisanja možemo ograničiti linearnom funkcijom ne, gdje je broj $e > 0$.

Sada je $T(n) \leq (d + e)n + T(\lfloor \frac{n}{5} \rfloor) + T(\lfloor \frac{3}{4}n \rfloor)$, za $n \geq 90$.

Za n koji su manji od 90, funkcija $T(n)$ je ograničena nekom konstantom c_1. Ako stavimo $c = max((d + e), c_1)$, onda imamo

$T(n) \leq cn$ za $n < 90$ i

$T(n) \leq cn + T(\lfloor \frac{n}{5} \rfloor) + T(\lfloor \frac{3}{4}n \rfloor)$, za $n \geq 90$. Sada je lako, kao u teoremi, indukcijom pokazati da je $T(n) = O(n)$.

Zadatak 1.8. *Neka je dato n brojeva v_i, svaki sa težinom w_i, respektivno. Težinska medijana je elemenat v_i takav da je totalna težina svih elemenata manjih od njega, otprilike jednaka težini svih elemenata većih od njega. Naći algoritam koji u najgorem slučaju rješava ovaj problem u linearnom vremenu.*

Zadatak 1.9. *Pretpostavimo da su n datih, sortiranih elemenata, uniformno distribuirani medju vrijednostima $[1, L]$. Pokazati (pomoću intuitivnih argumenata) kako naći neki zadati x u $O(\log \log n)$ awerage case vremenu $(L \gg n)$.*

Zadatak 1.10. *Razmotriti problem računanja $TRESHOLD_2$ (postojanje najmanje dva bita "1") na $\wedge \vee$ kapijama vrijednosti n bita. Analizirati složenost sljedeća dva algoritma, uključujući koeficijent vodećeg izraza, ali isključujući izraze nižeg reda.*

(a) Dijeljenjem na polovine,

(b) Aranžiranjem kvadrata.

Dodatno, provjeriti kako generalizovati (a) *i* (b) *na* $TRESHOLD_J$, $J \in \mathbb{N}$.

Zadatak 1.11. *Odakle dolazi "magični broj"* 5 *elemenata u štapovima? Imamo* $\frac{n}{5} + \frac{3n}{5} \leq n$. *Ako umjesto* 5 *uzmemo* 3, *onda ova nejednakost neće biti ispunjena. Ispitajte šta će se desiti ako umjesto* 5 *elemenata u štapovima uzmemo* 7 *elemenata.*

1.7 Dizajn i analiza složenosti brzih algoritama za sortiranje

U ovom dijelu, koristeći rekurziju, razvijamo algoritame za sortiranje u vremenu $O(n \log n)$. Dokazaćemo da je u opštem slučaju, za algoritme sortiranja poredjenjem, donja granica $T(n) = O(n \log n)$. Kada se o članovima niza znaju neki dodatni podaci, to jest, ako se ne radi o opštem slučaju zadavanja niza, onda se niz može sortirati i u linearnom vremenu. To pokazujemo na primjerima "bucket sort-a" i "radix sort-a".

1.7.1 Sortiranje merdžovanjem (spajanjem). Dizajn i analiza složenosti

Razmotrimo niz cijelih brojeva

$$x_1, x_2, \ldots, x_n.$$

Radi prostijeg računa, bez umanjenja opštosti, pretpostavimo da je n stepen broja 2. Jedan od načina da sortiramo dati niz je da razbijemo dati niz na dva podniza

$$x_1, x_2, \ldots, x_{\frac{n}{2}}$$

i

$$x_{\frac{n}{2}+1}, x_{\frac{n}{2}+2}, \ldots, x_n,$$

sortiramo svaki od njih i zatim ih merdžujemo (spojimo, smiješamo). Pod merdžovanjem podrazumijevamo spajanje dva već sortirana podniza u jedan sortiran niz.

Složenost u najgorem slučaju (worst-case) nalazimo primjenom metode "podijeli i vladaj". Pošto merdžovanje zahtijeva vrijeme $T(n) = O(n)$, iz opisa algoritma imamo

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

pa je

$$d = 1,\ b = 2,\ c = 2$$

i

$$\log_c b = \log_2 2 = 1 = d.$$

Prema formuli $1.$ glavnog argumenta rekurzije (1.1) odatle slijedi da je

$$T(n) = O(n \log n).$$

Dakle, ako nas interesuje worst-case, onda je sortiranje merdžovanjem bolje od sortiranja umetanjem, sortiranja mjehurićima i sortiranja izborom.
Implementaciju sortiranja merdžovanjem dajemo u programskom jeziku $C++$.

```
//Merge sort
template <class T>
void merge(T *a, int n1, int n2)
{T *temp = new T[n1+n2];
 int i=0, j1=0, j2=0;
    while (j1<n1 && j2<n2)
        temp[i++]=(a[j1]<=a[n1+j2]?a[j1++]:a[n1+j2++]);
    while(j1<n1)
        temp[i++]=a[j1++];
    while(j2<n2)
        temp[i++]=(a+n1)[j2++];
    for(i=0;i<n1+n2;i++)
        a[i]=temp[i];
        delete[] temp;
 }
template <class T>
void sort(T*a, int n)
{ if(n>1)
    {int n1=n/2;
    int n2=n-n1;
    sort(a,n1);
    sort(a+n1,n2);
    merge(a,n1,n2);
    }
}
```

1.7.2 Brzo sortiranje (Quick sort). Dizajn i analiza složenosti

Algoritam:

- Izabrati slučajno(random) element dijeljenja niza.
- Podijeliti niz u podskupove manjih i većih elemenata u odnosu na izabrani element.
- Posebno sortirati dobivene podskupove.
- Nadovezati sortirane podskupove.

Worst case vrijeme izvodjenja:
U slučaju da u svakom koraku dijeljenja izaberemo maksimalan (minimalan) element, za niz od n elemenata imamo

$$T(n) = n + (n-1) + \ldots + 1 = O(n^2).$$

Best case vrijeme izvodjenja:

U slučaju da u svakom koraku izaberemo medium element, prema formuli iz "Podijeli i vladaj" imamo $T(n) = 2T(\frac{n}{2}) + n$ što daje $T(n) = O(n \log n)$.
Average case analiza:

Pošto podjela može biti izvršena na bilo kojem mjestu i niza, u prosjeku imamo

$$T(n) \leqslant c \cdot n + \sum_{i=1}^{n} \frac{1}{n}(T(i-1) + T(n-i)) = c \cdot n + \frac{2}{n} \sum_{i=0}^{n-1} T(i),$$

pri čemu je $c \cdot n$ vrijeme podjele za n elemenata.

Mi želimo dokazati ograničenje $T(n) \leqslant kn \log n$.

Koristeći prethodnu formulu imamo

$$T(n) \leq cn + \frac{2}{n} \sum_{i=0}^{n-1} i \ \log i \leq cn + \frac{2k}{n} \left[\frac{n^2 \log n}{2} - \frac{n^2}{4} \right] \leq kn \log n + cn - \frac{k}{2} n$$

Stavljajući $k = 2c$ dobivamo $T(n) \leq kn \log n$. Ocjena je postignuta koristeći induktivnu hipotezu $T(i) \leq i \log i$ za $i < n$, te sljedeći kvaziargument

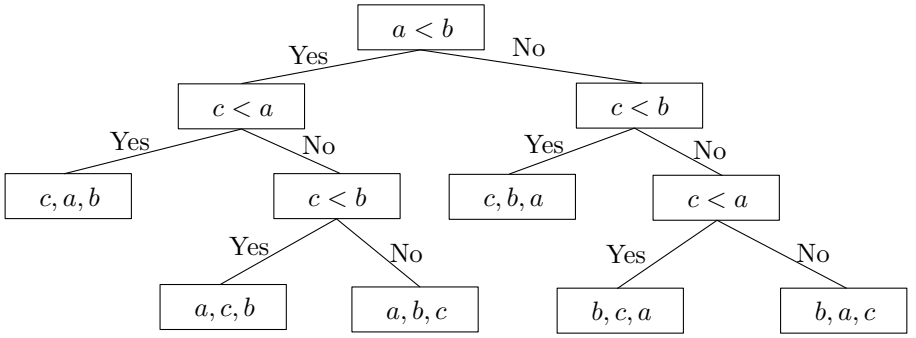
$$\sum_{i=0}^{n-1} i \log i = \int_{0+}^{n-1} x \log x dx = \frac{x^2}{2} \log x \ |_0^{n-1} - \int_{0+}^{n-1} \frac{x^2}{2} \frac{1}{x} dx$$

$$= \frac{(n-1)^2}{2} log(n-1) - \frac{x^2}{4} \ |_0^{n-1} = \frac{(n-1)^2}{2} \log(n-1) - \frac{(n-1)^2}{4}$$

```
//Quick sort
template <class T>
void quicksort(T *a, int lo, int hi)
{ if (lo >= hi) return;
    T pivot = a[hi];
    int i= lo-1;
    int j=hi;
    while(i<j)
        {while (a[++i]<pivot);
        while(j>=0 && a[--j]<pivot);
        if(i<j) swap (a[i], a[j]);
}
swap(a[i],a[hi]);
//Invarijanta: a[j]<=a[i]<=a[k] za
//             lo<=j<i<k<=hi
quicksort(a,Lo,i-1);
quicksort(a,i+1,hi);
}

template <class T>
void sort(T *a, int n)
    {quicksort(a,0,n-1);
    }
```

1.7.3 Donje ograničenje $O(n \log n)$ za algoritme sortiranja poredjenjem u opštem slučaju

Dokazujemo da je $\Omega(n \log n)$ donje ograničenje za vrijeme sortiranja n vrijednosti, ako se koristi poredjenje, u opštem slučaju. Slično kao u primjeru drveta odlučivanja za tri elementa a, b, c (Slika 1.7), gdje imamo $3!$ mogućih permutacija, predstavljenih listovima drveta, vidimo da i u opštem slučaju algoritam sortiranja poredjenjem može biti opisan drvetom odlučivanja sa najmanje $n!$ listova.



Slika 1.7: Primjer drveta odlučivanja za tri elementa

Jasno je da je worst case vrijeme jednako visini dohvatljivog dijela drveta. Svako drvo visine h može imati najviše 2^h listova. Za $n!$ listova visina mora biti veća ili jednaka $\log_2 n!$. Posto je $n! \approx (\frac{n}{e})^n$, to je $h \geqslant \log n! \approx n[\log_2 n - \log_2 e] = O(n \log n)$

Teorema 1.3. *Donja granica $\Omega(n \log n)$ za sortiranje poredjenjem, primjenljiva je i za average case kompleksnost.*

Dokaz

Opšta definicija average case kompleksnosti je

$A(n) = \sum_{|x|=n} \rho(x)T(x)$, pri čemu je $\rho(x)$ odgovarajuća gustina.

Kod nas je, u opštem slučaju, $\frac{1}{n!} \sum_{|x|=n} T(x)$. Označimo sa $D(T)$ sumu dubina listova binarnog drveta poredjenja T. Neka je $D(m) = minD(T)$ najmanja suma dubina listova po binarnim drvetima poredjenja sa m listova. Treba dokazati $D(m) = m \log m$.

Dokaz izvodimo indukcijom.

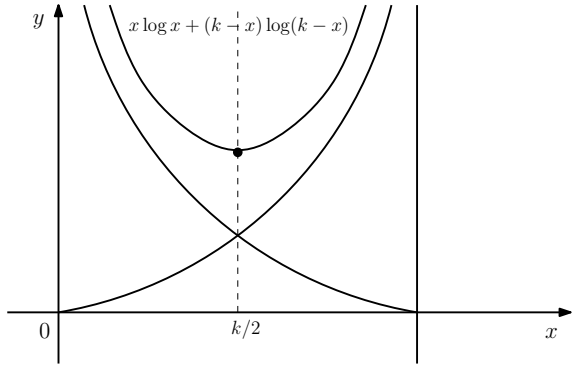
Za $m = 1$ je trivijalno 0.

Za $m > 1$:

Pretpostavimo da drvo TR ima k listova, tako da je i listova u lijevom poddrvetu TR_i i $k - i$ listova u desnom poddrvetu TR_{k-i}.

Važi $D(TR) = i + D(TR_i) + (k - i) + D(TR_{k-i}) \geqslant k + D(TR_i) + D(TR_{k-i})$

$D(k) \geqslant k + \min[D(i) + D(k - i)] \geqslant k + \min_{1 \leqslant i \leqslant k-1} [i \log i + (k - i)(\log(k - i)]$, po induktivnoj hipotezi!



Slika 1.8: $\min_{1 \leqslant x \leqslant k-1} [x \log x + (k-x)(\log(k-x)]$

Sa slike 1.8 vidimo da je minimum dostignut za $\frac{k}{2}$. Zbog toga je gornji izraz jednak

$$k + 2\frac{k}{2} \log \frac{k}{2} = k + k \log k - k = k \log k.$$

Sada je $D(n!) = n! \log n! \approx n! n \log n$. Odavde slijedi da je
$A(n) = \frac{1}{n!} D(n!) = \Omega(n \log n)$. □

Zadatak 1.12. *Neka je dat nesortiran niz od n vrijednosti $A[1..n]$. Za svaki element $A[i]$ posmatramo elemente prijatelje koji su veći od $A[i]$ po vrijednosti i nalaze se desno od njega u nizu $A[1..n]$. Naći efektivan algoritam i dokazati odgovarajuće donje ograničenje kada:*

1. *Želite naći za svaki $A[i]$ njegovog prijatelja koji mu je najbliži po vrijednosti.*

2. *Želite naći najbližu poziciju njegovog prijatelja.*

Zadatak 1.13. *Dokazati da je $\Omega(n \log n)$ donje ograničenje složenosti poredjenjem za jedinstvenost elementa. [Ako su svi elementi na ulazu različiti, onda imamo jedinstvenost elemenata.]*

1.8 Sortiranje u linearnom vremenu

U većini zadataka sortiranja dat je neki konkretan skup S i konkretna relacija linearnog uredjenja na tom skupu. Opšti algoritmi sortiranja mogu se uspješno

primijeniti, ali to ne znači da ćemo tako postići najbolje ocjene vremena sortiranja u najgorem slučaju i u statističkom slučaju. Zaista, ne moramo se ograničiti samo na poredjenje elemenata skupa S, već možemo koristiti sve raspoložive osobine skupa S.

Na primjer, ako znamo da je S skup brojeva, onda možemo koristiti aritmetičke operacije. Jedan od prostijih modela zadataka tog tipa je "bucket sort".

Bucket sort

Neka je S skup prirodnih brojeva $1, \ldots, m$. Neka niz $a[1], \ldots, a[n]$ elemenata tog skupa, koji želimo sortirati, ima dužinu n, koja je znatno veća od veličine m. Nema potrebe za poredjenje elemenata u ovom slučaju. Mnogo lakše je sortirati ovaj niz praveći za svako j, $1 \leq j \leq m$ listu elemenata. Pregledajući niz $a[1], \ldots, a[n]$, elemenat $a[i]$ šaljemo u listu sa brojem j, ako je $a[i] = j$. Po završetku tog koraka, dovoljno je spojiti liste, od prve liste, do liste sa brojem m.

Ovaj algoritam naziva se bucket sort. Prva faza ovog algoritma ima cijenu $O(n)$, jer za svaki elemenat $a[i]$ obavljamo konačan broj operacija, za $i \in \{1, \ldots, n\}$. D ruga faza, to znači povezivanje lista, košta $O(m)$, jer za svaki korak povezivanja treba konstantno vrijeme, a imamo m lista. Algoritam sortira u vremenu $T(n) = O(n + m) = O(n)$.

Algoritam možemo uopštiti na proizvoljan skup realnih brojeva. Neka je niz $a[1], \ldots, a[n]$ niz realnih brojeva u zatvorenom intervalu $[min, max]$.

Podijelimo zatvoreni interval realnih brojeva, od min do max na $m \leq n$ intervala jednake dužine. Označimo sa $h = (max - min)/m$ dužinu tih podintervala, a sa $b_j = min + jh$ lijeve krajeve tih podintervala. Za $j = 0, \ldots, m-2$ dobivamo podintervale, zatvorene sa lijeve strane, sa krajevima b_j, b_{j+1}, a za $j = m-1$ interval koji je zatvoren sa obje strane, sa krajevima b_{m-1}, b_m.

Elemenat $a[i]$ stavljamo u listu sa brojem j, ako $a[i]$ pripada $j-$tom podintervalu, ili drugačije, kada je $min + jh \leq a[i] < min + (j+1)h$. Odatle dobivamo broj liste $j = \lfloor (a[i] - min)/h \rfloor$. Poslije prenumeracije tih intervala od 1 do m, broj liste će biti $j + 1$, osim za $a[i] = max$, gdje je broj jednak m.

Pošto u jednoj listi elementi nisu jednaki, treba ih unutar liste sortirati. Možemo u takvoj situaciji primijeniti sortiranje umetanjem, jer elementi formiraju jednosmjernu listu, pa umetanje novog elementa na pravo mjesto može biti lako realizovano. Ovaj algoritam naziva se *sortiranje višestruke liste (multiple list sort)* i sličan je bucket sortu.

Teorema 1.4. *U najgorem slučaju sortiranje višestruke liste ima složenost* $T(n) = O(n^2)$. *Uz pretpostavku da je raspodjela podataka uniformna u intervalu* $[min, max]$, *prosječno vrijeme sortiranja je* $T(n) = O(n^2/m)$.

Dokaz. U najgorem slučaju, na cijenu tog algoritma, najveći uticaj ima umetanje elementa u odgovarajuću listu. Dužina liste u $i-$tom koraku iznosi najviše i, pa možemo primijeniti ocjenu ci, za neku konstantu $c > 0$. Sabirajući po $i = 1, \ldots, n$, dobiva se ocjena te faze $O(n^2)$. Pošto ostale operacije koštaju $O(n)$, dobivamo da je u najgorem slučaju složenost algoritma $T(n) = O(n^2) + O(n) = O(n^2)$.

Za dokaz drugog dijela teoreme pretpostavljamo da su dati brojevi raspodijeljeni uniformno u intervalu od min do max. Prema Bernulijevoj raspodjeli, vjero-

vatnoća da će se u listi sa brojem j naći tačno k elemenata je

$$P_k = \left(\begin{array}{c} n \\ k \end{array} \right) \left(\frac{1}{m} \right)^k \left(1 - \frac{1}{m} \right)^{n-k}.$$

Za $k-$elementni niz, algoritam sortiranja umetanjem obavlja prosječno $\frac{1}{2} \left(\begin{array}{c} k \\ 2 \end{array} \right)$ poredjenja. Prosječan broj poredjenja elemenata u drugoj fazi je

$$T(n) = m \sum_{k=2}^{n} \frac{1}{2} \left(\begin{array}{c} k \\ 2 \end{array} \right) P_k = m \sum_{k=2}^{n} \frac{1}{2} \left(\begin{array}{c} k \\ 2 \end{array} \right) \left(\begin{array}{c} n \\ k \end{array} \right) \left(\frac{1}{m} \right)^k \left(1 - \frac{1}{m} \right)^{n-k}.$$

Pošto je $\left(\begin{array}{c} n \\ k \end{array} \right) \left(\begin{array}{c} k \\ 2 \end{array} \right) = \left(\begin{array}{c} n \\ 2 \end{array} \right) \left(\begin{array}{c} n-2 \\ k-2 \end{array} \right)$, dobivamo

$$\begin{aligned} T(n) &= \frac{m}{2} \sum_{k=2}^{n} \left(\begin{array}{c} n \\ 2 \end{array} \right) \left(\begin{array}{c} n-2 \\ k-2 \end{array} \right) \left(\frac{1}{m} \right)^k \left(1 - \frac{1}{m} \right)^{n-k} \\ &= \frac{m}{2} \left(\begin{array}{c} n \\ 2 \end{array} \right) \sum_{k=2}^{n} \left(\begin{array}{c} n-2 \\ k-2 \end{array} \right) \left(\frac{1}{m} \right)^k \left(1 - \frac{1}{m} \right)^{n-k} \\ &= \frac{1}{2m} \left(\begin{array}{c} n \\ 2 \end{array} \right) \sum_{k=0}^{n-2} \left(\begin{array}{c} n-2 \\ k \end{array} \right) \left(\frac{1}{m} \right)^k \left(1 - \frac{1}{m} \right)^{n-2-k} \\ &= \frac{1}{2m} \left(\begin{array}{c} n \\ 2 \end{array} \right) \left(\frac{1}{m} + 1 - \frac{1}{m} \right)^{n-2} \\ &= \frac{n(n-1)}{4m}. \end{aligned}$$

Zbog toga je složenost u prosječnom slučaju $T(n) = O(\frac{n^2}{m})$. □

Ako uzmemo $m = n$ ili $m = n/2$, dobićemo da je složenost u prosječnom slučaju $O(n)$. Ipak, algoritmi ovog tipa imaju nedostatke, koje ne treba zaboravljati. Prvi nedostatak je trošenje memorije. Oba, bucket sort i sortiranje višestruke liste, zahtijevaju uvodjenje n kopija elemenata, po jedne za svako $a[i]$, n povezivanja i najmanje m povezivanja u kojima treba imati adrese lista.

Uz pretpostavku da aritmetičke operacije imaju jediničnu cijenu, navedeni algoritmi imaju vrijeme izvodjenja koje smo naveli, ako se računa cijena aritmetičkih operacija. U zadacima u kojima se aritmetičke operacije tretiraju kao logičke operacije na bitovima, takve operacije ne koštaju kao konstantan broj koraka, već su u najmanju ruku proporcionalne dužini sistema bitova. U tom slučaju, složenost navedenih algoritama neće više biti linearna.

Radiks (radix) sort

Radiks sort je algoritam koji je korišten za sortiranje bušenih kartica u mašinama za sortiranje. Zadatak sortiranja prirodnih brojeva sa d cifara radiks sort rješava

tako što ih prvo sortira po najmanje značajnim ciframa. Tada ih stavi u jedan niz tako da su brojevi koji se završavaju nulom, prije brojeva koji se završavaju jedinicom, itd. Zatim čitav niz sortira po ciframa drugim po značaju i stavlja ih u niz kao i ranije. Proces se nastavlja dok svi brojevi ne budu sortirani po svim ciframa. podrazumijevaćemo da se za sortiranje cifara koristi stabilan algoritam. Indukcijom se može dokazati da ćemo u tom slučaju, na kraju procesa, dobiti sortiran niz.

Ako sortiramo niz $[430, 368, 768, 347, 466]$, u prvom prolazu dobićemo niz $[430, 466, 347, 368, 768]$, u drugom prolazu $[430, 347, 466, 368, 768]$, a na kraju sortiran niz $[347, 368, 430, 466, 768]$. Program za radiks sort je lako napisati, neposredno iz opisane ideje sortiranja. U sljedećoj proceduri podrazumijevamo da svaki od n elemenata u nizu A, ima d cifara, pri čemu je na mjestu 1 najmanje značajna cifra, a na mjestu d najznačajnija cifra.

```
RADIKS_SORT(A<d)

    for i = 1 to d
        koristiti stabilan sort za sortiranje
        niza po cifri na i-tom mjestu
```

Tvrdnja. Neka je dato n brojeva koji su $d-$cifreni. Neka svaka cifra može uzeti k mogućih vrijednosti. Ako stabilni sort koji koristimo, ima vremensku složenost $\Theta(n + k)$, onda radiks sort sortira ove brojeve u vremenu $\Theta(d(n + k))$,
Dokaz. Svaki prolaz na n $d-$cifrenih brojeva ima složenost $\Theta(n + k)$. Postoji d prolaza, pa je ukupna vremenska složenost radiks sorta $\theta(d(n + k))$. □

Naše razmatranje sortiranja nije obuhvatilo spoljašnje sortiranje i sortiranje na mrežama. Nismo razmatrali neke algoritme unutrašnjeg sortiranja kao što je algoritam Shella ili algoritam skaliranja. Kompletnija informacija o sortiranju može se naći u čuvenoj knjizi D. Knutha ([16]).

Glava 2

Dinamičko programiranje

Algoritmi tipa podijeli i vladaj rješavaju problem tako što ga podijele u nezavisne podprobleme. Ako su podproblemi zavisni, onda primjenjujemo dinamičko programiranje. U ovom kontekstu metoda podijeli i vladaj bi radila više posla nego što je potrebno, jer bi ponavljala rješavanje zajedničkih podproblema. Dinamičko programiranje rješava svaki podproblem samo jednom, sačuva rezultat u tabeli i na taj način izbjegne ponovno rješavanje podproblema. Riječ "programiranje" u nazivu dinamičko programiranje, ne odnosi se na kodiranje već na tabularno rješavanje problema.

Problemi $optimizacije$ su tipični problemi koji se rješavaju metodom dinamičkog programiranja.

Algoritam dinamičkog programiranja može se podijeliti u četiri etape.

1. Nalaženje strukture $nekog\, optimalnog$ rješenja.

2. Rekurzivno definisanje vrijednosti $nekog\, optimalnog$ rješenja.

3. Računanje vrijednosti $nekog\, optimalnog$ rješenja metodom odozdo na gore

4. Konstruisanje $nekog\, optimalnog$ rješenja iz izračunatih podataka

Metodu dinamičkog programiranja ilustrovaćemo na primjeru množenja niza matrica. Na primjeru množenja niza matrica vidjećemo koje karakteristike treba imati problem da bi se mogao rješavati metodom dinamičkog programiranja. Dinamičko programiranje primijenićemo na problem nalaženja najdužeg zajedničkog podniza dva niza, a u problemu optimalne triangulacije poligona pokazaćemo kako je rješenje ovog problema neočekivlano slično problemu množenja niza matrica. Napomenućemo da se i Fibonačijevi brojevi obično računaju metodom dinamičkog programiranja.

2.1 Množenje niza matrica

Neka je dat niz matrica $\langle A, A_2, \ldots, A_n \rangle$ i neka treba naći njihov proizvod

$$A_1 A_2 \ldots A_n. \tag{2.1}$$

Množenje matrica je asocijativno, pa kako god postavimo zagrade koje odredjuju redoslijed množenja, dobivamo traženi proizvod. Nakon što znamo raspored zagrada proizvod (2.1) možemo računati koristeći standardan algoritam za množenje parova matrica, kao podprogram. Standardni algoritam za množenje matrica navodimo u pseudo kodu. Atributi $redovi$ i $kolone$ označavaju broj redova i kolona u matrici.

```
MnozenjeMatrica(A,B)
    if kolone [A] <> redovi[B]
        then error "neodgovarajuce dimenzije"
        else for i=1 to redovi [A]
            do for j=1 to kolone [B]
                do C[i,j]= 0
                    for k = 1 to kolone [A]
                        do C[i,j] = C[i,j] + A[i,k]B[k,j]
            return C
```

Podsjetimo se da se dvije matrice A i B mogu pomnožiti samo ako je broj kolona matrice A jednak broju redova matrice B. Ako je matrica A tipa $p \times q$, a matrica B tipa $q \times r$, onda je rezultat njihovog množenja matrica C tipa $p \times r$. Vrijeme potrebno da se izračuna matrica C je odredjeno brojem množenja skalara koji je jednak pqr.

Raspored zagrada pri množenju matrica ima veliki uticaj na broj množenja skalara i konsekventno, na vrijeme izvodjenja algoritma. Za ilustraciju, razmotrimo proizvod niza od tri matrice $\langle A_1,\ A_2,\ A_3 \rangle$. Pretpostavimo da su tipovi tih matrica 10×100, 100×5 i 5×50, respektivno. Ako množimo prema rasporedu zagrada $((A_1 A_2) A_3)$, tada imamo $10 \cdot 100 \cdot 5$ za množenje matrica $A_1,\ A_2$. Dobivna matrica je tipa 10×5, pa za množenje matricom A_3 treba $10 \cdot 5 \cdot 50$ operacija množenja skalara. Ukupno za dobivanje proizvoda prema ovom rasporedu zagrada treba 7500 množenja skalara.

Ako množimo prema rasporedu zagrada $(A_1 (A_2 A_3))$, tada nam za množenje matrica A_2 i A_3 treba $100 \cdot 5 \cdot 50$ množenja skalara. Dobivena matrica ima tip 100×50 i da bi je pomnožili matricom A_1 treba još $10 \cdot 100 \cdot 50$ množenja skalara. To je ukupno 75000 množenja skalara, što je drastično povećanje u odnosu na množenje pri prethodnom rasporedu zagrada.

Sada možemo formulisati *Problem množenja niza matrica* na sljedeći način: neka je dat niz $\langle A, A_2, \ldots, A_n \rangle$ od n matrica, gdje za $i = 1, 2, \ldots, n$, matrica A_i ima tip $p_{i-1} \times p_i$. Treba naći raspored zagrada pri kojem će broj operacija množenja skalara biti minimalan.

Računanje broja rasporeda zagrada

Algoritam grube sile generiše jedan po jedan raspored zagrada, računa za taj raspored broj operacija množenja i čuva najmanji dobiveni broj sa odgovarajućim rasporedom zagrada. Pokažimo da je ovaj algoritam eksponencijalan.

Označimo sa $P(n)$ broj mogućih rasporeda zagrada za niz od n matrica. Pošto možemo razdvojiti niz matrica izmedju $k - te$ i $k + 1$ matrice, za svako $k = 1, 2, \ldots, n - 1$, a onda nezavisno rasporediti zagrade u dva dobivena podniza, to dobivamo rekurentnu formulu

$$P(n) = \begin{cases} 1 & \text{za } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{za } n \geq 2. \end{cases}$$

Rješenja dobivene rekurentne jednačine su *Catalan-ovi brojevi*

$$P(n) = \frac{1}{n+1}\binom{2n}{n},$$

koji rastu kao $\Omega(4^n/n^{3/2})$. To znači da je algoritam grube sile eksponencijalan i da iscrpno pretraživanje nije dobra strategija za rješavanje našeg problema.

Struktura nekog optimalnog rješenja

Prvi korak u dinamičkom programiranju je nalaženje i opisivanje strukture nekog optimalnog rješenja. Uvodimo zapis $A_{i..j}$ za matricu koja je rezultat računanja proizvoda $A_iA_{i+1}\cdots A_j$. Neki optimalan raspored zagrada za proizvod $A_1A_2\cdots A_n$ dijeli proizvod izmedju A_k i A_{k+1} za neki cio broj k iz intervala $1 \leq k \leq n$. Dakle, za neku vrijednost k, prvo računamo matrice $A_{1..k}$ i $A_{k+1..n}$, a zatim ih množimo da dobijemo proizvod $A_{1..n}$.

Ključno zapažanje je da za ukupan optimalan raspored zagrada svaki učestvujući raspored zagrada podniza $A_1A_2\cdots A_k$ mora biti optimalan raspored zagrada, inače ga možemo zamijeniti boljim. Slično je i sa rasporedom zagrada u $A_{k+1}, A_{k+2}\cdots A_n$.

Znači, optimalno rješenje problema rasporeda zagrada sadrži u sebi optimalna rješenja rasporeda zagrada podproblema.

Rekurzivno rješenje

Drugi korak u dinamičkom programiranju je definisanje vrijednosti optimalnog rješenja preko vrijednosti optimalnih rješenja podproblema. Mi uzimamo za podproblem odredjivanje minimalne cijene za problem rasporeda zagrada u $A_iA_{i+1}\cdots A_j$ za $1 \leq i \leq j \leq n$. Neka je $m[i, j]$ minimalan broj skalarnih množenja potrebnih da se izračuna matrica $A_{i..j}$. Tada će najmanja cijena za matricu $A_{1..n}$ biti $m[1, n]$.

Definišimo veličinu $m[i, j]$ rekurzivno. Ako je $i = j$, tada imamo samo jednu matricu $A_{i..i} = A_i$, tako da množenja nisu potrbna, pa je $m[i, i] = 0$ za

$i = 1, 2, \ldots, n$. Za računanje $M[i, j]$, kada je $i < j$, iskoristićemo strukturu optimalnog rješenja iz prvog koraka. Ako niz podijelimo po nekom $i \leq k < j$, tada imamo podproizvode $A_{i..k}$ i $_{k+1..j}$. Proizvod posljednje dvije matrice zahtijeva $p_{i-1}p_kp_j$ množenja skalara. Zbog toga je

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j,$$

pa se ooptimalna vrijednost definiše na sljedeći način:

$$m[i, j] = \begin{cases} 0 & \text{za } i = j, \\ min_{i \leq k < j}\{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j\} & \text{za } i < j. \end{cases} \tag{2.2}$$

Ako definišemo da $s[i, j]$ bude k za koje je dostignuta optimalna vrijednost $m[i, j]$, tada ćemo moći zadati i optimalan raspored zagrada.

Računanje optimalnog rješenja

Koristeći formulu (2.2)možemo napisati program koji rekurzivno računa minimalnu cijenu $m[1, n]$ za množenje niza matrica $A_1A_2 \ldots A_n$. Na žalost, pokazuje se da i rekurzivni aloritam zahtijeva eksponencijalno vrijeme.

Teškoće nastaju jer imamo neki broj podproblema koje rekurzivni algoritam rješava više puta, na raznim granama rekurzivnog drveta. Za svaki izbor i i j, koji zadovoljavaju $1 \leq i \leq j \leq n$ imamo po jedan podproblem ili ukupno

$$\begin{pmatrix} n \\ 2 \end{pmatrix} + n = \Theta(n^2).$$

Umjesto da rješavamo problem rekurzivno, mi ćemo primijeniti treći korak metode dinamičkog programiranja i računaćemo rješenje metodom odozdo na gore. U sljedećem pseudokodu ćemo podrazumijevati da matrica A_i ima tip $p_{i-1} \times p_i$, za $i = 1, 2, \ldots, n$. Na ulazu u program imamo niz $\langle p_0, p_1, \ldots, p_n \rangle$, dužina je $duzina[p] = n + 1$. Procedura koristi pomoćnu tabelu $m[1..n, 1..n]$ za smještanje vrijednosti $m[i, j]$ i pomoćnu tabelu $s[1..n, 1..n]$ za smještanje vrijednosti indeksa k koji je dobiven prilikom računanja $m[i, j]$.

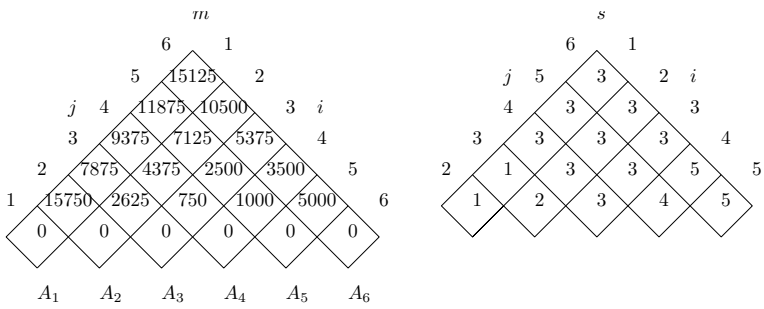
```
PoredakNizaMatrica(p)
n = duzina[p] - 1
for i=1 to n do
       m[i,i] = 0
for l=2 to n do // l je duzina niza
       for i=1 to n-l+1 do
          j=i+l-1
            m[i,j]=infinite
            for k=1 to j-1 do
                  q=m[i,k]+m[k+1,j]+p_{i-1}p_{k}p_{j}
                  if q<m[i,j]
                       then m[i,j]=q
                            s[i,j]=k
return m and s
```

Slika (2.1) ilustruje rad ove procedure na nizu od šest matrica. Tipovi razmatranih matrica su prikazani u tabeli ispod slike. Pošto smo definisali $m[i,j]$ samo za $i \leq j$, onda se koristi samo gornja polovina tabele m. Figuru smo rotirali da glavna dijagonala bude horizontalna tako da se vidi kako se redovi računaju odozdo ka gore.

Konstrukcija optimalnog rješenja

Četvrti korak paradigme dinamičkog programiranja je konstruisanje optimalnog rješenja.Sljedeći rekurzivni program računa proizvod niza matrica za zadate $A = \langle A_1, A_2, \ldots, A_n \rangle$, pri čemu je tabela s sračunata programom $PoredakNizaMatrica$, a indeksi i i j su zadati. Inicijalni poziv programa je $Mno\check{z}enjeNizaMatrica(A, s, 1, n)$.

```
MnozenjeNizaMatrica(A,s,i,j)
if j>i
    then    X=MnozenjeNizaMatrica(A,s,i,s[i,j])
            Y=MnozenjeNizaMatrica(A,s,s[i,j]+1,j)
            return MnozenjeMatrica(X,Y)
    else return Ai
```



Slika 2.1: Tabele m i s sračunate sa PoredakNizaMatrica za n=6 (Prema [9])

Tipovi korištenih matrica $A_1, A_2, A_3, A_4, A_5, A_6$ dati su u sljedećoj tabeli.matrica:

Matrica	A_1	A_2	A_3	A_4	A_5	A_6
Tip	30×35	35×15	15×5	5×10	10×20	20×25

Zadatak 2.1. Problem sječenja cijevi. *Neka je data cijev dužine $n\,cm$ i tabela cijena c_i, $i = 1, 2, \cdots, n$, koje odgovaraju cijevima dužine $i\,cm$, respektivno. Odrediti maksimalnu zaradu z_n koja se može ostvariti sječenjem cijevi i prodajom dijelova. Uzeti u obzir i slučaj u kome se najveća zarada može ostvariti bez ikakvog sječenja cijevi.*

Zadatak 2.2. *U fabrici postoje dvije linije za sklapanje finalnog proizvoda. Svaka linija ima $n \in \mathbb{N}$ radnih mjesta. Radno mjesto j, na liniji i označavamo sa S_{ij}, a*

vrijeme sklapanja na tom radnom mjestu označavamo sa a_{ij}. Kostur proizvoda ulazi u fabriku, ide na liniju $i \in \{1, 2\}$ za vrijeme e_i. Nakon prolaska kroz j-to radno mjesto, proizvod ide na $j+1$-to radno mjesto na jednoj od linija. Ako proizvod ostaje na istoj liniji, onda nema dodatne cijene za transfer. Ako proizvod prelazi na drugu liniju, onda je potrebno vrijeme t_{ij} za transfer na drugu liniju sa radnog mjesta S_{ij}. Nakon izlaska sa n-tog radnog mjesta na liniji, potrebno je još vrijeme x_i da se kompletira proizvod. Treba odrediti koja radna mjesta treba izabrati na liniji 1, a koja na linji 2 da bi se minimizovalo totalno vrijeme prolaska proizvoda kroz fabriku.

2.2 Najduži zajednički podniz

Neka su dati nizovi $X = \langle x_1, x_2, ..., x_m \rangle$ i $Z = \langle z_1, z_2, ..., z_k \rangle$. Kažemo da je Z podniz niza X ako postoji strogo rastući niz $\langle i_1, i_2, ..., i_l \rangle$ indeksa niza X, tako da je za svako $j = 1, 2, ..., l$ u važnosti $x_{i_j} = z_j$.

Primjer 2.1. *Neka su dati nizovi $X = \langle A, B, C, B, D, A, B \rangle$ i $Z = \langle B, C, D, B \rangle$. Niz Z je podniz niza X sa odgovarajućim indeksima $\langle 2, 3, 5, 7 \rangle$.*

Za dva niza X i Y niz Z je zajednički podniz, ako je Z podniz od oba niza X i Y.

Primjer 2.2. *Neka su dati nizovi $X = \langle A, B, C, B, D, A, B \rangle$ i $Y = \langle B, D, C, A, B, A \rangle$. Niz $Z = \langle B, C, A \rangle$ je podniz i niza X i niza Y. Niz $\langle B, C, A \rangle$ nije najduži zajednički podniz, jer je niz $\langle B, C, B, A \rangle$ zajednički i ima dužinu 4. Pri tome je $\langle B, C, B, A \rangle$ najduži zajednički podniz nizova X i Y, jer ne postoji zajednički podniz dužine 5 ili više.*

LCS problem. Dati su nizovi $X = \langle x_1, x_2, ..., x_m \rangle$ i $Y = \langle y_1, y_2, ..., y_n \rangle$. Naći najduži zajednički podniz Z nizova X i Y. (Problem najdužeg zajedničkog podniza, *Longest Common Subsequence* problem)

2.2.1 Osobine najdužeg zajedničkog podniza

Pokušajmo sa algoritmom grube sile za rješavanje LCS problema. Numerisati sve podnizove niza X i probati koji je od njih najduži i zajedniči sa Y podnizom. Postoji 2^m podnizova od X, pa algoritam zahtijeva eksponencijalno vrijeme, što ga čini nepaktičnim.

Za efikasno rješavanje problema definišemo i-ti prefiks od $X, i = 0, 1, ..., m$ kao $X_i = \langle x_1, x_2, ..., x_i \rangle$. Na primjer, za $X = \langle A, B, C, B, D, A, B \rangle$ je $X_4 = \langle A, B, C, B \rangle$.

Teorema 2.1. *Neka je $X_i = \langle x_1, x_2, ..., x_m \rangle$ i $Y_i = \langle y_1, y_2, ..., y_n \rangle$ i neka je $Z_i = \langle z_1, z_2, ..., z_k \rangle$ LCS za X i Y.*

1. *Ako je $x_m = y_n$ tada je $z_k = x_m = y_n$ i $Z_{k-1} = LCS(X_{m-1}, Y_{n-1})$*
2. *Ako je $x_m \neq y_n$ tada iz $z_k \neq x_m$ slijedi da je $Z = LCS(X_{m-1}, Y)$*
3. *Ako je $x_m \neq y_n$ tada iz $z_k \neq y_n$ sledi da je $Z = LCS(X, Y_{n-1})$*

Dokaz

1. Ako je $z_k \neq x_m$, tada možemo dodati $y_n = x_m$ na Z i dobiti zajednički podniz X i Y dužine $k+1$, što je suprotno pretpostavci $x_m = y_n = z_k$. Pretpostavka $Z_{k-1} \neq LCS(X_{n-1}, Y_{m-1})$ lako dovodi do kontradikcije.

2. Ako je $z_k \neq x_m$, tada je Z zajednički podniz od X_{m-1} i Y. Ako bi postojao zajednički podniz X_{m-1} i Y sa dužinom većom od k, tada bi taj podniz W bio zajednički podniz X_m i Y, što je suprotno pretpostavci

3. Simetrično dokazu iz tačke 2. □

Rekurzivno rješenje problema

Na osnovu prethodnih razmatranja, definišemo cijenu nalaženja najdužeg zajedničkog podniza. Cijenu definišemo kao dužinu LCS od X_i i Y_j :

$$c[i,j] = \begin{cases} 0 & \text{, za } i = 0 \text{ ili } j = 0 \\ c[i-1, j-1] + 1 & \text{, za } i,j > 0,\ x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{za } i,j > 0, x_i \neq y_j \end{cases}$$

U definiciji je intencija da, u svakom koraku, iz razmatranja isključimo dva podproblema koji ne odgovaraju uslovu ispunjenom u tom koraku. Takvo isključivanje podproblema je karakteristično za dinamičko programiranje.

Računanje dužine LCS

Dajemo proceduru za računanje koja vraća tabele b i c za računanje LCS nizova X i Y. Iako problem ima samo $\theta(mn)$ podproblema, koristićemo dinamičko programiranje i postupkom odozdo na gore konstruisati optimalno rješenje. Program je napravljen prema definiciji iz prethodne sekcije.

```
LCS_LENGTH (X,Y)
m <- length [X]
n <- length [Y]
for i<- 1 to m
        do c[i,0] <-0
for j <- 1 to n
    do c[0,j] <- 0
for i<-1 to m
        do for j<-1 to n
        do if xi = yj
                then c[i,j] <-c[i-1,j-1]+1
                        b[i,j] <- "\\"
                else if c[i-1,j]>=c[i,j-1]
                                then c[i,j] <- c[i-1,j]
                                        b[i,j] <- "|"
                else c[i,j] <- c[i,j-1]
                                        b[i,j] <-"<-"
return c and b
```

Program ima vremensku složenost $\Theta(mn)$, jer se svaki elemenat tabele računa u konstantnom vremenu.

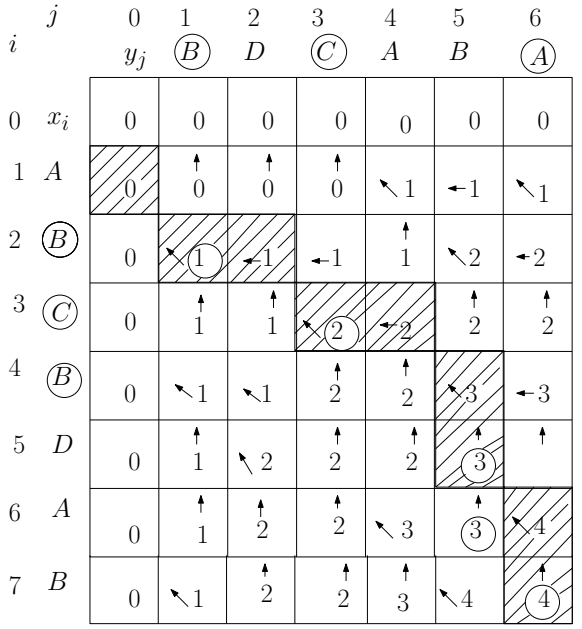
Konstrukcija LCS

Sljedeći program koristi tabelu $b[i, j]$ iz prethodnog programa i konstruiše LCS nizova X i Y.

```
PRINT_LCS(b,X,i,j)
if i=0 or j=0
    then return
if b[i,j] = "\"
    then PRINT_LCS(b,X,i-1,j-1)
    print xi
else if b[i,j] = "|"
    then PRINT_LCS(b,X,i-j,j)
else PRINT_LCS(b,X,i,j-i)
```

Program ima složenost $O(m + n)$, jer umanjuje najmanje jedan od i i j u svakom pozivu. Primjećujemo da se tabela $b[i, j]$ može izostaviti, jer se elemenat LCS može odrediti gledajući preko koja tri prethodna člana je izračunato $c[i, j]$.

Na slici (2.2) je dat primjer konstrukcije LCS za nizove $[A, B, C, B, D, A, B]$ i $[B, D, C, A, B, A]$.



Slika 2.2: Konstrukcija LCS (Prema [9])

2.3 Optimalna triangulacija poligona

Dat je konveksan poligon $P = \langle v_0, v_1, ..., v_{n-1} \rangle$ i težinska funkcija w definisana na trouglovima formiranim stranama i tetivama P. Problem je naći triangulaciju koja minimizira težinu trouglova u triangulaciji. **Primjer težine:**
Jedan od načina da dobijemo težinu trougla je da težinsku funkciju definišemo kao

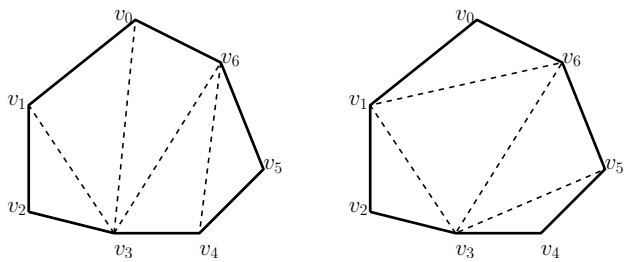
$$\omega(\Delta v_i v_j v_k) = \mid v_i v_j \mid + \mid v_j v_k \mid + \mid v_k v_i \mid,$$

gde je $\mid v_i v_j \mid$ Euklidovo rastojanje tacaka v_i i v_j.
Na slikama 2.3 i 2.4 dati su primjeri triangulacije.

Triangulacija se često primjenjuje u računarstvu. U procesiranju slika se metod triangulacije koristi za rekonstrukciju slike. Geometrijski modeli u računarskoj grafici su često predstavljeni kao triangulirane površi. Konstruisan je hardver za brzo i jeftino renderisanje i sjenčenje trouglova. Cijena postupka je neznatno viša od cijene učitavanja trouglova. Triangulacija se koristi i u kombinatornoj topologiji, gdje se poliedri i uopšte topološke mnogostrukosti, opisuju njihovom triangulacijom. Problem homeomorfizma je problem nalaženja algoritma koji za dvije topološke mnogostrukosti, zadate njihovom triangulacijom, nalazi jesu li homeomorfne. U nekim specijalnim slučajevima, ako uvedemo neka ograničenja, na primjer, fiksiramo dimenziju, ili fiksiramo početni elemenat para, onda problem homeomorfizma može biti riješen. Dokazano je da ne postoji rješenje problema homeomorfizma poliedara (topoloških mnogostrukosti), u opštem slučaju. Problem homeomorfizma za trodimenzionalne poliedre je otvoren problem.

Napominjemo da je opšti problem triangulacije skupova tačaka, a takodje i problem triangulacije grafova, jako **NP**-kompletan. Oba problema imaju značajne primjene, na primjer u računarskoj geometriji, ali ih ovaj put nećemo izučavati. Navodimo samo formulaciju posljednjeg problema, koja će nam u daljem izlaganju poslužiti za dokazivanje jake **NP**-kompletnosti.

Primjer 2.3. Triangulacija grafova. *Neka je $G = (V, E)$ neusmjereni graf, pri čemu su V vrhovi, a G grane grafa. Neka je $|V| = 3q$ prirodan broj. Pitanje je da li postoji podjela skupa V na q disjunktnih podskupova $V_1, V_2, \ldots, V_q$, od kojih svaki sadrži tačno tri vrha, pri čemu za svaki podskup $V_i = \{v_{[i1]}, v_{[i2]}, v_{[i3]}\}$ lukovi $(v_{[i1]}, v_{[i2]})$, $(v_{[i2]}, v_{[i3]})$, $(v_{[i3]}, v_{[i1]})$ pripadaju skupu grana grafa E?*

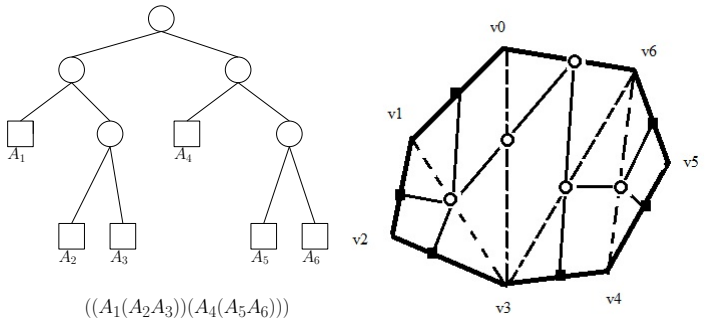


Slika 2.3: Prikaz dvije triangulacije sedmougla

Korespondencija triangulacije poligona sa postavljanjem zagrada pri množenju niza matrica

Zanimljivo je primijetiti da jedna optimalna triangulacija konveksnog poligona P, u odnosu na težinsku funkciju, daje "parsing" drvo za optimalno razmještanje zagrada pri množenju niza matrica $A_1A_2...A_n$. Ovo postaje jasno ako pridružimo matrice stranicama $(0,1),(1,2),(2,3),\ldots,(n-1,n)$ nekog poligona, a svakoj tetivi (i,j) pridružimo proizvod matrica koje su ispod ove tetive. Pridruživanje je ilustrovana na slici 2.4.

Zadatak 2.3. *Naći kontraprimjer i obrazložiti zašto obrnuto tvrdjenje nije tačno.*



Slika 2.4: Optimalna triangulaciji i postavljanja zagrada pri množenju niza matrica

Struktura rješenja

Triangulacija podpoligona mora biti optimalna, inače, mogli bismo poboljšati globalnu strukturu (triangulaciju). Ova činjenica omogućava primjenu metode dinamičkog programiranja i rješavanje problema triangulacije odozdo na gore.

Rekurzivno rješenje

Svaka stranica (i,j) poligona uključena je u tačno jedan trougao triangulacije. Taj trougao $\Delta v_i v_j v_k$ se dobiva biranjem nekog tjemena k, $(i+1 \leq k \leq j-1)$, poligona. Ako sa $t(i,j)$ označimo minimalnu cijenu triangulacije poligona nad stranicom (i,j), a sa $t(i,k)$ i $t(j,k)$ minimalne cijene triangulacije poligona nad tetivama (i,k) i (k,j), respektivno, tada je

$$t[i,j] = \begin{cases} 0, & ako\ je\ i = j \\ min_{i+1\leq k\leq j-1}\{t[i,k] + t[k,j] + \omega(\Delta v_i v_k v_j)\}, & \text{za } i < j \end{cases}$$

Pseudo-kod za navedeni algoritam dat je u sljedećem primjeru.

for $i = 1$ **to** $n-1$ **do** $t[i,i+1]$
for $skok = 2$ **to** $n-1$
  for $i = 1$ **to** $n - skok$ **do**
    $j = i + skok$

$$t[i,j] = \min_{k=i+1}^{j-1}(t[i,k] + t[k,j] + \omega(\Delta v_i v_k v_j))$$

return $t[1,n]$

Zadatak 2.4. *Dokazati da je vremenska složenost navedenog algoritma triangulacije* $O(n^3)$*, a da je prostorna složenost* $O(n^2)$*.*

2.4 Parsiranje kontekstno slobodnih gramatika

Kompajleri ispituju je li program legalan u datom programskom jeziku, a ako nije, vraćaju nam greške. Ovo zahtijeva precizan opis jezika, na primjer, kontekstno slobodnom gramatikom. Parsiranje datog teksta S u odnosu na datu kontekstno slobodnu gramatiku G je algoritamski problem konstruisanja drveta parsiranja shodno pravilima substitucije koja definišu S kao jedinstven ne-terminalni simbol G.

Pretpostavljamo da tekst S ima dužinu $n \in \mathbb{N}$ i da gramatika G ima konstantnu veličinu. Takav je slučaj sa gramatikama koje definišu programske jezike, kao što su C, ili $Java$, bez obzira na veličinu programa koje želimo kompajlirati.

Dalje ćemo pretpostaviti da je definicija svakog pravila gramatike u normalnoj formi Chomskog. Ovo znači da se desne strane svakog netrivijalnog pravila sastoje od tačno dva ne-terminalna simbola, na primjer $X \to YZ$, ili tačno jednog terminalnog simbola, na primjer, $X \to \alpha$. Svaka kontekstno-slobodna gramatika može se svesti na normalnu formu Chomskog skraćivanjem desnih strana pravila na račun dodavanja novih ne-terminalnih simbola. Prema tome, nema gubitka opštosti uz naše pretpostavke.

Ključno zapažanje za razvoj algoritma je da primjena pravila $X \to YZ$ na string S, dijeli S na nekoj poziciji i tako da lijevu stranu stringa $S[1,i]$, treba generisati simbolom Y, a desnu stranu $S[i+1,n]$ simbolom Z. Pri tome čuvamo terminalne simbole generisane u bilo kojem podproblemu.

Ako sa $B[i,j,X]$ označimo Booleovu funkciju koja je tačna ako i samo ako je string $S[i,j]$ generisan simbolom X, onda možemo napisati logičku formulu

$$B[i,j,X] = \bigvee_{(X \to YZ) \in G} (\bigvee_{i=k}^{j} B[i,k,Y] \cdot B[k,j,Z])$$

Ovdje je $\bigvee$ logička disjunkcija, a $\cdot$ je logička konjunkcija.

Zadatak 2.5. *Dokazati da je vremenska složenost navedenog algoritma parsiranja kontekstno slobodnih gramatika* $T(n) = O(n^3)$*.*

2.5 Optimalno binarno drvo za pretraživanje

Ulaz: lista riječi sa fiksnim vjerovatnoćama pojavljivanja $p_1, p_2, ..., p_n$.
Problem: Aranžirati ove riječi u binarno drvo pretraživanja tako da minimiziramo očekivano vrijeme pretraživanja.

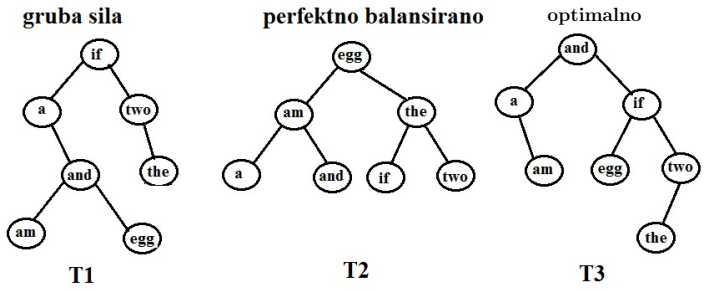
Pošto je broj poredjenja do pristupa elementu na dubini d jednak $d+1$, treba minimizirati $\sum_{i=1}^{n} p_i(1+d_i)$.

$rijec\, w_i$	$vjerovatnoca\, p_i$	$broj$	niz	$broj$	niz	$broj$	niz
a	0.22	2	0.44	3	0.66	2	0.44
am	0.18	4	0.72	2	0.36	3	0.54
and	0.20	3	0.60	3	0.60	1	0.20
egg	0.05	4	0.20	1	0.05	3	0.15
if	0.25	1	0.25	3	0.75	2	0.50
the	0.02	3	0.06	2	0.04	4	0.08
two	0.08	2	0.16	3	0.24	3	0.24
	1.00		2.43		2.70		2.15

Ako se struktura problema predstavi drvetom, onda je lako dobiti formulu za cijenu drveta optimizacije:

$$C_{left,right} = min\{p_i + C_{left,i-1} + C_{i+1,right} + \sum_{j=left}^{i-1} p_j + \sum_{j=i+1}^{right} p_j\} = min\{C_{left,i-1} + C_{i+1,right} + \sum_{j=left}^{right} p_j\}$$

Na slici (2.5) predstavili smo drveta $T1$, koje odgovara algoritmu grube sile, $T2$, koje je prerfektno balansirano i $T3$, koje je optimalno. Drveta $T1, T2, T3$, odgovaraju trećoj i četvrtoj, petoj i šestoj, te sedmoj i osmoj koloni tabele, respektivno.



Slika 2.5: Različita drveta za pretraživanje

2.6 Najkraći putevi medju svim parovima vrhova grafa

Neka je dat graf $G = (V, E)$, gdje smo sa V označili skup vrhova, a sa E skup grana grafa. Pretpostavimo da za rješenje problema koristimo nizove.

Za traženje najkraćeg puta medju svim parovima vrhova grafa mogli bismo koristiti algoritam Dijkstra (E. W. Dijkstra) sa jedinstvenim izvorom tako što bismo ga startovali posebno za svaki vrh kao izvor.

Algoritam Floyd-Warshall za najkraći put izmedju svih parova vrhova grafa ima vremensku složenost $O(n^3)$, što asimptotski nije bolje od n poziva algoritma Dijkstra. Ipak, petlje su preciznije, a program kraći tako da ovaj algoritam bolje radi u praksi. Floyd-Warshall algoritam je poseban i po tome što je jedan od rijetkih algoritama na grafovima koji bolje radi na matrici susjedstva nego na listi susjedstva.

Rekurzivno rješenje problema se zadaje sa:
$D_{k,i,j} = min\{D_{k-1,i,j}, D_{k-1,i,k} + D_{k-1,k,j}\}$
$D_{0,i,j} = c_{ij}$, za $(i,j) \in G$
$(v_i, v_j) \notin G \Rightarrow D_{ij} = +\infty$
U formuli smo označili sa $D_{|V|,i,j}$ najkraći put izmedju v_i i v_j, to jest put koji je dobiven korištenjem svih vrhova grafa V, a sa $D_{k,i,j}$ najkraći put izmedju v_i i v_j koji koristi samo vrhove v_1, v_2, ..., v_k. Pri tome c_{ij} označava cijenu grane (v_i, v_j) u matrici cijena C, koja je tipa $|V| \times |V|$. U slučaju da $(v_i, v_j) \notin E$ stavljamo da je $c_{ij} = \infty$.

Neka je dat cijenama grana labelisan i usmjeren graf $G = (V, E)$. Za svaki par vrhova $u, v \in V$ treba naći cijenu njkraćeg puta iz u do v u grafu G. Dajemo standardni algoritam dinamičkog programiranja za nalaženje najkraćeg puta medju svim vrhovima u, v grafa G (algoritam Floyd-a). Neka je $n = |V|$.

```
function floydwarshall(C, n){
        for(i = 1, i <= n, i++){
                for(j = 1; j <= n; j++)
                        if(i, j) ∈ E
                                D[i][j] = c_ij;
                        else
                                D[i][j] = ∞;
                A[i][i] = 0; }
for(k = 1; k <= n; k++)
        for(i = 1; i <= n; i++)
                for(j = 1; j <= n; j++)
                        if D[i][k] + D[k][j] < D[i][j] then
                         D[i][j] = D[i][k] + D[k][j];
return(D); }
```

2.7 Fibonačijevi (Fibonacci) nizovi

Zadatak 2.6. *Fibonačijevi nizovi se definišu kao:* $a_1 = 1$, $a_2 = 1$, $\ldots$, $a_n = a_{n-1} + a_{n-2}$. *Pokazati kako izračunati* a_n *koristeći* $O(\log n)$ *operacija. Ne koristiti opšte formule.* **Uputa:**

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_{n-1} \\ a_{n-2} \end{bmatrix} = \begin{bmatrix} a_n \\ a_{n-1} \end{bmatrix}$$

Zadatak 2.7. *Uporediti rekurzivnu i iterativnu varijantu programa za računanje Fibonačijevih brojeva i objasniti prednosti dinamičkog programiranja u tom slučaju.*

Glava 3

Greedy (gramzivi) algoritmi

Metod dinamičkog programiranja rješava probleme optimizacije tako što prolazi kroz niz koraka i na svakom koraku ima skup izbora. Dinamičko programiranje razmatra sve ove izbore, ali za razliku od rekurzivnog programiranja, ako su za neki izbor računanja već izvedena, onda se ona ne ponavljaju već se koriste ranije zapamćeni rezultati. U mnogim primjerima je dinamičko programiranje efikasno za nalaženje najboljeg izbora.

Greedy algoritmi prave izbor koji izgleda najbolji u datom momentu. Oni uzimaju lokalno najbolji izbor, očekujući da će on dovesti i do globalno najboljeg izbora. Greedy algoritmi ne vode uvijek do optimalnih rješenja, ali su efikasni na velikom broju problema važnih u praksi.

3.1 Gramzivi algoritmi i problemi optimizacije

Postupak: Neki "lokalni optimum" je izabran i kada se algoritam završi očekujemo da je lokalni optimum jednak "globalnom optimumu". Ako se ne traži apsolutno najbolji odgovor, tada se greedy algoritam koristi za generisanje približnog odgovora. U slučaju da se traži tačan odgovor, moramo koristiti komplikovanije algoritme.

Svakodnevni primjeri:

1. Razmjena novčanice.
2. Začepljenje u saobraćaju (Traffic jam): da li ući u ulicu u kojoj vas možda nakon kilometar očekuje začepljenje ili se vratiti $5km$ nazad da bi se izbjeglo začepljenje.
3. Pravljenje rasporeda procesa na jednom ili više procesora.

Ideju na kojoj se baziraju gramzivi algoritmi ilustrujemo rješavajući sljedeća dva zadatka.

Zadatak 3.1. *Neka je zadata matrica A sa realnim, ne-negativnim elementima:*

$$A = \begin{bmatrix} 8 & 6 & 2 \\ 4 & 5 & 4 \\ 3 & 4 & 2 \end{bmatrix}.$$

Treba izabrati podskup elemenata matrice tako da

(a) u svakoj koloni se nalazi najviše jedan izabrani elemenat

(b) suma izabranih elemenata je najveća moguća.

Zadatak rješavamo tako što biramo najveći element koji možemo, a da pri tome ne narušavamo uslov (a). Zaustavljamo se kada ne bude moguće dobavljanje nekog elementa, a da pri tome ne narušimo uslov (a). Algoritmi takvog tipa nazivaju se gramzivi algoritmi.

Za zadanu matricu algoritam nalazi da je rješenje $8 + 6 + 4$.

$$A = \begin{bmatrix} \mathbf{8} & \mathbf{6} & 2 \\ 4 & 5 & \mathbf{4} \\ 3 & 4 & 2 \end{bmatrix}.$$

Nije teško vidjeti da za proizvoljnu matricu sa ne-negativnim elementima, algoritam nalazi podskup koji zaista daje optimalno rješenje.

Naš drugi zadatak je malo drugačiji od prethodnog.

Zadatak 3.2. *Neka je zadata matrica A sa realnim, ne-negativnim elementima:*

$$A = \begin{bmatrix} 8 & 6 & 2 \\ 4 & 5 & 4 \\ 3 & 4 & 2 \end{bmatrix}.$$

Treba izabrati podskup elemenata matrice tako da

(a) u svakoj koloni i u svakom redu se nalazi najviše jedan izabrani elemenat

(b) suma izabranih elemenata je najveća moguća.

Primjenjujući gramzivi algoritam nalazimo rješenje $8 + 5 + 2$.

$$A = \begin{bmatrix} \mathbf{8} & 6 & 2 \\ 4 & \mathbf{5} & 4 \\ 3 & 4 & \mathbf{2} \end{bmatrix},$$

koje nije pravilno, jer je $8 + 4 + 4 = 16$ veći zbir,

$$A = \begin{bmatrix} \mathbf{8} & 6 & 2 \\ 4 & 5 & \mathbf{4} \\ 3 & \mathbf{4} & 2 \end{bmatrix}.$$

Postavlja se pitanje kada će gramzivi algoritam dati tačan rezultat. Formulišimo naša prethodna dva zadatka u opštijem obliku.

Zadatak 3.3. *Neka je dat konačan skup E, familija njegovih podskupova $\mathcal{I} \subseteq \mathcal{P}(\mathcal{E})$ i funkcija $\omega : E \to \mathbb{R}^+$, gdje je $\mathbb{R}^+$ skup ne-negativnih realnih brojeva. Naći podskup $S \in \mathcal{I}$ sa najvećom sumom $\sum_{e \in S} \omega(e)$.*

Lako je vidjeti da su zadaci 3.1 i 3.2 specijalni slučajevi zadatka 3.3. Sada naše pitanje možemo formulisati tako da pitamo koje uslove treba zadovoljavati familija $\mathcal{I}$ da bi gramzivi algoritam davao tačno rješenje zadatka 3.3

Odgovor je da je par $< E, \mathcal{I} >$ matroid.

Definicija 3.1. *Neka je E konačan skup, a $\mathcal{I} \subseteq \mathcal{P}(\mathcal{E})$ familija koja zadovoljava uslove*

(M1) $\emptyset \in \mathcal{I}$

(M2) Ako je $A \in \mathcal{I}$, i $B \subseteq A$, onda je i $B \in \mathcal{I}$

(M3) Za proizvoljne $A, B \in \mathcal{I}$, takve da je $|B| = |A| + 1$, postoji elemenat $e \in B \setminus A$, tako da je $A \cup \{e\} \in \mathcal{I}$.

*Par $M =< E, \mathcal{I} >$ naziva se **matroid**.*

Skupove iz $\mathcal{I}$ nazivamo nezavisnim, a skupove iz $\mathcal{P}(\mathcal{E}) \setminus \mathcal{I}$ zavisnim, po analogiji sa linearnom algebrom. Edmonds J. i Rado R. su dokazali (vidi [10], [22]) ako model problema ima strukturu matroida, onda greedy algoritmi uvijek vode do globalnog rješenja problema. Ako model problema nije matroid, onda postoji funkcija $\omega : E \to \mathbb{R}^+$ takva da S nije nezavisan skup sa najvećom težinom.

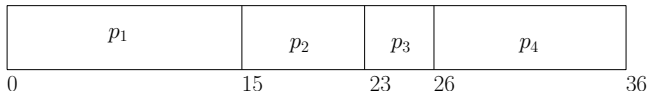
3.2 Minimizacija vremena čekanja procesa na jednom ili više procesora

Pravljenje rasporeda procesa na jednom ili više procesora razmatramo detaljnije. U principu su problemi pravljenja rasporeda NP-kompletni.

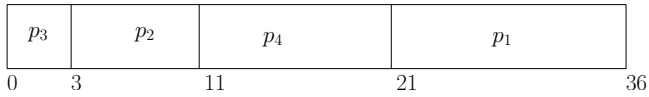
Zadatak 3.4. *Dati su poslovi $p_1, p_2, ..., p_n$, koji se završavaju za vrijeme $t_1, t_2, ..., t_n$ respektivno. Imamo jedan procesor. Koji je najbolji način da se rasporede ovi poslovi tako da je srednje vrijeme čekanja minimalno?*

$posao$	$vrijeme$
$p1$	15
$p2$	8
$p3$	3
$p4$	10

Prosječno vrijeme čekanja, ako se poslovi izvode redoslijedom (p_1, p_2, p_3, p_4), je $t_{sr} = \dfrac{15 + 23 + 26 + 36}{4} = 25$ (Slika 3.1). Optimalno prosječno vrijeme čekanja $t_{sr} = \dfrac{3 + 11 + 21 + 36}{4} = 17.75$, ako se poslovi izvode u redoslijedu (p_3, p_2, p_4, p_1) (Slika 3.2).



Slika 3.1: Jedan prikaz rasporeda poslova



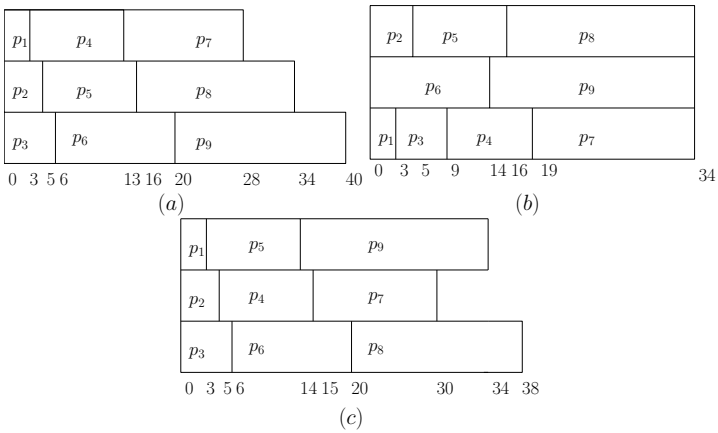
Slika 3.2: Drugi prikaz rasporeda poslova

Rješenje problema. Aranžiranje po najkraćim vremenima čekanja uvijek daje optimalno rješenje.
Dokaz. Neka je raspored $p_{i1}, p_{i2}, ..., p_{in}$. Prvi posao završava u vremenu t_{i1}. Drugi posao završava u vremenu $t_{i1} + t_{i2}$, treći $t_{i1} + t_{i2} + t_{i3}$,...
$C = \sum_{k=1}^{n}(n-k+1)t_{ik} = (n+1)\sum_{k=1}^{n} t_{ik} - \sum_{k=1}^{n} kt_{ik}$ Ako je stavljeno x iza y uz $t_{ix} < t_{iy}$, tada razmjenom x i y smanjujemo totalno vrijeme čekanja. Zbog toga je svaki raspored po rastućim vremenima *suboptimalan*. □

Više procesora. Slično kao u slučaju jednog procesora dokazujemo da se optimalna cijena postiže ako prvom procesoru damo posao sa najmanjim vremenom čekanja, drugom procesoru posao sa najmanjim vremenom čekanja u preostalim poslovima, i tako redom, dok ne iscrpimo sve poslove.

$posao$	$vrijeme$
$p1$	3
$p2$	5
$p3$	6
$p4$	10
$p5$	11
$p6$	14
$p7$	15
$p8$	18
$p9$	20

Minimizacija konačnog vremena završavanja. Kada će i poslednji posao biti završen? Prema slici 3.3, u primjerima a i c vrijeme završavanja je 40 i 38, respektivno. Ovaj problem ekvivalentan je problemu pakovanja ranca, pa je prema tome NP-kompletan (tj. problem minimizacije konačnog vremena). U slučaju b sa slike 3.3, minimalno vrijeme izvršavanja ne može biti poboljšano, jer su svi procesori uvijek zauzeti.



Slika 3.3: Prikaz rasporeda poslova na više procesora

3.3 Huffman-ovi kodovi. Huffman-ov algoritam.

Skup ASCII sastoji se od 100 printabilnih karaktera. $\lceil \log_2 100 \rceil = 7$, pa je potrebno 7 bitova za reprezentaciju. Pošto je $2^7 = 128$, imamo i neke neprintabilne karaktere. Osmi bit je dodat za kontrolu parnosti.

Primjer 3.1. *Datoteka sadrzi* $a, e, i, s, t, space, newline.$

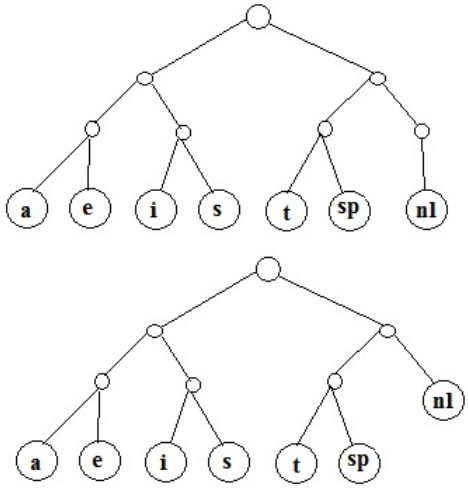
Karakter	Kod	Frekvencija	Broj bitova
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
space	101	3	39
new line	110	1	3
total			174

Problem. Zainteresovani smo za redukovanje dužine datoteke zbog prenosa telefonskom linijom ili zbog smanjenja veličine prostora za zapis datoteke na disku. Je li moguće uvesti novi kod i tako smanjiti dužinu datoteke i ukupan broj potrebnih bita?

Odgovor. Da! To je moguće i možemo uštedjeti i 25 procenata, a na velikim datotetkama i do 50 - 60 procenata prostora.

Opšta strategija je da se dozvoli da dužina koda varira od karaktera do karaktera i omogućiti da karakteri sa visokim frekvencijom imaju kraći kod. Ako nam se svi karakteri pojavljuju sa istom frekvencijom, tada vjerovatno nećemo imati velike uštede.

Predstavljanje koda drvetom



Slika 3.4: Predstavljanje koda iz primjera 3.1 drvetom

Neka grana lijevo znači 0, a grana desno znači 1. Na primjer, na slici 3.4, 010 predstavlja "i". Cijena koda je $\sum d_i f_i$, gdje karakter c_i ima dubinu d_i i frekvenciju f_i. Jeftiniji je donji kod na slici. Kod kod kojeg su vrijednosti samo u listovima, naziva se *prefiksni kod.*

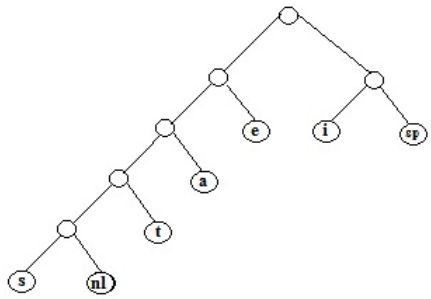
Zadatak. Izračunati cijenu kodova u dva drveta data na slici 3.4.

Primjedba. Optimalan kod je predstavljen potpunim drvetom, inače jednog potomka možemo dići na viši nivo i tako smanjiti cijenu, kao u donjem drvetu na slici 3.4.

Ako su znaci samo na listovima drveta, tada svaka sekvenca može biti kodirana na jedinstven način. Na primjer, pretpostavimo da je kodirani string 01001111000101100010000111. Slijedi da 0 nije znak, 01 nije znak, ali 010 predstavlja i, pa je prvi karakter i. Tada je 011 s, 11 je nl, a zatim slijede a, sp, t, i, e, nl. Nijedan od znakova nije prefiks nekog drugog koda znaka. Ako se karakteri pojavljuju i u čvorovima, tada dekodiranje nije jednožnačno.

Naš problem se svodi na nalaženje *Kompletnog binarnog drveta sa minimalnom totalnom cijenom,* pri čemu se svi znakovi nalaze na listovima.

Primjer 3.2. Optimalan prefiksni kod

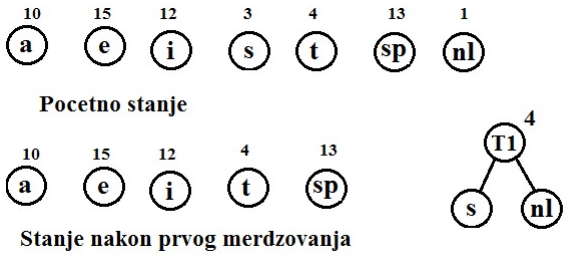


Slika 3.5: Drvo optimalnog koda

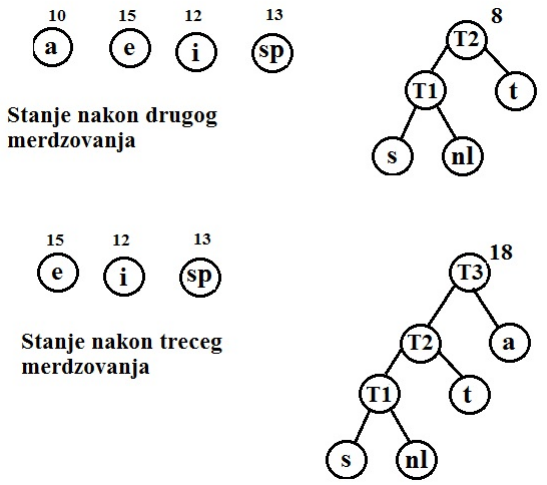
Karakter	Kod	Frekvencija	Broj bitova
a	001	10	30
e	01	15	30
i	10	12	24
s	00000	3	15
t	0001	4	16
space	11	13	26
new line	00001	1	5
total			146

Razmjenom djece u drvetu, možemo dobiti različite optimalne kodove.

Huffman-ov algoritam.
Huffmanov algoritam ilustrujemo na slikama 3.6, 3.7, 3.8 i 3.9. Neka je C broj različitih znakova. Svaki znak predstavljamo drvetom i razmatramo šumu drveta. Težina drveta jednaka je zbiru frekvencija njegovih listova. $C-1$ puta izaberemo dva drveta T_1 i T_2 sa najmanjom težinom (u slučaju jednakih težina uzimamo proizvoljno) i formiramo novo drvo sa poddrvetima T_1 i T_2. Inicijalno (ulaz) je C šuma drveta sa jednim čvorom za svaki znak. Na kraju (izlaz) postoji jedinstveno drvo i to je *optimalno Huffman-ovo* drvo kodiranja.



Slika 3.6: Početno stanje i stanje nakon prvog merdžovanja



Slika 3.7: Stanje nakon drugog i trećeg merdžovanja

Ideje za dokaz korektnosti Huffman-ovog algoritma:

1. Drvo mora biti potpuno, inače možemo jedan list podići na nivo za jedan viši i smanjiti cijenu.

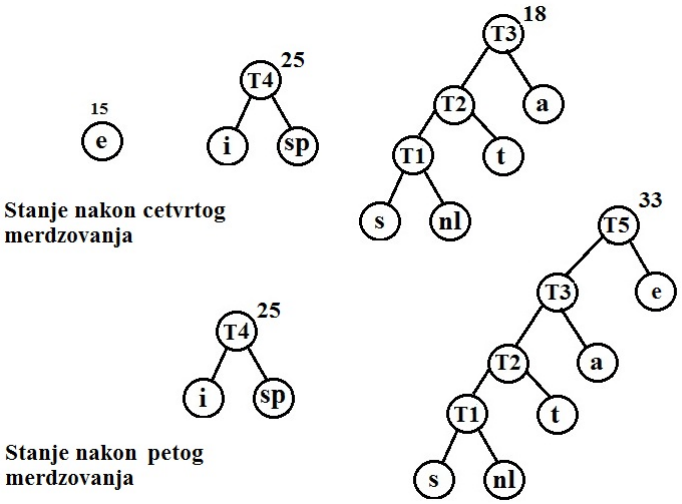
2. Dva najmanje frekventna karaktera α i β moraju biti najdublji listovi. Dokaz kontradikcijom: Ako α i β nisu najdublji listovi, tada to mora biti neko γ, jer je drvo kompletno. Razmjena (Swaping) γ sa α i β smanjuje cijenu drveta!

3. Dva lista iste dužine mogu zamijeniti mjesta ne narušavajući optimalnost.

4. Inicijalni korak je dobar.

5. Indukcija kod spajanja drveta. **Skica dokaza.**
   Dokaz korektnosti Huffman-ove konstrukcije je primjer korištenja matematičke indukcije primijenjene na drveta. Lako se može prilagoditi za dokaz da je rekurzivni program za Huffman-ov kod totalno korektan.

   Pretpostavimo da su težine $w_1, w_2, \ldots, w_n$ i da su indeksirane tako da je $w_1 \leq w_2 \leq \ldots \leq w_n$. Pošto optimalno Huffman-ovo drvo ima $n-1$ unutrašnjih čvorova i n listova, a svaki čvor ima nula ili dva sljedbenika, slijedi da optimalno drvo mora biti potpuno binarno drvo. Optimalno binarno drvo uvijek može biti nadjeno, jer je broj potpunih binarnih drveta sa n listova, konačan. Jasno je da Huffman-ova konstrukcija daje optimalna drveta za $n = 1$ i $n = 2$. To predstavlja bazu indukcije po broju n težina ili listova. Induktivna hipoteza je da Huffman-ova konstrukcija daje optimalna drveta za svakih $n-1$ težina.



Slika 3.8: Stanje nakon četvrtog i petog merdžovanja

Neka je dato neko drvo Tn sa n listova, u kojem je čvor $w_1 + w_2$, sa listovima w_1 i w_2, poddrvo. Neka je T'_{n-1} drvo sa $n-1$ listova, dobiveno iz T_n zamjenom pomenutog poddrveta listom $w_1 + w_2$. Tada je dužina puta sa težinama u T_n jednaka dužini puta sa težinama od $T'_{n-1} + (w_1 + w_2)$.

Obrnuto, ako je dato T'_{n-1}, tada T_n možemo dobiti iz njega. Dakle, T_n mora biti optimalno u odnosu na težine $w_1, w_2, \ldots, w_n$ ako i samo ako je T'_{n-1} optimalno u odnosu na težine $(w_1 + w_2), w_3, \ldots, w_n$. Ovo slijedi iz činjenice da neoptimalnost jednog drveta povlači mogućnost da ono može biti zamijenjeno optimalnim, a to bi popravilo dužinu puta sa težinama drugog drveta.

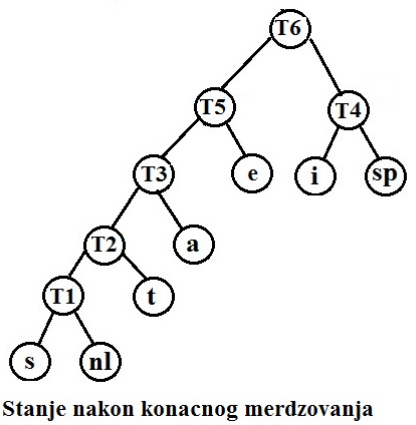
Po induktivnoj hipotezi Huffman-ova konstrukcija daje drvo T'_{n-1} koje ima $n-1$ listova i optimalno je u odnosu na težine $(w_1 + w_2), w_3, \ldots, w_n$. Odgovarajuće drvo T_n, sa n listova, je drvo dato Huffman-ovom konstrukcijom za težine $w_1, w_2, \ldots, w_n$. Pošto je T'_{n-1} optimalno, to je i T_n optimalno. □

Pitanje. Zašto je ovaj algoritam greedy?
Odgovor. U svakoj fazi mi izvodimo merdž lokalno, bez obzira na globalno stanje.
Pitanje. Ocijeniti složenost algoritma.
Odgovor.

1. Ako drveta držimo na heap-u (priority queue) tada je $T(C) = O(C \log C)$, jer imamo izgradnju heap-a, $2C - 2$ brisanja i $C - 2$ inserta.

2. Povezana lista reda za čekanje (Linked list priority queue) daje $T(C) = O(C^2)$. Pošto je u $ASCII$ kodu C malo, kvadratna ocjena je prihvatljiva.

Napomene.



Slika 3.9: Stanje nakon konačnog merdžovanja

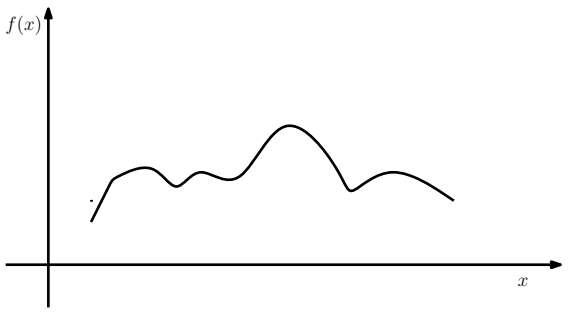
1. Kodirana informacija mora biti transformisana na početak datoteke, inače je nećemo moći dekodirati. To je slabo i skupo za male fajlove.

2. Algoritam ima dva prolaza. Prvi prolaz utvrdjuje frekvenciju podataka, a drugi kodira. Dvoprolaznost je skupa za velike fajlove.

3.4 Metoda "penjanja uz brdo" ("Hill Climbing"). Problem prelaska pustinje džipom

Algoritam "Hill Climbing" daje neki "lokalni optimum" za koji očekujemo da je jednak "globalnom optimumu". Ako apsolutno najbolje rješenje nije tačno, tada se "hill climbing" koristi za generisanje približnog rješenja (Slika 3.10).

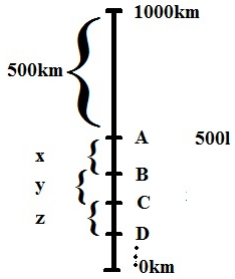
U slučaju da se traži tačno rješenje, tada "hill climbing" algoritam treba poboljšati do tačnog algoritma. Ovdje metodu ilustrujemo primjerom prelaska pustinje džipom.



Slika 3.10: Metoda "Hill Climbing" (penjanje uz brdo)

Primjer 3.3. *Pustinja je duga* $1000km$*. Džip troši* $1l$ *benzina na* $1km$*. Neka na početku pustinje postoji neograničena količina goriva i neka je dozvoljeno praviti skladišta goriva na proizvoljno izabranom mjestu u pustinji. Naći minimalnu količinu goriva potrebnu da džip predje pustinju, ako džip može nositi maksimalno* $500l$ *goriva.*

Rješenje.



Slika 3.11: Skica rješenja problema skladišta

1. Neka džip iz poslednjeg skladišta A uzme dovoljnu količinu goriva za dolazak do kraja pustinje. Koristeći princip optimalnosti zaključujemo da skladište A mora sadrzati $500l$ benzina.

2. Skladište A punimo koristeći skladište B, udaljenosti xkm od skladišta A. Da bi smo u A mogli uskladištiti $500l$ benzina moramo imati u B $(3x + 500)$l benzina. Zbog optimalnosti džip u svakoj turi nosi maksimalnu količinu od $500l$ benzina. To nam daje jednačinu $3x + 500 = 1000$ za tačku B, pa slijedi da je tako maksimizirano $x = \frac{500}{3}$.

3. Slično za tačku C dobijamo $5y + 1000 = 1500$, odakle je $y = \frac{500}{5}$. Dalje je $z = \frac{500}{7}$ itd.

Konačno imamo:

$$(500 + x + y + z + ...) = 1000$$
$$(500 + \frac{500}{3} + \frac{500}{5} + \frac{500}{7} + ...) = 1000$$
$$500(1 + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + ...) = 1000 \text{ tj.}$$
$$(1 + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + ...) = 2$$

Optimalnost algoritma.
Broj skladišta i količinu benzina u njima odredjujemo iz broja članova harmonij-

skog reda čiji zbir po prvi put premaši 2, znači

$$(1+\frac{1}{3}+\frac{1}{5}+\frac{1}{7}+\frac{1}{9}+\frac{1}{11}+\frac{1}{13}).$$

Navedeni dokaz optimalnosti je heuristički. Formalan dokaz je težak. Naročito je teško formalno dokazati da je nadjena količina od $3837,5l$ minimalna. □

Glava 4

Aproksimativni algoritmi

Veliki broj, ako ne i većina problema optimizacije koje u praksi treba riješiti, su NP-teški. Iz teorije složenosti slijedi da je nemoguće naći efektivan algoritam za takve probleme, osim ako je $P = NP$, što teško može biti tačno. Ovo ne odstranjuje potrebu za rješavanje tih problema. Primijetimo da NP-težina znači samo, ako je $P \neq NP$, onda ne postoji algoritmi koji će naći tačno optimalna rješenja za sve instance problema u vremenu koje je polinomijalno u odnosu na dužinu ulaza. Ako oslabimo posljednji zahtjev, onda će još uvijek biti moguće riješiti problem na dovoljno prihvatljiv način.

Postoje tri mogućnosti za slabljenje pomenutih zahtjeva poslije kojih problem može biti na zadovoljavajući način riješen u praksi:

- Heuristike sa super-polinomijalnim vremenom složenosti
- Stohastička analiza heuristika
- Algoritmi aproksimacije

U ovoj glavi ćemo oslabiti zahtjev za nalaženje optimalnog rješenja i zadovoljićemo se približnim rješenjem. U praksi je obično teško naći razlike izmedju optimalnog rješenja i rješenja koje je blisko optimalnom. To nas dovodi do pojma "aproksimativnog" rješenja problema optimizacije. Neki problemi su jednostavni za aproksimaciju, kao problem pakovanja ranca, pravljenje rasporeda i pakovanje u binove. Neki su problemi tako teški da je za njih teško naći i vrlo slabu aproksimaciju. Takvi su problem bojenja grafova, problem trgovačkog putnika i problem klike. Postoje i problemi koji izgleda imaju srednju složenosti , kao što su pokrivanje grafa vrhovima, problem trgovačkog putnika u Euklidovom prostoru ili Štajnerova drveta. Za neke probleme je lako objasniti zbog čega im se lako nalaze aproksimativna rješenja, ali je većina NP-kompletnih problema još uvijek misterija.

4.1 Pravljenje rasporeda: minimizacija konačnog vremena čekanja. Problem pakovanja ranca

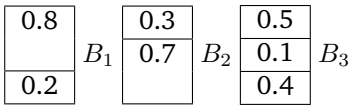
Neka su dati poslovi $p_1, p_2, ..., p_n$ sa vremenima izvršavanja $t_1, t_2, ..., t_n$ respektivno. Treba naći raspored izvršavanja poslova koji će minimizirati neku zadatu funkciju. U sekciji 3.2 opisali smo i riješili problem minimizacije srednjeg vremena čekanja na jednom ili više procesora.

U problemu minimizacije *konačnog vremena izvršavanja* traži se raspored u kome će vrijeme izvršavanja i posljednjeg procesa biti minimalno(optimalno). Mada je po formulaciji posljednji problem sličan prethodnom, pokazuje se da je on zapravo mnogo teži. Ovaj problem je NP - kompletan jer je ekvivalentan problemu pakovanja ranca. U 3.2 dali smo ilustraciju problema optimizacije konačnog vremena izvršavanja za jedan specijalni slučaj.

Napominjemo da mi razmatramo nonpreemptivne rasporede, to jest, ako je neki posao počet, tada se on izvršava do kraja i to bez prekida. Umjesto problema pakovanja ranca mi razmatramo ekvivalentan problem pakovanja u bin-ove (kutije, korpe). Daćemo aproksimativno rješenje problema pakovanja u kutije (binove). Aproksimativni algoritmi su brži u odnosu na optimalne, ali ne daju uvijek optimalno rješenje. Dokazujemo da naša rješenja nisu daleko od optimalnih.

Dato je n komada kapaciteta $s_1, s_2, ..., s_n$. Svi komadi zadovoljavaju $0 < s_i < 1$. *Problem pakovanja u binove* je problem pakovanja svih komada u što manji broj kutija (binova) kapaciteta 1.

Primjer 4.1. *Neka su zadati komadi* $s_1 = 0.2, s_2 = 0.5, s_3 = 0.4, s_4 = 0.7, s_5 = 0.1, s_6 = 0.3, s_7 = 0.8$. *Jedan od načina pakovanja u binove* B_1, B_2 *i* B_3 *pokazan je na sljedećoj slici.*



Razlikujemo on-line i off-line algoritme pakovanja u binove. On-line problem traži da svaki komad mora biti ubačen u kutiju prije nego što sledeći komad bude pokazan i vidjen. U Off-line problemima ne moramo raditi ništa prije nego što svi komadi budu učitani i vidjeni.

4.2 On-line algoritmi: Pakovanja u binove

Definisaćemo NP-težak problem optimizacije i objasniti dva pojma aproksimacije.

Definicija 4.1. Problem optimizacije Π *opisujemo sa sljedeće tri komponente:*

- **Predmeti D** *su skup ulaznih predmeta.*
- **Rješenje S** *je skup svih dostižnih (feasible) rješenja* $I \in D$.

- **Vrijednost** f *je funkcija koja dodjeljuje vrijednost svakom rješenju, to jest* $f : S(I) \to \mathbf{R}$.

- **Maksimizacija problema** Π. *Neka je dat* $I \in D$. *Naći rješenje* $\sigma^I_{opt} \in S(I)$ *tako da je*
$$(\forall \sigma \in D(I))(f(\sigma^I_{opt}) \geq f(\sigma))$$

Slično se definiše **minimizacija problema**.

Sljedeći primjer ilustruje definiciju.
Pakovanje u binove (BP). Neka je dat skup predmeta, veličine izmedju 0 i 1. Treba upakovati predmete u binove jedinične dimenzije tako da broj upotrijebljenih binova bude minimalan. Drugim riječima

- **Predmeti.** $I = \{s_1, s_2, \ldots, s_n\}$, takvi da je $(\forall i)(s_i \in [0, 1])$.

- **Rješenje.** Skup podskupova $\sigma = \{B_1, B_2, \ldots, B_k\}$, koji je disjunktna particija I, tako da $(\forall i)(B_i \subset I)$ i $\sum_{j \in B} s_j \leq 1$.

- **Vrijednost.** Vrijednost rješenja je broj iskorištenih binova ili $f(\sigma) = |\sigma| = k$.

Naša namjera je da pokažemo da je neki problem NP-težak, a da onda nadjemo aproksimativno rješenje toga problema. Dajemo sljedeće potrebne definicije pojmova *svodljivosti* i *aproksimativnog algoritma*.

Definicija 4.2. *Ako je neki* NP*-težak problem odlučivosti* Π_1 *polinomijalno svodljiv na nalaženje rješenja problema optimizacije* Π_2, *onda je* Π_2 NP*-težak*.

Definicija 4.3. Aproksimacioni algoritam *za optimizacioni problem* Π, *je algoritam polinomijalne vremenske složenosti takav da za ulaz predmeta* I *za* Π, *daje neki izlaz* $\sigma \in S(I)$, *gdje je* $S(I)$ *skup rješenja problema. Sa* $A(I)$ *označavamo vrijednost* $f(\sigma)$ *rješenja dobivenog pomoću* A.

Definicija 4.4. Apsolutno aproksimirajući algoritam *je algoritam polinomijalne vremenske složenosti za problem* Π, *takav da za neku konstantu* $k > 0$ *vrijedi*

$$(\forall I \in D)|A(I) - OPT(I)| \leq k.$$

Poznato je da je bojenje planarnih grafova sa 3 boje NP-težak problem. Takodje je poznato da se svaki planarni graf može obojiti sa 5 boja. Razmotrimo sljedeći algoritam A koji boji planarne grafove. A prvo ispita je li graf 2-obojiv, to jest, je li graf bipartitan i računa bojenje sa 2 boje, ako je to moguće. Inače, A nalazi trivijalno bojenje u 5 boja. To znači da A nikada ne koristi više od 2 dodatne boje. Sada možemo formulisati teoremu:

Teorema 4.1. *Neka je dat proizvoljan planarni graf* G. *Performanse aproksimacionog algoritma* A *su takve da je* $|A(G) - OPT(G)| \leq 2$.

Apsolutne garancije performansi su vrlo poželjne, ali izgleda malo vjerovatno da se mogu dobiti takve garancije za bilo koju interesantnu klasu teških problema optimizacije. Izlaz je u oslabljenju zahtjeva u definiciji " dobrog" algoritma aproksimacije i uvodjenju relativno garantovanih performansi. To nas dovodi do definicije količnika performansi $R_A(I)$ za aproksimacioni algoritam A. Kao motivacioni primjer navodimo pravljenje rasporeda na više procesora i koristimo ga kao osnovu za definisanje relativnih garancija performansi. Zanimljivo je da čitava oblast aproksimacionih algoritama ima korjen u radu [12], u kome je dat Grahamov algoritam "list scheduling".

Uzimamo najprostiju verziju problema pravljenja rasporeda na više procesora. Na ulazu je dato n poslova $P_1, P_2, \ldots, P_n$. Svaki posao ima odgovarajuće vrijeme izvodjenja $t_1, t_2, \ldots, t_n$, pri čemu je svaki t_i, $(1 \leq i \leq n)$, racionalan broj. Poslovi se rasporedjuju na m identičnih mašina ili procesora i cilj je da se minimizira konačno vrijeme izvodjenja. Poslove pridjeljujemo mašinama na online način. Aktuelni posao dodjeljuje se mašini koja je u tom momentu najmanje zauzeta. Skup svih dostižnih rješenja se sastoji od svih particija n poslova u m podskupova, a vrijednost nekog rješenja je maksimum vremena izvodjenja nad svim vremenima izvodjenja tih podskupova. Poznato je da je problem NP-$kompletan$ čak i u slučaju $m = 2$. Označimo opisani algoritam sa A. Tada važi sljedeća teorema.

Teorema 4.2. *Neka A označava algoritam rasporeda. Tada za sve instance I ulaza, važi*

$$\frac{A(I)}{OPT(I)} \leq 2 - \frac{1}{m},$$

Osim toga, ovo ograničenje je precizno, u smislu da postoji instanca I^*, za koju je

$$\exists I^* \text{ tako da je } \frac{A(I^*)}{OPT(I^*)} \leq 2 - \frac{1}{m}.$$

Dokaz. Prvo ćemo dokazati gornju granicu količnika. Bez gubitka opštosti možemo pretpostaviti da nakon dodjele svih poslova mašina M_1 ima najveće opterećenje. Opterećenje L označava vrijeme ukupnog izvodjenja svih poslova dodijeljenih mašini M_1. Neka je P_j posljednnji posao koji je dodijeljen mašini M_1.

Tvrdimo da ukupno opterećenje svake mašine nije manje od $L - t_j$. Tvrdnja važi zbog toga što u trenutku kada je posao P_j bio dodijeljen M_1, mašina M_1 je bila najmanje opterećena i imala je opterećenje $L - t_j$. Odatle slijedi da je

$$\sum_{i=1}^{n} t_i \geq m(L - t_j) + t_j.$$

Ali, pošto je $OPT(I) \geq \frac{\sum_{i=1}^{n} t_i}{m}$, jer neki procesor mora imati toliko opterećenje na kraju izvodjenja i pošto je i $A(I) = L$, slijedi

$$OPT(I) \geq (L - t_j) + \frac{t_j}{m} \geq A(I) - (1 - \frac{1}{m})t_j.$$

Pošto neki proces mora uraditi posao P_j, slijedi da je $OPT(I) \geq t_j$, a to daje traženi rezultat.

Da je količnik dostignut dokazujemo konstrukcijom instance I^*. Uzimamo $n = m(m-1)+1$. Neka svaki od $(n-1)$ poslova ima vrijeme izvodjenja 1, a posljednji posao ima vrijeme izvodjenja $t_n = m$. Odatle slijedi da je $OPT(I^*) = m$ dok je $A(I^*) = 2m-1$, što daje traženo donje ograničenje količnika. □

Definicija 4.5. *Neka je A aproksimacioni algoritam za problem optimizacije Π. Količnik performansi $R_A(I)$, algoritma A za istancu I kao ulaz, je definisan sa*

$$R_A(I) = \frac{A(I)}{OPT(I)}$$

u slučaju problema minimizacije Π.

Sa druge strane, ako je Π problem maksimizacije, onda je odnos performansi

$$R_A(I) = \frac{OPT(I)}{A(I)}.$$

Napomena. U drugom dijelu definicije 4.5 uzimamo recipročnu vrijednost da bi količnik bio manji od jedinice. Što je količnik bliže jedinici to je aproksimativni algoritam bolji.

Rezultati dokazani u teoremi 4.2 motivišu nas da uvedemo sljedeće definicije:

Definicija 4.6. *Apsolutni količnik performansi R_A aproksimativnog algoritma A, za problem optimizacije Π je*

$$R_A = inf\{r | R_A(I) < r, \text{ za svako } I \in D\}.$$

Primjer 4.2. LPT algoritam *je algoritam rasporedjivanja koji prvo uredi poslove u opadajućem redoslijedu prema njihovim vremenima izvodjenja, a zatim nastavlja dalje isto kao gore opisani algoritam Grahama.*

Primjer 4.3. *Za algoritam Grahama imamo $R_A = 2 - \frac{1}{m}$.*

Zadatak 4.1. *Dokazati da za LPT algoritam vrijedi $R_{LPT} = \frac{4}{3} - \frac{1}{3m}$.*

Definicija 4.7. *Asimptotski količnik performansi R_A^∞, nekog aproksimativnog algoritma A, za problem optimizacije Π, je*

$$R_A^\infty = inf\{r | \exists N_0 \text{ takav da je } R_A(I) \leq r \text{ za sve } I \in D_\Pi \text{ sa } OPT(I) \geq N_0.\}$$

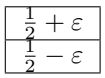
Definicija 4.8. Najbolji ostvariv količnik performansi $R_{MIN}(\Pi)$ *za optimizacioni problem Π, definiše se kao $R_{MIN}(\Pi) =$*

$$= inf\{r \geq 1 | \exists \text{ algoritam } A \text{ polinomijalne vremenske složenosti za } \Pi \text{ sa } R_A^\infty < r\}$$

Pokazujemo da On-line algoritmi pakovanja u binove ne mogu uvijek dati odgovor koji je optimalan. Čak ako mu dozvolimo neograničeno računanje, on-line algoritam mora smjestiti komad prije nego što razmotri sljedeći komad. Situaciju ilustrujemo sljedećim primjerom :

Primjer 4.4. *Neka su* $I_1 = \{i_1 | |i_1| = \frac{1}{2} - \varepsilon\}$, $|I_1| = m$ *i* $I_2 = \{i_2 | |i_2| = \frac{1}{2} + \varepsilon\}$, $|I_2| = m$, $0 < \varepsilon < 0.01$. *Neka je* $I = I_1 \cup I_2$, *gdje je* $\cup$ *operacija konkatenacije (dopisivanja)nizova.*

1. Pretpostavimo da postoji optimalan algoritam A koji pakuje niz I optimalno u m kutija, tako da jedan bin izgleda kao na slici.

2. Neka je $J = I_1$. Niz J može biti upakovan u $[\frac{m}{2}]$ kutija, ali će se optimalnim algoritmom A pakovati isto kao i prije niz I, dakle u m kutija.

   Iz (1) i (2) slijedi da ne postoji optimalan on-line algoritam za pakovanje u binove.

Dajemo sljedeću kvantitativnu ocjenu.

Teorema 4.3. *Postoji ulaz koji forsira on-line algoritam pakovanja u binove da koristi najmanje* $\frac{4}{3}$ *od optimalnog broja binova.*

Dokaz.
Neka je $m \in 2\mathbb{N}$, a niz I kao u primjeru 4.4. Pretpostavimo suprotno.

1. Nakon m komada A koristi b binova, a optimalno je $\frac{m}{2}$.
   Na osnovu pretpostavke slijedi $\frac{2b}{m} < \frac{4}{3}$ i odatle $6b < 4m$.

2. Nakon $2m$ komada svi su komadi upakovani. Svi binovi nakon $b-$tog sadrže najviše jedan komad, jer su svi $\frac{1}{2} - \varepsilon$ komadi upakovani u b binova.
   $\Rightarrow$ Trebamo $2m - b$ binova više $\Rightarrow \frac{2m-b}{m} < \frac{4}{3} \Rightarrow 6b > 4m$.

Zakljuci pod (1) i (2) su u kontradikciji i to dokazuje tvrdnju. □

Postoje tri algoritma, jednostavna za implementaciju, koji garantuju ne više od $2 *$ *(optimalan broj binova)* : *"NEXT FIT", "FIRST FIT", "BEST FIT"*. Razmotrićemo ih u sljedećim primjerima.

4.2.1 "Next fit" pakovanje

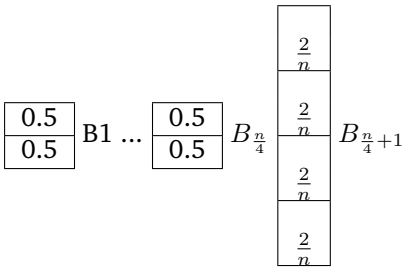
Algoritam Next fit (sljedeći odgovarajući): Komad se stavlja u istu kutiju kao i prethodni, a ako to nije moguće stavlja se u novoformiranu kutiju.

Teorema 4.4. *Neka je* m *optimalan broj kutija potrebnih za pakovanje niza* I *komada. "Next fit" algoritam ne koristi više od* $2m$ *binova. Postoji i niz* I^* *na kome* NF *algoritam koristi* $2m - 2$ *binova (kutija).*

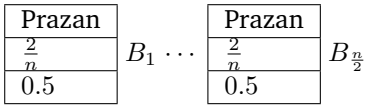
Dokaz. B_j i B_{j+1} sadrže komade sa zbirom većim od 1 - inace bi svi komadi iz njih stali u jedan bin. Primijenjeno na sve parove susjednih binova daje dokaz, jer najviše pola prostora može biti izgubljeno (nepopunjeno).

Za dokaz drugog dijela teoreme prikazujemo optimalno pakovanje niza I^*, a zatim "next fit" pakovanje koje zauzima $2m-2$ binova.

Primjer 4.5. *Dajemo primjer optimalnog pakovanja niza I^* (gornja slika) i "next fit" pakovanja niza I^* (donja slika), pri čemu je $\boldsymbol{I}^* = (0.5, \frac{2}{n}, 0.5, \frac{2}{n}, ...)$*



Optimalno pakovanje niza I^*

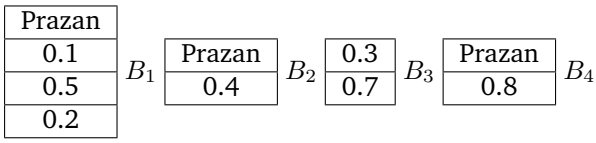


"Next fit" pakovanje niza I^* □

4.2.2 "Best fit" pakovanje

Algoritam "Best fit" (najbolje smještanje): Umjesto stavljanja u prvi bin (kutiju, korpu, boks), traži se bin u koji komad najbolje pasuje ("tijesno pasuje").

Primjer 4.6. *"Best fit" za niz* $\lceil 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8 \rceil$



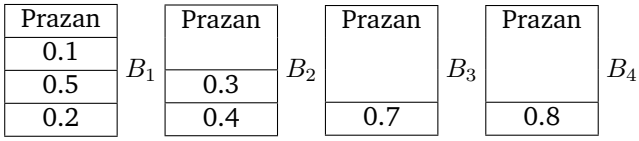
Bez dokaza navodimo da "best fit" u najgorem slučaju troši 1.7 od optimalnog broja binova.
Jednostavan je za kodiranje ako trebamo $0(n \log n)$ vrijeme izvodjenja.

4.2.3 "First fit" pakovanje

Algoritam "First fit" (Prvi odgovarajući)
"First fit" strategija skenira binove (kutije,korpe,boksove) i smješta novi komad u

prvu pregradu (prvi bin) sa dovoljno mjesta da se prihvati. Nova kutija se kreira samo ako prethodne kutije ne mogu da prihvate element.

Primjer 4.7. *"First fit" za niz ⌈ 0.2,0.5,0.4,0.7,0.1,0.3,0.8 ⌉*



Teorema 4.5. *Neka je m optimalan broj binova za pakovanje niza I.*

1. Tada "First fit" koristi manje ili jednako od $\left\lceil \frac{17}{10}m \right\rceil$ binova.

2. Postoji niz koji zahtijeva $\frac{17}{10}(m-1)$ binova.

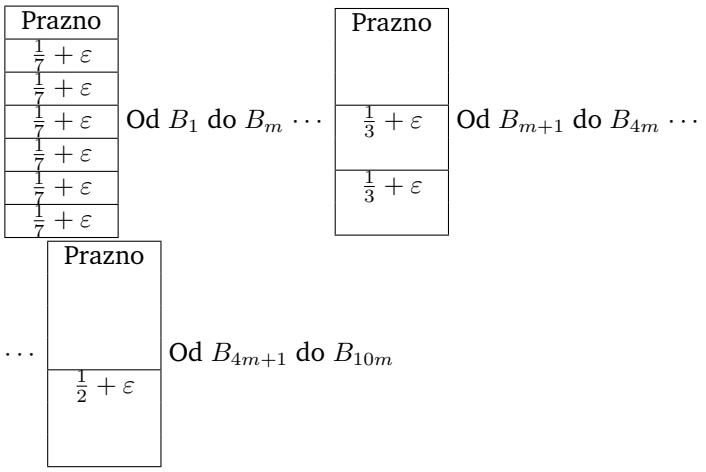
Dokaz. Ovdje izostavljamo dokaz **(1)**
(2) Za ilustraciju dajemo sljedeći primjer: Neka u nizu I imamo

$6m$ komada dužine $\frac{1}{7}+\varepsilon$

$6m$ komada dužine $\frac{1}{3}+\varepsilon$

$6m$ komada dužine $\frac{1}{2}+\varepsilon$.

Lako se vidi da *optimalno* imamo $6m$ binova, svaki za sve tri vrste elemenata. "First fit" koristi $10m$ binova umjesto $6m$.

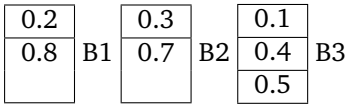


Napomena. Praktična mjerenja pokazuju da u slučaju vrijednosti uniformno rasporedjenih u intervalu $[0, 1]$, za "first fit" trebamo oko 2 posto više binova nego optimalno.

4.3 Off-Line algoritmi: "First fit decreasing"

U off-line obradi korisno je elemente sortirati prije rasporedjivanja u binove. Za "First fit" algoritam sortiramo u opadajućem (decreasing) poretku, a za "best fit" zgodnije je sortirati elemente u rastućem (increasing) poretku.

Primjer 4.8. *"First fit" za niz* $\langle 0.8, 0.5, 0.4, 0.3, 0.2, 0.1\rangle$



Želimo dokazati da "first-fit" decreasing algoritam koristi najviše $\frac{4m+1}{3}$ binova, ukoliko optimalno pakovanje koristi m binova.

Lema 4.1. *Neka n elemenata niza sortiranog u opadajućem poretku imaju dužine $s_1, ..., s_n$ i pretpostavimo da se optimalno mogu smjestiti u m binova. Tada svi komadi koji se smještaju u dodatne binove po principu "first fit" decreasing imaju dužinu manju ili jednaku $\frac{1}{3}$.*

Dokaz. Dokažimo da za s_i element koji je prvi smješten u $m+1-$om binu važi $|s_i| \leqq \frac{1}{3}$.

Pretpostavimo da je $\mid s_i \mid > \frac{1}{3}$. Tada je svaki od $s_1, s_2, ..., s_{i-1} > \frac{1}{3}$ jer su sortirani u opadajućem poretku. Zbog toga svaki od binova $B_1, ..., B_m$ imaju najviše dva elementa.
Neka postoje B_x i B_y takvi da je $1 \leq x < y \leq m$ i neka su u B_x dva komada x_1 i x_2, a u B_y jedan komad y_1. Tada $(x_1 \geq y_1 \wedge x_2 \geq s_i) \Rightarrow x_1 + x_2 \geq y_1 + s_i$.
Iz zaključka implikacije slijedi da s_i može biti smješten u B_y, što je suprotno pretpostavci.

Slijedi da moramo prvo imati niz binova sa po jednim komadom, a zatim ide niz binova koji sadrže po dva komada. Neka je prvih j, a drugih $m - j$. Neka su $s_1, ..., s_j$ po jedan u binu, a $s_{j+1}, s_{j+2}, ..., s_{i-1}$ u $m - j$ binova, po dva komada zajedno $\Rightarrow 2(m - j)$ je ukupan broj komada u tih m-j binova. Sada slijedi da se ne mogu svi elementi staviti u m binova, što je kontradikcija. □

Lema 4.2. *Broj objekata u dodatnim binovima najviše može biti $m - 1$.*

Dokaz.
Pretpostavimo, suprotno tvrdnji, da u dodatnim binovima imamo $\geq m$ komada (elemenata niza). Pošto se optimalno svi elementi niza $s_1, ..., s_n$ mogu smjestiti u

m binova, to je $\sum_{i=1}^{n} s_i \leq m$. Sa druge strane, ako označimo

$s_i = \begin{cases} w_j, & 1 \leq j \leq m \\ x_i, & \text{ako je } s_i \text{ u dodatnom binu(ima ih} \geq m \text{ po pretpostavci)} \end{cases}$

imamo da je $\sum_{i=1}^{n} s_i \geq \sum_{j=1}^{m} w_j + \sum_{j=1}^{m} x_j \geq \sum_{j=1}^{m}(w_j + x_j) \Rightarrow w_j + x_j > 1$, inače bi x_j moglo biti smješteno u $B_j \Rightarrow \sum_{j=1}^{n} s_i > \sum_{j=1}^{m} 1 > m$ Kontradikcija! Time je lema dokazana. □

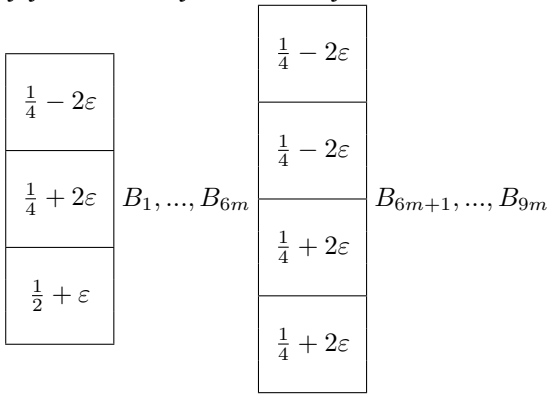
Teorema 4.6. *Neka je m optimalan broj binova za pakovanje niza I elemenata $s_1, ..., s_n$. Tada "first-fit" decreasing ne treba više od $\frac{4m+1}{3}$ binova.*

Dokaz. m-1 elemenata veličine koja je $\leq \frac{1}{3}$
$\Rightarrow$ treba $\leq \lceil \frac{m-1}{3} \rceil$ ekstra binova. $\Rightarrow$ Ukupan broj binova je najviše $\lceil \frac{4m-1}{3} \rceil \leq \frac{4m+1}{3} \rceil$. □

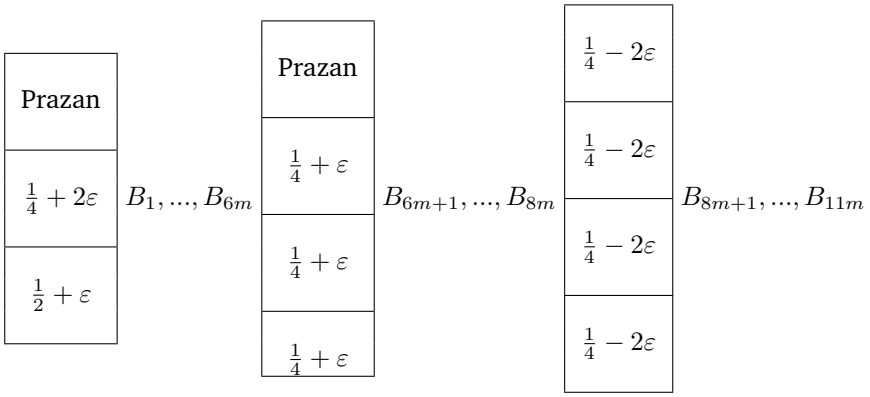
Teorema 4.7. *Neka je m optimalan broj binova potrebnih za pakovanje niza $I = \langle s_1, s_2, ..., s_n \rangle$*

1. Tada "first fit" decreasing ne koristi više od $\frac{11}{9}m$ +4 binova
2. Postoji niz koji za FFD trazi $\frac{11}{9}$m binova.

Dokaz. (2) Neka je niz I dat na sljedeći način:
$\frac{1}{2} + \varepsilon - 6m$ elemenata
$\frac{1}{4} + 2\varepsilon - 6m$ elemenata
$\frac{1}{4} + \varepsilon - 6m$ elemenata
$\frac{1}{4} - 2\varepsilon - 12m$ elemenata
Na gornjoj slici imamo optimalno pakovanje niza I, što sa FFD pakovanjem na donjoj slici dokazuje našu tvrdnju.



Optimalno pakovanje niza I



"First fit decreasing" pakovanje niza I

Zadatak 4.2. *Naći R_A količnike, gdje je algoritam*

$$A \in \{"next\ fit", "best\ fit", "first\ fit"\},$$

algoritam pakovanja u binove (a) u "on-line" verziji, (b) u "off-line"verziji.

Glava 5

Metoda "Backtracking" (pretraživanje sa vraćanjem)

Kada dobijemo posao da konstruišemo algoritam za nalaženje rješenja nekog specifičnog problema, obično ne počinjemo sa primjenom nekih utvrdjenih pravila, već pokušavamo metodu probe i grešaka kombinovanu sa metodom razbijanja problema na module i podmodule. Interesantno je pokušati generalisati takav pristup da bi se dobila "opšta metoda" za rješavanje bilo kojeg problema. Parcijalni poslovi povezani sa razbijanjem na module često su prirodno predstavljeni rekurzijom i sastoje se od konačnog broja podposlova. Čitav proces može biti vidjen kao proces probe ili pretraživanja koji postepeno pravi i podsijeca drvo podposlova. U velikom broju problema ovo drvo raste veoma brzo, najčešće eksponencijalno, zavisno od zadatih parametara, što komplikuje pretraživanje. Ponekad, korištenjem heuristika, drvo pretraživanja može biti podsječeno i redukovano do prihvatljivih granica, ali najčešće za ulaz velike dimenzije nije praktično.

U ovoj glavi diskutujemo opšte principe razbijanja poslova rješavanja problema u manje poslove, primjenu metode probe i grešaka i mogućnost primjene rekurzije.

5.1 Pretraživanje lavirinta

Jedan od načina pretraživanja lavirinta bez rizika da se zaluta primjenjivao se još u antičkim vremenima i poznat je iz mita o Tezeju i Minotauru. Sastojao se u tome da se pri kretanju kroz lavirint razmotava klupko konca. Konac garantuje da uvijek možemo izaći iz lavirinta, ali mi smo zainteresovani da prodjemo svaki dio lavirinta i da pri tome ne ponavljamo nijedan put ako to nije neophodno. Za postizanje toga cilja mi nećemo koristiti konac, nego ćemo razviti druge metode koje se mogu implementirati na računaru. U novije vrijeme je razvijena tehnika pretraživanja u dubinu (DFS) (vidi [9]) kao metoda za skeniranje konačnog, neorijentisanog grafa. Inače je ideja algoritma pretraživanja u dubinu poznata od

devetnaestog vijeka kao metoda za pretraživanje lavirinta. Razvili su je Tremo (Trémaux) i kasnije Tari (G. Tarry). Algoritam pretraživanje u dubinu je specijalan slučaj algoritma Tari. Dodatna struktura pretraživanja u dubinu čini tu tehniku tako korisnom.

Pretpostavimo da je dat konačan, povezan graf $G = (V, E)$ koji ćemo zvati *lavirint*. Startujući iz jednog vrha želimo proći po granama, od vrha do vrha, posjetiti sve vrhove i onda stati. Tražimo algoritam da garantuje da će cio graf biti skeniran bez nepotrebnog lutanja po lavirintu i da će procedura omogućiti da se vidi kada je posao završen. Pri tome, na početku šetnje u lavirintu ne zna se ništa o strukturi lavirinta i nije moguće planiranje obilaska unaprijed. Odluka o pravcu kretanja mora biti donesena "on line". Koristićemo markere koje ćemo postavljati u lavirintu da nam pomognu da prepoznamo kuda se vratiti na mjesto koje smo ranije posjetili i da nam pomognu u donošenju odluke kuda ići dalje. Markiraćemo *prolaze*, naime spojeve grana sa vrhovima. Ako je graf predstavljen listom incidencije, tada se svaka grana pojavljuje dva puta u listi incidencije njena dva kraja kao dva njena *prolaza*. Dovoljno je koristiti dva tipa markera, F za prvi *prolaz* korišten da se posjeti vrh i E za bilo koji drugi *prolaz* korišten da se napusti vrh. Nijedan marker se ne mijenja i ne briše. Kao što se lako vidi, nema korištenja memorije, osim za markere na *prolazima* i za izvršenje algoritma. Drugim riječima, potezi su kontrolisani konačnim automatom.
Dajemo pseudokod algoritma.

$function\ Tremaux(\mathbf{G}, \mathbf{s})\{$
$\mathbf{v} = \mathbf{s}$;
while$((\exists\ nemarkiran$ **prolaz** $u\ \mathbf{v})\ ili\ (u\ \mathbf{v}\ \exists$ **prolaz** $markiran\ F))$
    if $(\exists nemarkiran$ **prolaz** $na\ granu\ v \xrightarrow{e} u)\{$
        $markiraj$ **prolaz** od **e** na **v** sa **E**
        if (**u** $nema\ markirane$ **prolaze**)$\{$
            $markiraj$ **prolaz** od **e** na **u** sa **F**;
            $\mathbf{v} = \mathbf{u}\}$;
        else $markiraj$ **prolaz** od **e** na **u** sa **E**;
    else $(postoji$ **prolaz** u **v** $markiran\ sa$ **F**$\{$
        $koristi$ **prolaz** $markiran\ sa$ **F** $za\ prelaz\ na\ susjedni\ vrh$ **u**
        $\mathbf{v} = \mathbf{u}\}$
    $\}$
$\}$

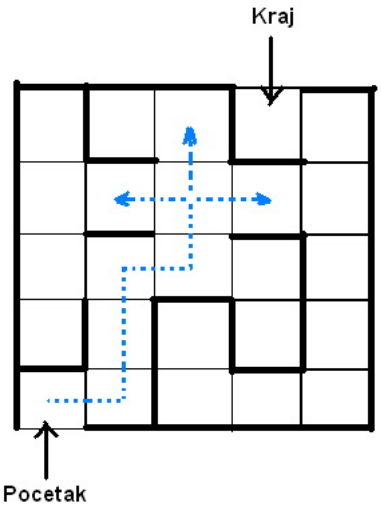
Zadatak 5.1. *1. Nacrtati neki konačan povezan graf (G,V) i provjeriti na njemu rad algoritma Tremaux(G,s).*

2. Dokazati da će algoritam Tremo završiti rad u vrhu iz kojeg je startovao i pri tome će svaku granu grafa G obići po jednom u oba smjera.

3. Implementirati algoritam Tremo u nekom programskom jeziku visokog nivoa.

Na drvovidnoj strukturi pretraživanja u dubinu zasniva se tehnika pretraživanja sa vraćanjem.

Ideju pretraživanja sa vraćanjem (backtracking) ćemo lako objasniti i na klasičnom primjeru izlaska iz lavirinta (slika 5.1). Naš je cilj da iz nekog zadatog kvadrata dodjemo u drugi zadati kvadrat premještajući se postepeno po kvadratima. Problem je u tome što u nekim kvadratima postoje pregrade koje onemogućuju prolaz. Jedan od načina prolaska kroz lavirint je kretanje od početnog kvadrata



Slika 5.1: Primjer lavirinta

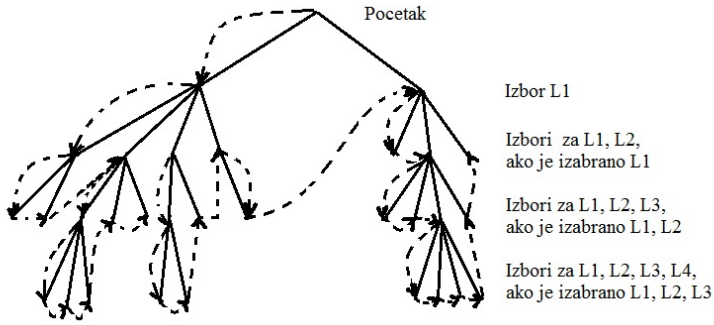
po sljedeća dva pravila.

1. U svakom kvadratu izabrati još neiskorišten put.
2. Ako iz kvadrata razmatranog u datom trenutku ne vode neiskorišteni putevi, tada se treba vratiti jedan korak nazad po zadnjem predjenom putu, po kojem smo ušli u kvadrat.

Suština pretraživanja sa vraćanjem sastoji se u tome da produžavamo put dok je moguće, a kada nije moguće, vratimo se po putu i pokušamo drugi izbor po najbližem putu, ako takav izbor postoji.

Backtracking možemo predstaviti drvetom pretraživanja (slika 5.2) tako da čvor drveta predstavlja dato stanje, a grane drveta predstavljaju korake iz datog stanja u sljedeća stanja. Tada drvo pretražujemo po putevima označenim isprekidanim linijama. Na nekom nivou biramo rješenja d_i uz pretpostavku da su $d_1 d_2 \ldots d_{i-1}$ izabrani na prethodnim nivoima. Predstavnike koji ne odgovaraju odbacujemo, to jest podrezujemo drvo.

U najopštijem slučaju pretpostavljamo da se rješenje sastoji od vektora $(a_1, a_2, \ldots, a_n)$, konačne, ali nedefinisane dužine, koji zadovoljava dodatna ograničenja. Pri tome je svako a_i iz nekog linearno uredjenog skupa A_i. Na taj način pri iscrpnom pretraživanju možemo razmotriti sve elemente skupa. $A_1 \times A_2 \times \ldots \times A_i,\ i = 0, 1, 2, \ldots$ kao moguća rješenja.



Slika 5.2: Drvo pretraživanja sa vraćanjem

Za polazno rješenje biramo prazan vektor () i na osnovu postojećih ograničenja nalazimo elemente iz A_1 koji su kandidati za a_1. Označićemo taj skup sa S_1. Za a_1 uzimamo najmanji elemenat iz S_1. Rezultat je parcijalno rješenje (a_1).

Uopšte, različita ograničenja koja opisuju rješenja, govore nam iz kojeg skupa $S_k \subseteq A_k$ uzimamo kandidate a_k za parcijalno rješenje $(a_1, a_2, \ldots, a_{k-1})$.

Ako parcijalno rješenje $(a_1, a_2, \ldots, a_{k-1})$ ne možemo raširiti dodavanjem elementa a_k, tada je $S_k = \emptyset$. U tom slučaju se vraćamo i biramo novi element a_{k-1}. Ako ni izabrani a_{k-1} ni izabrani S_{k-1} ne daju mogućnost raširenja, vraćamo se još jedan korak nazad i biramo element $a_{k-2} \in S_{k-2}$ i tako dalje.

5.2 Problem osam kraljica

Primjenjujemo opisanu metodu na rješavanje problema osam kraljica. U opštem problemu treba na šahovsku tablu dimenzije $n \times n$ postaviti što više kraljica tako da ni jedna ne napada drugu.

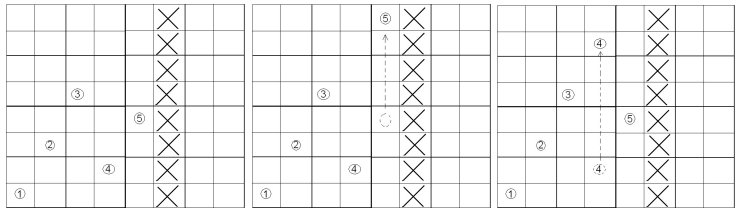
Pošto kraljica napada sva polja u koloni (stupcu),u redu i na dijagonali na kojoj se nalazi, jasno je da možemo postaviti najviše n kraljica koje jedna drugu ne napadaju. Problem se svodi na:

1. Postavljanje kraljica na ploči tako da jedna drugu ne napadaju.
2. Ako rješenje postoji, onda naći ukupan broj rješenja.

U svakoj koloni može se nalaziti tačno jedna kraljica, a to znači da rješenje možemo predstaviti vektorom $(a_1, a_2, \ldots, a_n)$ u kojem je a_i broj reda kraljice iz kolone **i**.

U svakom redu može postojati tačno jedna kraljica pa $i \neq j \Rightarrow a_i \neq a_j$. Kraljice ne smiju jedna drugu napadati po dijagonali i zbog toga moramo imati $|a_i - a_j| \neq |i - j|$ za $i \neq j$.

Da bi smo u ovom problemu dodali a_k u $(a_1, a_2, \ldots, a_{k-1})$ jednostavno poredimo a_k sa svakim a_i, $i < k$. Uvijek je $S_{n+1} = \emptyset$ za ploču $n \times n$. Ilustrujmo i slikama 5.3 backtrack na ploči 8×8.



Slika 5.3: Postavljanje kraljica na ploči 8×8 metodom backtrackinga

Jedan algoritam grube sile provjerava $\binom{n^2}{n}$ načina na koje se n kraljica može postaviti na ploču $n \times n$ (oko $4.4 \cdot 10^9$ za $n = 8$) i bira iz njih one koji zadovoljavaju uslove.

Druga ideja mogla je koristiti činjenicu da svaka kolona sadrži najviše jednu kraljicu, što bi dalo n^n mogućih položaja za istraživanje (za $n = 8$ oko $1.7 \cdot 10^7$). Ako primijetimo da nikoje dvije kraljice ne mogu biti u jednom redu, to znači da vektor $(a_1, \ldots, a_n)$ mora biti permutacija $(1, \ldots, n)$ što daje $n!$ mogucnosti (za $n = 8$ oko $4.0 \cdot 10^4$ mogućnosti).

Na kraju, pošto dvije kraljice ne mogu biti na jednoj dijagonali, broj mogućnosti se skraćuje još više (za $n = 8$ na oko 2056 mogućnosti što je računarski prihvatljivo).

Proces skraćivanja u problemu sa kraljicama možemo produžiti i dalje. Dva rješenja možemo smatrati ekvivalentnim ako se jedno od njih prevodi u drugo pomoću niza rotacija **i/ili** simetrija.

Ako utvrdimo postojanje kraljice u ćošku table, tada ta kraljica napada sve ostale ćoškove. Zbog toga nema rješenja u kojem je kraljica postavljena u više nego jednom ćošku. Ako je nadjeno rješenje u kojem je kraljica na polju $(1, 1)$, tada ono može biti prevedeno rotacijama i/ili simetrijama u ekvivalentno rješenje u kojem je polje $(1, 1)$ prazno. Uopšte, ako nadjemo sva neekvivalentna rješenja, tada možemo konstruisati i sva ostala rješenja.

Sljedeće usavršavanje odnosi se na spajanje grana drveta. Ako primijetimo da su dva poddrveta izomorfna, dovoljno je razmotriti samo jedno od njih.

Metoda preuredjenja drveta korisna je u slučaju u kojem neka rješenja imaju sličan izgled. Drvo možemo preurediti da prije svega pretražimo takva rješenja.

Metod dekompozicije je razbijanje problema na niz problema uz moguće korištenje metoda rekurzije i/ili dinamičkog programiranja.
Rješenje problema komponuje se od rješenja podproblema.

Za implementaciju našeg algoritma ključna je procedura u kojoj ispitujemo u kojem retku $k-$te kolone stoji kraljica. U grubim crtama ta procedura izgleda na sljedeći nacin :

```
boolean flag=true ;
    int i=1 ;
        while (i<k && flag)
            { if (a_i=a_k) || (|a_i-a_k|=|i-k|)
```

```
                then flag=false;
                i=i+1 ;
            }
```

Uz korištenje ove procedure program traženja rješenja u problemu n kraljica možemo skicirati na sljedeći način :

```
s_1=1;
k=1;
while k>0
    { while  s_k <= n
        {a_k = s_k;
        s_k = s_k + 1;
        while (s_k <=n && kraljica ne mo\v{z}e biti
        u redu s_k kolone k)
        {s_k = s_k + 1} ;
        if k = n then zapisati rjesenje
             (a_1,...,a_n) }
        k=k+1;
        s_k=1;
        while(s_k<=n && kraljica ne mo\v{z}e biti
        u redu s_k kolone k)
        {s_k=s_k+1}
        }
    k=k-1;
}
```

U ovom programu, umjesto čitavog skupa s_k čuva se samo minimalan element s_k. Provjera uslova $S_k \neq \emptyset$ odgovara uslovu $s_k \leq n$. Pošto s_{n+1} nije manje od $n+1$ to je S_{n+1} uvijek prazan skup.

Dajemo za ilustraciju neka detaljna rješenja problema n kraljica. Ideje ovih rješenja razlikuju se u nekoj mjeri od ideja za rješavanje koje smo prethodno izložili. Drugo rješenje dato je prema ideji iz knjige N. Wirth-a. Stare varijante programa su testirane na TURBO C++, v.6.0 i daju rješenja za $n < 70$, ali smo ih u ovom izdanju zamijenili varijantama prilagodjenim novijim kompajlerima.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
int a[30],count=0;

int mjesto(int pos)//u zavisnosti od pozicije
{
 int i;
 for(i=1;i<pos;i++)
 {
```

```
  if((a[i]==a[pos])||((abs(a[i]-a[pos])==abs(i-pos))))
   return 0;
 }
 return 1;
}
void ispis_rjesenja(int n)//broj kraljica
{
 int i,j;
 count++;
 printf("\n\nRjesenje #%d:\n",count);
 for(i=1;i<=n;i++)
 {
  for(j=1;j<=n;j++)
  {
   if(a[i]==j)
    printf("Q\t");//oznacimo kraljicu
   else
    printf("*\t");
  }
  printf("\n");
 }
}
void kraljica(int n)//trazenje rjesenja- procedura
{
 int k=1;
 a[k]=0;
 while(k!=0)
 {
  a[k]=a[k]+1;
  while((a[k]<=n)&&!mjesto(k))
   a[k]++;//odgovara pozicija
  if(a[k]<=n)
  {
   if(k==n)
   ispis_rjesenja(n);//nadjeno rjesenje printamo,
   //i nastavljamo pretragu
   else
   {
    k++;
    a[k]=0;
   }
  }
  else
   k--;
 }
}
int main()
{
 int i,n;
```

```
 kraljica(8);
 printf("\nRjesenja=%d",count);
 getch();
return 0;
}
```

Program za n kraljica koji umjesto rekurzije koristi stekove.

```
/*EightQueens Program koristi dva steka. Stekovi su
implementirani kori\v{s}tenjem nizova */

 # include <stdio.h>
 # include <stdlib.h>
 # include <time.h>

 typedef struct  { int x,y;
                   } position ;
 void SolveProblem(int n);
 int N=0;
int main()
 {printf("\nVELICINA ( N ) za NxN tablu :");
  scanf("%d",&N);
  printf("\nIn Koordinate kraljica na tabli N su dati sa  (Row,Col) .");
  printf("\none su numerisane od 1 - N :\n");
  SolveProblem(N);
  getchar();
  system("pause");
  return 0;
 }
 void SolveProblem(int n)
 {
 int counter1,counter2=-1,counter3=-1;
 static int counter=0,choice;
 int d[100][3]={0};
 int *stack2;
 position Position1,Position2,Position3;
 position *head1=(position *)malloc(n*n*sizeof(position));
  stack2=(int *)malloc(n*sizeof(int));
 for(counter1=n-1;counter1>=0;counter1--)
        { Position1.x=0;
          Position1.y=counter1;
          head1[++counter2]=Position1;
         };
  while(counter2>=0){ Position1=head1[counter2--];
                  while(counter3>=0 && Position1.x<=counter3){
                  Position2.x=counter3;
                  Position2.y=stack2[counter3--];
                  d[Position2.y][0]=0;
                  d[Position2.x+Position2.y][1]=0;
```

```
                d[Position2.x-Position2.y+n][2]=0;};
                stack2[++counter3]=Position1.y;
                d[Position1.y][0]=1;
                d[Position1.x+Position1.y][1]=1;
                d[Position1.x-Position1.y+n][2]=1;


              if(counter3==n-1)
                {
                 counter++;
                 printf("\nSOLUTION # %d:",counter);
                 for(counter1=0;counter1<=counter3;counter1++)
                  printf("(%d,%d) " ,counter1+1, stack2[counter1]+1);
                 if(counter%10==0)
                  {printf("\nEnter 1 to Continue , 0 to end.");
                                  scanf("%d",&choice);
                                  if(choice==0)
                                  exit(0);
                                  };
                       Position2.x=counter3;
                 Position2.y=stack2[counter3--];
                 d[Position2.y][0]=0;
                 d[Position2.x+Position2.y][1]=0;
                 d[Position2.x-Position2.y+n][2]=0;}
          else{for(counter1=n-1;counter1>=0;counter1--)
              if(d[counter1][0]==0 && d[Position1.x+1+counter1][1]==0
                              && d[n+Position1.x+1-counter1][2]==0)
               {Position3.x=Position1.x+1;
                Position3.y=counter1;
                head1[++counter2]=Position3;
                };
                  }
      }

  }
```

Dajemo još jedan program Dame1.cpp (ime definiše ulaznu tačku za konzolne aplikacije). U pitanju je pretežno C++ kod. Ako se umjesto cout i cin ubace printf i scanf, program može da prodje kao C kod. Ovo bi trebala da bude jedna od bržih implementacija koja rješava problem kraljica na šahovskoj tabli (iako izgleda malo duža)jer ima samo jednu petlju po pozivu funkcije. U programu se koriste statičke promjenjive (možda bi bilo bolje da su dinamičke), ali poenta algoritma nije u u tome.

```
// Izgled main funkcije (_tmain) je do Visual C++ kompajlera

#include <stdafx.h>
#include <iostream>
```

```
using namespace std;

int const MAX = 64;

// velicina table
int tabvel;

// kolone u kojima je dama
bool kolone[MAX];

// diagonala u kojima je dama; dijagonale su kodirane kao:
//RED - KOL + (VELICINA - 1) zbog toga sto nizovi moraju
//poceti od nule
bool diag[2 * MAX - 1];

// anti dijagonale u kojima je dama; kodirane su kao: RED + KOL
bool antidiag[2 * MAX - 1];

// matrica sluzi samo za lakse iscrtavanje rjesenja,
//nema uticaj na algoritam
bool matrica[MAX][MAX];

// ukupan broj rjesenja
int brres;

//
char ch;

void PrikaziResenje(void)
{
brres++;
cout << endl << "Rjesenje " << brres << "." << endl << endl;
for (int red = 0; red  < tabvel; red++)
{
for (int kol = 0; kol < tabvel; kol++)
{
if (matrica[red][kol])
cout << " * ";
else
cout << " - ";
}
cout << endl;
}

if ((brres % 10) == 0)
```

```
{
cout << endl << "Jos rjesenja (d)? ";
cin >> ch;
if (ch != 'd')
exit(0);
}
}

void DameRek(int red)
{
// ako smo uspjeli da postavimo N dama onda ispisujemo rjesenje
if (red == tabvel)
PrikaziResenje();
else
{
for (int kol = 0; kol < tabvel; kol++)
{
  //redovi se ne provjeravaju jer uvijek stavljamo damu u sljedeci red
if (!kolone[kol] && !diag[red - kol + (tabvel - 1)]
                            && !antidiag[red + kol])
{
// stavljanje nove dame na tablu
matrica[red][kol] = true;
kolone[kol] = true;
diag[red - kol + (tabvel - 1)] = true;
antidiag[red + kol] = true;

DameRek(red + 1);

// skidanje dame sa table
matrica[red][kol] = false;
kolone[kol] = false;
diag[red - kol + (tabvel - 1)] = false;
antidiag[red + kol] = false;
}
}
}
}


int _tmain(int argc, _TCHAR* argv[])
{
// pocetna inicijalizacija nizova
for (int i = 0; i < 2*MAX-1; i++)
{
```

```
tabvel = 0;
if (i < MAX)
{
kolone[i] = false;
for (int j = 0; j < MAX; j++)
matrica[i][j] = false;
}
diag[i] = false;
antidiag[i] = false;
}

cout << "Dimenzija table?: ";
cin >> tabvel;

// dame se stavljaju na tablu pocev od prvog reda
//(prvi red ima indeks 0)
if (tabvel > 0)
DameRek(0);

cout << endl << "Kraj";
cin.get();
cin.get();
return 0;
}
```

Zadatak 5.2. *Napisati program koji rješava problem n kraljica na šahovskoj ploči koristeći:*

1. *BFS tehniku pretraživanja.*
2. *eliminacione heuristike,*
3. *programski jezik Prolog.*

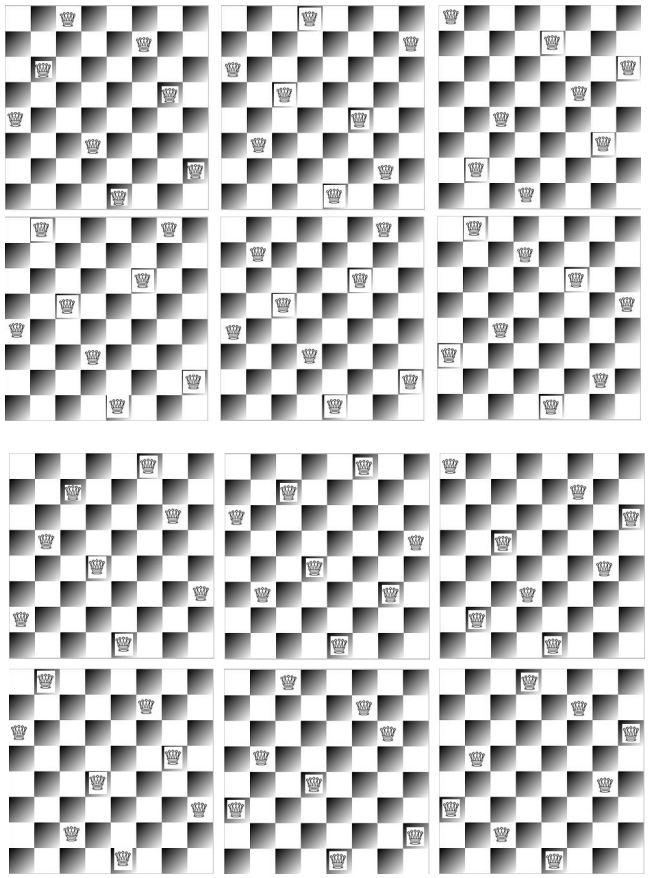
Problem n kraljica. Koliki je maksimalan broj kraljica koje se mogu postaviti na šahovsku ploču $n \times n$ tako da "nikoje dvije" ne napadaju jedna drugu?
Odgovor : Na ploči $n \times n$ može se postaviti n kraljica koje ne napadaju jedna drugu.

Broj načina na koje se n kraljica mogu postaviti na ploču $n \times n$ da ne napadaju jedna drugu je $(1, 0, 0, 2, 10, 4, 40, 92, 352, 724, 2680, 14200, 73712, 365596, 2279184, \ldots)$ za $n = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, \ldots$

Na slici 5.4 je dato 12 neekvivalentnih rješenja problema kraljica na šahovskoj ploči za $n = 8$. Od njih se rotacijama i simetrijama dobijaju svih 92 rješenja.

Na slici 5.5 dat je minimalan broj kraljica koje zauzimaju (napadaju) sva polja ploče 8×8.

Algoritmi rješenja problema $n kraljica$ se često koriste kao $benchmark$ za ispitivanje efikasnosti novih kompajlera i debagera.



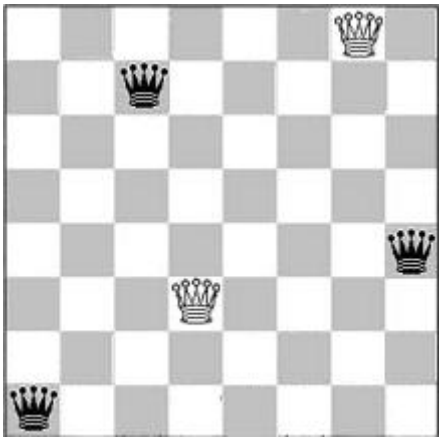
Slika 5.4: Dvanaest osnovnih rješenja problema 8 kraljica na tabli 8×8

Termin "backtrack" uveo je američki matematičar D. H. Lehmer 1950 g. Jezik za procesiranje stringova $SNOBOL$ je prvi imao ugradjene mehanizme za pretraživanje sa vraćanjem.

Pretraživanje sa vraćanjem se može primijeniti samo na probleme koji dozvoljavaju mogućnost parcijalnih rješenja, a pri tome postoji relativno brz test koji odredjuje kada će se doći do konačnog rješenja. U neuredjenom skupu backtracking je često puno brži nego algoritam grube sile koji rješava problem enumeracijom svih kandidata, jer koristeći metodu backtrackinga možemo u jednom testu eliminisati veliki broj kandidata.

Problemi na koje se prirodno primjenjuje metoda pretraživanja sa vraćanjem najčešće su NP-kompletni. Pokazali smo da je algoritam sortiranja sa vremenom $T(n) = O(n^2)$, loš algoritam za sortiranje, ali bi bilo koji polinomijalni algoritam za problem trgovačkog putnika, problem pakovanja ranca, ili za bilo koji NP-kompletan problem, predstavljao veliko otkriće.

Uspjeh u rješavanju instanci male dimenzije NP-kompletnih problema ne



Slika 5.5: Primjer pet kraljica koje pokrivaju sva polja na tabli 8×8

smije nas odvesti na pogrešan zaključak da je pretraživanje sa vraćanjem efikasna tehnika. U najgorem slučaju ono može generisati sve moguće kandidate u eksponencijalno ili još brže rastućem prostoru. Za uspješan rad se očekuje da će algoritam biti u mogućnosti da odsiječe dovoljno grana njegovog prostor-vrijeme drveta prije nego što mu nestane vremena ili memorije. Uspjeh algoritma varira ne samo od problema do problema, nego i od slučaja do slučaja jednog te istog problema. Postoji nekoliko trikova koji nam pomažu da redukujemo dužinu drveta prostora-vremena. U slučaju $n kraljica$ na šahovskoj tabli problem ima nekoliko simetrija koje nam dozvoljavaju da refleksijom ili rotacijom iz jednih rješenja dobijemo druga. Na primjer, ne moramo razmatrati postavljanje prve kraljice u zadnjih $\lfloor n/2 \rfloor$ kolona, jer svako rješenje sa prvom kraljicom u pravougaoniku $(1, i)$, $\lceil \frac{n}{2} \rceil \leq i \leq n$, može se refleksijom dobiti iz rješenja u kome je prva kraljica u pravougaoniku $(1, n - i + 1)$. Zahvaljujući ovoj činjenici moguće je skratiti prostor-vrijeme drvo za polovinu. Drugi trikovi se najčešće zasnivaju na sortiranju i uopšte, rearanžiranju podataka.

Ocjenu veličine drveta algoritma za pretraživanje sa vraćanjem je vrlo teško dobiti analitički. Jedna od ideja za ocjenu dužine drveta algoritma za pretraživanje zasniva se na uzimanju slučajnog puta u drvetu i računanju njegove dužine. Razmatranja na tu temu mogu se naći u [15].

5.3 Rekonstrukcija položaja tačaka iz datih rastojanja

Backtracking je korisan alat za rješavanje problema sa ograničenjima, kao što su razne slagalice, verbalni aritmetički problemi, sudoku i razne igrice. Pogodan je kao tehnika za parsiranje, za rješavanje problema pakovanja ranca i drugih problema optimizacije u kombinatorici. Pretraživanje u jezicima logičkog progra-

miranja često se bazira na backtrackingu. U ovoj sekciji ćemo, koristeći metodu pretraživanja sa vraćanjem, pokazati algoritam za rješavanje problema rekonstrukcije položaja tačaka iz datih rastojanja. Algoritam ima važne primjene u fizici i molekularnoj biologiji.

Ako je dato n različitih tačaka $p_1, p_2, \cdots, p_n$, tada one odredjuju $n(n-1)/2$ rastojanja $|x_i - x_j|$, $i, j \in \{1, \cdots, n\}$, $i \neq j$ (Slika 5.6).

Slika 5.6: Rastojanja tačaka na pravoj

Postavljamo direktan i obrnut problem.

- **Direktan problem.** Date su tačke $p_1, p_2, \cdots, p_n$. Skup rastojanja $d_1, d_2, \cdots d_k$ možemo konstruisati za $T = O(n^2)$ vrijeme. Za $T = O(n^2 \log n)$ možemo dobiti ova rastojanja sortirana.
- **Obrnut problem.** Data su rastojanja $d_1, d_2, \cdots d_k$. Treba konstruisati skup tačaka $p_1, p_2, \cdots, p_n$, koji im odgovara.

Pokazuje se da je obrnuti problem teži od direktnog problema. Množenje i faktorizacija brojeva je analogni primjer u kojem je rekonstrukcija teža od konstrukcije.

Razmotrimo primjer u kojem je skup rastojanja dat sa
$D = \{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 10\}$. Ovdje je $|D| = 15$, odakle imamo da je $n = 6$. Startujemo sa $x_1 = 0$. Jasno je da je $x_6 = 10$, jer je 10 najveće rastojanje u D (Slika 5.7).

Slika 5.7: Početna faza rješavanja

Sada možemo izbaciti 10 iz skupa D, pa imamo $D = D - \{10\}$. Drugim riječima $D = \{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8\}$. Pošto je $max(D) = 8$, slijedi daje $x_2 = 2$ ili $x_5 = 8$. Bez gubitka opštosti uzimamo $x_5 = 8$, jer $x_2 = 2$ daje simetrično rješenje. Sada imamo (Slika 5.8) $x_6 - x_5 = 2$, $x_5 - x_1 = 8$, pa prelazimo na $D = D - \{2, 8\} = \{1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7\}$.

Slika 5.8: Slučaj $x_5 = 8$

Maksimalno rastojanje u skupu D je sada $max(D) = 7$. Slijedi da je $x_4 = 7$ ili $x_2 = 3$. Odatle zaključujemo da razdaljine $x_6 - 7 = 3$ i $x_5 - 3 = 1$ moraju postojati u D, što je zaista tako. Sa druge strane, ako stavimo $x_2 = 3$, tada rastojanja $3 - x_1 = 3$, $x_5 - 3 = 5$, moraju biti u D. To je takodje tačno.

Slika 5.9: Slučaj za $x_4 = 7$

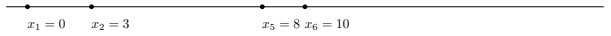
Probamo $x_4 = 7$ (Slika 5.9).

Sada je $D = \{2, 2, 3, 3, 4, 5, 5, 5, 6\}$. Zbog toga imamo $x_1 = 0$, $x_4 = 7$, $x_5 = 8$, $x_6 = 10$. Najveće preostalo rastojanje je 6. Odatle slijedi da je $x_3 = 6$ ili $x_2 = 4$.

Ako pretpostavimo da je $x_3 = 6$, onda slijedi da je $x_4 - x_3 = 1$, što je nemoguće, jer 1 nije više u D.

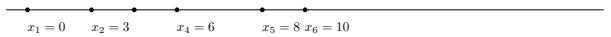
Ako pretpostavimo da je $x_2 = 4$, onda slijedi da je $x_2 - x_0 = 4$, $x_5 - x_2 = 4$, što je takodje nemoguće, jer se 4 pojavljuje samo jednom.

Pošto nismo uspjeli sa $x_4 = 7$, probamo $x_2 = 3$ (Slika 5.10). Ako i ovo ne daje rješenje, saopštićemo da rješenje ne postoji.



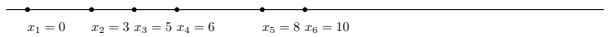
Slika 5.10: Slučaj za $x_2 = 3$

Sada je $D = \{1, 2, 2, 3, 3, 4, 5, 5, 6\}$, a kandidati su $x_4 = 6$ i $x_3 = 4$. Slučaj $x_3 = 4$ je nemoguć! Ako uzmemo $x_4 = 6$, onda dobivamo $D = \{1, 2, 3, 5, 5\}$ (Slika 5.11).



Slika 5.11: Slučaj za $x_4 = 6$

Jedina preostala mogućnost je $x_3 = 5$ i ona rješava problem, jer je $D = \emptyset$ (Slika 5.12).

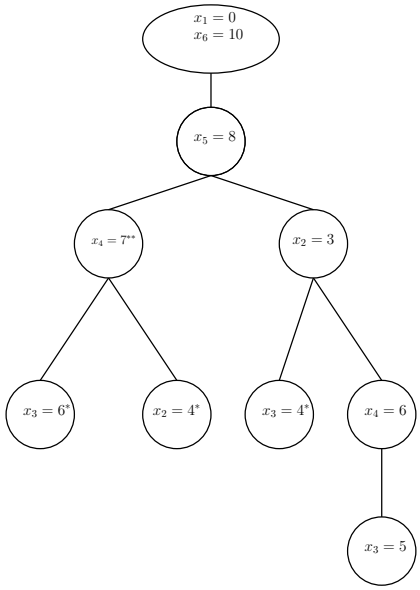


Slika 5.12: Slučaj za $D = \emptyset$

Sada ćemo napisati pseudo-kod, a zatim i kod za implementaciju opisanog algoritma. Korisno će nam poslužiti i slika 5.13.

```
/* Pseudo-kod za implementaciju algoritma odredjivanja
tacaka iz datih rastojanja */
int r_distance(int x[], dist_skup D, unsigned int n)
{
    x[1] = 0;
    x[n] = delete_max(D);
    x[n-1] = delete_max(D);
    if (x[n] - x[n-1] in D)
    {
        delete (x[n] - x[n-1], D);
        return stavi(x, D, n, 2, n - 2);}
else
    return FALSE;
}
```

```
/* Backtracking algoritam za postavljanje tacaka */
/* x[lijevi] ... x[desni]   */

/* x[1] ...x[lijevi - 1] i x[desni + 1] ... x[n] su
   vec postavljene. Ako postavi() vrati tacno, tada ce
   x[lijevi] ... x[desni] imati vrijednost
*/

int postavi(int x[], dist_set D, unsigned int n,
                int Lijevi, int Desni)
{   int d_max, found = FALSE;
        if D == \emptyset then
        return TRUE;
        d_max = find_max(D);

/* Provjeri x[Desni] == d_max        */

if ( | x[j] - d_max | in D za svako 1 <= j < Lijevi
            i Desni < j <= n)
    {
        x[Desni] = d_max;
        for (1<= j < lijevi, Desni < j <= n)
            delete (|x[j] - d_max |, D);
        found = postavi (x, D, n, lijevi, Desni);

    if (!found) /*Backtrack */
      for ( 1<=j < lijevi, Desni < j <=n)
      /* vrati obrisane  */
        insert( |x[j] - d_max|, D);

/* Ako je propao prvi pokusaj, pokusati
    x[Lijevi] = x[n] - d_max    */

if (!found && ( \x[n] - d_max - x[j]| in D
    za svako 1 <= j < Lijevi i Desni < j <= n ))
    x[Lijevi] = x[n] - d_max;

    for (1 <= j < lijevi, Desni < j <=n)
        delete ( |x[n] - d_max - x[j]|, Desni);

    found = postavi(x, D, n, + 1, Desni);
    if (!found)     /* Vrati pobrisano  */
        for (1< Lijevi, Desni < j <= n)
            insert (|x[n] - d_max - x[j]|, D);
        return found;
    }
```



Slika 5.13: Drvo algoritma za dobivanje tačaka iz datih rastojanja

Glava 6

Strukture podataka

U kompjuterskim naukama je generalno prihvaćeno da su strukture podataka od vitalnog značaja u programiranju. Ipak, koncenzus mišljenja nije dostignut. Često se u programerskoj praksi strukture podataka ne tretiraju na pravi način, a tek su u savremenim jezicima implementirani alati za rad sa strukturama podataka. Medju teoretičarima programiranja bilo je puno pokušaja da se definiše semantika programskih konstrukcija, kao što su veze, procedure, korutine, paralelni procesi i druge konstrukcje. Dosadašnje formalne teorije struktura podataka daju alate koji omogućuju identifikovanje (specifikaciju) struktura podataka, ali po pravilu im nedostaju alati za dokazivanje osobina programa.

Ovdje predstavljamo elemente praktične specifikacije struktura podataka, a takodje dajemo rudimente aksiomatskog zasnivanja apstraktnih tipova podataka. Veze izmedju specifikacije struktura podataka i osobina programa ovom prilikom nismo eksplicitno razmatrali.

6.1 Dinamičke strukture podataka

Tipična implementacija dinamičkog skupa:
Svaki element je predstavljen nekim objektom, a polja tog objekta mogu biti posjećena i manipulisana pomoću pokazivača na objekat.
Razlikujemo:

KEY (ključ)

Satelitski podaci

U našoj implementaciji nećemo koristiti satelitske podatke.

Operacije

Na dinamičkim strukturama razlikujemo dvije grupe operacija:
Queries (upitne, selektorske) operacije i
Modifikatorske operacije.
Search(S,k) je operacija koja vraća key$[x] = k$ ili NIL.

Insert(S,x) ubacuje element x u skup S.
Delete(S,x) briše element x iz skupa S.
Minimum(S) vraća minimum S.
Maximum(S) vraća najveći element iz S.
Successor(S,x) vraća NIL ili elemenat neposredno veći od x.
Predecessor(S,x) vraća NIL ili elemenat neposredno manji od x.
Dajemo sljedeće primjere:
Rječnik (Dictionary)
Rječnik je struktura za konačne skupove sa operacijama: insert, delete, member. Rječnici su jedna od najčešće prisutnih struktura. Koristimo ih kada god:

- pitamo je li neki elemenat iz univerzuma u datom konačnom skupu,
- proširujemo konačan skup ubacivanjem novog elementa ili
- brišemo neki elemenat iz konačnog skupa.

Postoje brojni primjeri primjene rječnika, kao na primjer u sistemima biblioteka, kontroli sadržaja memorije, i tako dalje. Takodje postoje strukture koje su proširenje strukture rječnika.

Rječnici formiraju apstraktan tip podataka pošto oni mogu biti implementirani na različite načine. Ovdje ćemo opisati algebarsku strukturu rječnika, ali nećemo razvijati formalizovanu teoriju te strukture. Slično ćemo postupiti i sa drugim strukturama.

Definicija 6.1. *Algebarska struktura se naziva rječnik kada god se njen nosač sastoji od dva disjunktna skupa E, S koji se nazivaju sorte, a ima sljedeće operacije: $empty : S \to B_0$, $member : E \times S \to B_0$, $insert : E \times S \to S$, $delete : E \times S \to S$, $amember : S \to E$. Pri tome je $amember$ parciijalna operacija definisana akko njen argument nije prazan, a struktura zadovoljava sljedeće postulate:*

(P1) $(\neg empty(s) \Rightarrow member(amember(s), s))$,

(P2) $empty(s)$ *akko ne postoji elemenat* e *takav da je* $member(e, s)$, ,

(P3) za svako s *instrukcija* $s := delete(amember(s), s)$ *može biti ponovljena samo konačan broj puta dok* s *postane prazan,*

(P4) za svako $e \in E$, *za svako* $s \in S$
$member(e, insert(e, s))$
$\neg member(e, delete(e, s))$,

(P5) za svako e, e', s
$(e' \neq e \Rightarrow (member(e', s) \equiv member(e', insert(e, s))))$

(P6) za svako e, e', s
$(e' \neq e \Rightarrow (member(e', s) \equiv member(e', delete(e, s)))).$

Rječnik ćemo zadavati kao strukturu

$$< E \cup S, \text{ empty, Insert, Delete, Member, amember} >,$$

pri čemu je skup $E \cup S$ nosač strukture i na njemu su zadate navedene operacije. U izlaganju nećemo razvijati pojam polisortnih struktura, ali ćemo ga koristiti u

primjerima i smatrati da je poznat iz matematičke logike.

Stek (Stack , $LIFO$) je struktura sa "Last In First Out" (" zadnji u prvi iz ") osobinom. Univerzum strukture podataka stek sastoji se od dva disjunktna skupa E i S. Elemente S nazivamo $stek$, dok će elementi E biti jednostavno nazivani $elementi$. Osnovne relacije i operacije na sistemu stekova su sljedeće:

$=$ identitet u E,

$empty$, poseban podskup od S, $empty \subset S$,

$push : E \times S \to S$,

$pop : E \times S \to S$,

$top : S - empty \to E$.

Svaki sličan relacioni sistem sa sličnom signaturom ćemo zvati *sistem podataka stekova* ako zadovoljava sljedeće uslove:

(P1) Za svaki stek s postoji neka iteracija pop operacija takva da je rezultat prazan stek $(\forall s \in S)(\exists i \in \mathbb{N})\, empty(pop^i(s))$.

(P2) Za svaki neprazan stek s imamo: $s\, je\, jednako\, push(top(s),\, pop(s))$, za svaki elemenat e i svaki stek s.

(P3) $e = top(push(e, s))$.

(P4) s je jednako $pop(push(e, s))$.

(P5) $\neg empty(push(e, s))$.

Stek se može se zadati nosačem $E \cup S$ i operacijama na skupu $E \cup S$, kao uredjena sedmorka

$$< E \cup S,\ \text{length, push, pop, top, bottom, empty} > .$$

Operacija $push$ odgovara operaciji $Insert$ i stavlja element na vrh steka.

Operacija pop odgovara operaciji $Delete$ i briše element sa vrha steka.

Operacija top daje vrh steka.

Operacija $bottom$ daje dno steka.

Operacija $empty$ ispituje da li je stek prazan.

Za operacije $empty$, $push$ i pop u listingu je naveden pseudo-kod.

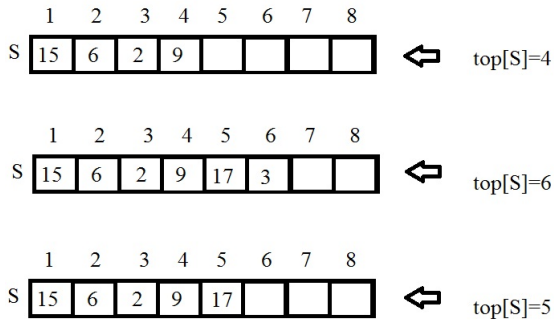
Na slici 6.1 je prikazana implementacija steka $S\,[1...top\,[S]]$ listom i operacija top na steku. Sa slike se mogu vidjeti i sljedeće operacije na steku:

$S\,[1]$ - (dno steka, bottom),

$S\,[top]$ - vrh steka

top $[S] = 0$, prazan (empty) stek

Ako na popunjen stek pokušamo staviti novi element, onda ćemo dobiti poruku o grešci **stack overflow**. Slično, ako sa praznog steka pokušamo skinuti element, onda ćemo dobiti poruku o grešci **stack underflow**.



Slika 6.1: Operacija *top* na steku

```
STACK_EMPTY(S)
    if top[S]=0
        then return TRUE
    else return FALSE
PUSH(S,x)
    top[S]<−top[S]+1
        S[top[S]]<−x
POP(S)
    if STACK_EMPTY(S)
        then error "underflow"
    else top[S]<−top[S]−1
return S[top[S]+1]
```

Red za čekanje (Queue, $FIFO$), je struktura sa "First In First Out" ("prvi u prvi iz") osobinom. Red za čekanje može se opisati kao struktura

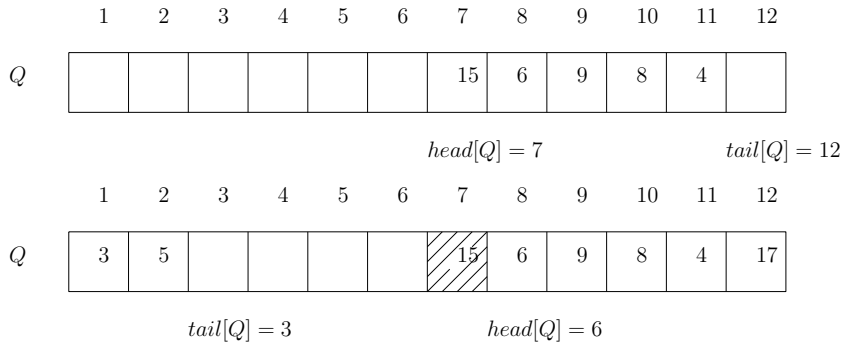
$$< E \cup Q, \text{em, enqueue, dequeue, first} > .$$

Ako varijable sorte E označimo sa $e, e', \ldots$, a varijable sorte Q sa $q, q', q_1, \ldots$, onda specifična signatura teorije i tumačenje na modelu koji se sastoji od nizova, može biti dato sa:

$em : Q \to B_0$, $em(s)$ znači da je niz prazan,
$inqueue : E \times Q \to Q$, dodaje element e u niz s na kraj niza,
$dequeue : Q \to Q$, briše prvi elemenat iz s,
$first : Q \to E$, daje prvi elemenat niza s, ako niz s nije prazan,
$=_E E \times E \to B_0$, je jednakost na E,
$=_Q Q \times Q \to B_0$, je jednakost na Q.
Aksiome reda za čekanje:
Ax1 $\mathbf{while}\, \neg em(q) \{q = dequeue(q)\} \mathbf{return\ true}$,
Ax2 $(em(q) \Rightarrow (q =_Q dequeue(inqueue(e, q))))$,
Ax3 $(\neg em(q) \Rightarrow inqueue(e, outqueue(q)) =_Q dequeue(inqueue(e, q)))$,
Ax4 $(em(q) \Rightarrow (e =_E first(inqueue(e, q))))$,
Ax5 $(\neg em(q) \Rightarrow first(inqueue(e, q)) =_E first(q))$,
Ax6 $\neg em(inqueue(e, q))$,

Ax7 $q =_Q q' \equiv \{q_1 = q; q_2 = q'; bool = \textbf{true};$
$\textbf{while}\, \neg em(q_1) \wedge \neg em(q_2) \wedge bool\{$
$\textbf{if}\, first(q_1) \neq first(q_2)\, \textbf{then}\, bool = \textbf{false};$
$q_1 = dequeu(q_1); q_2 = dequeu(q_2);$
$\}$
$\}$
$\textbf{return}\, (bool \wedge em(q_1) \wedge em(q_2))$

Pri tome operacija $enqueue$ odgovara operaciji $Insert$ i umeće element na kraj reda za čekanje, a operacija $dequeue$ odgovara operaciji $delete$ i briše element sa početka reda za čekanje. Na slici 6.2 prikazana je implementacija reda za čekanje korištenjem liste. U listingu je naveden pseudo-kod za operacije $enqueue$ i $dequeue$.



Slika 6.2: Implementacija reda za čekanje pomoću liste

```
Enqueue(Q,x)
    Q[tail[Q]] <- x
if tail [Q]=length[Q]
    then tail[Q] <- 1
    else tail [a] <-tail [a]+1
Dequeue(Q)
x <- Q[head[Q]]
if head[Q]=length[Q]
    then head[Q] <- 1
    else head[Q] <- head[Q]+1
return x
```

6.2 Povezane (linkovane) liste

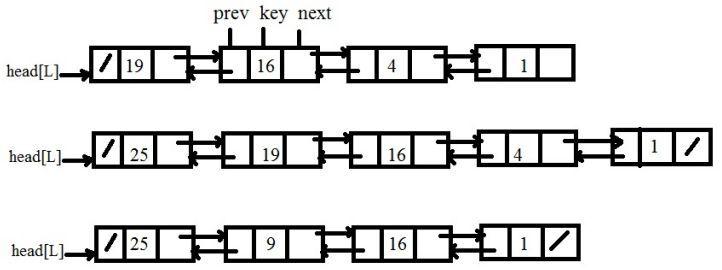
Povezane liste su linearno uredjeni objekti. Razlikovaćemo jednostruko povezane, dvostruko povezane i cirkularne liste. U dvostruko povezanoj listi imamo sljedeće operacija $next[x]$ i $prev[x]$. Na slici 6.3 prikazana je dvostruko povezana lista sa dodavanjem elementa 25 na glavu i brisanjem elementa 4 iz liste. Primjećujemo da

- Ako je $prev[x] = NIL$, onda je x početak liste.
- Ako je $head[L] = NIL$, onda je lista L prazna.

Razmatraćemo sortirane i nesortirane liste.

```
LIST_SEARCH (L,x)
    x <- head[L]
    while x< >NIL and key[x]<>k
        do x <- next [x]
return x
```

Složenost traženja elementa liste je u najgorem slučaju (worst case) $T(n) = \Theta(n)$.



Slika 6.3: Dvostruko povezana lista

Navodimo i proceduru $LIST\_INSERT(L, x)$ za umetanje u listu L elementa x, kao i proceduru $LIST\_DELETE(L, x)$ za brisanje elementa x iz liste L.

```
LIST_INSERT(L,x)
    next[x] <- head[L]
    if head[L]< > NIL
        then prev[head[L]] <- x
head[L] <- x
prev[L] <- NIL
```

```
LIST_DELETE (L,x)
    if prev[x]< >NIL
        then next[prev[x]]<-next[x]
        else head[L]<-next[x]
    if next[x]< >NIL
        then prev[next[x]]<-prev[x]
```

U najgorem slučaju, složenost umetanja u listu ili brisannja iz liste, je $T(n) = \Theta(n)$.

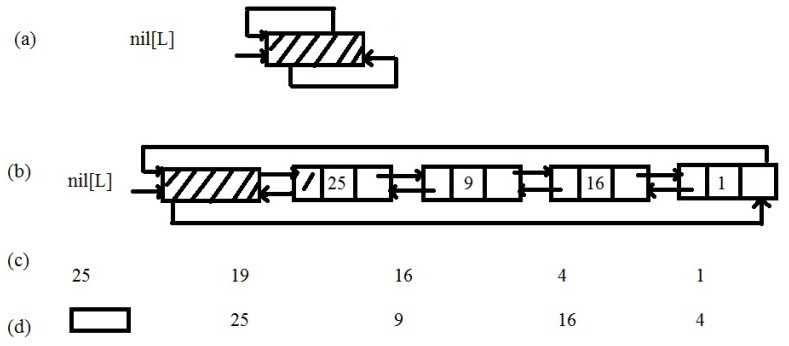
6.3 Graničnici (Sentineli)

U sljedećem primjeru opisujemo brisanje iz liste koristeći *graničnik sentinel*.

```
LIST_DELETE'␣(L,x)
next[prev[x]]␣<-␣next[x]
prex[next[x]]␣<-␣prev[x]
```

$Graničnik\,(Sentinel)$ je uvedeni element da olakša rad sa graničnim vrijednostima. Na primjer, ako pretražujemo listu i ne nadjemo vrijednost, zaustavićemo se na graničiku (sentinelu).

Primjer 6.1. *Na slici 6.4 je prikazano korištenje sentinel elementa u radu sa dvostruko povezanom listom. Sa $nil[L]$ smo označili NIL objekat.*



Slika 6.4: Rad sa listom korištenjem graničnog (sentinel) elementa

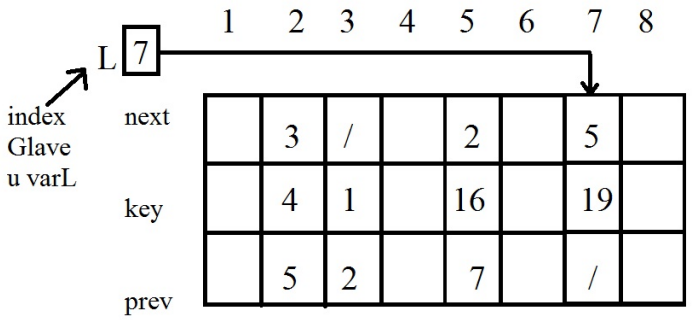
Navodimo procedure $LIST\_SEARCH'$ i $LIST\_INSERT'$ koje, koristeći *graničnik (sentinel)* pretražuju listu L i umeću u listu L elemenat x.

```
LIST_SEARCH'
x <- next[nil[L]]
while x <> nil[L] and key [x] <> k
do x <- next[x]
return x
```

```
LIST_INSERT' (L,x)
next[x] <- next[nil[L]]
prev[next[nil[L]]] <- x
next[nil[L]] <- x
prev[x] <- nil[L]
```

6.3.1 Implementacija pointera i objekata matricom

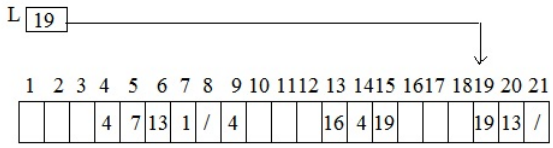
Na slici 6.5 prikazana je implementaciju pointera matricom. Indeks glave je zapisan u varijabli L. U svakoj koloni se nalaze indeks sljedećeg elementa, ključ i prethodni element.



Slika 6.5: Implementacija pointera matricom

6.3.2 Implementacija pointera jednodimenzionalnim nizom

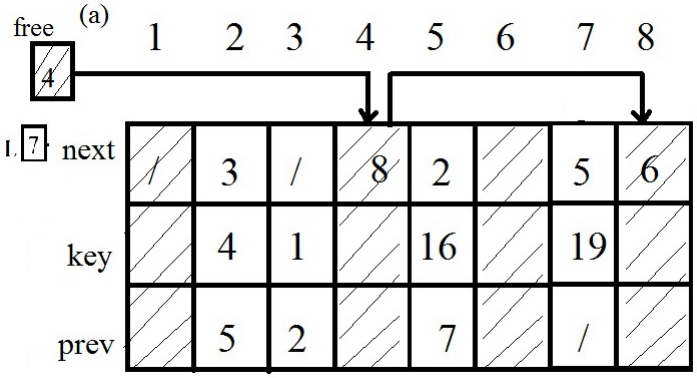
Na slici 6.6 dat je primjer implementacije pointera nizom. Ideja je da se indeks glave zapiše u varijablu L, a da se onda u tri polja, počevši od indeksa glave, napišu ključ, indeks sljedbenika i indeks prethodnika glave. Zatim se isto radi sa poljem sljedbenika i tako redom.



Slika 6.6: Implementacija pointera jednodimenzionalnim nizom

6.3.3 Alociranje i oslobadjanje elemenata.

Garbage collector-i vode računa o neiskorištenim lokacijama i vraćaju ih u evidenciju slobodne memorije. Na slikama 6.7, 6.8 i 6.9 pokazano je upravljanje slobodnom i zauzetom memorijom pomoću dva pokazivača.



Slika 6.7: Tabela za alociranje i oslobadjanje memorije

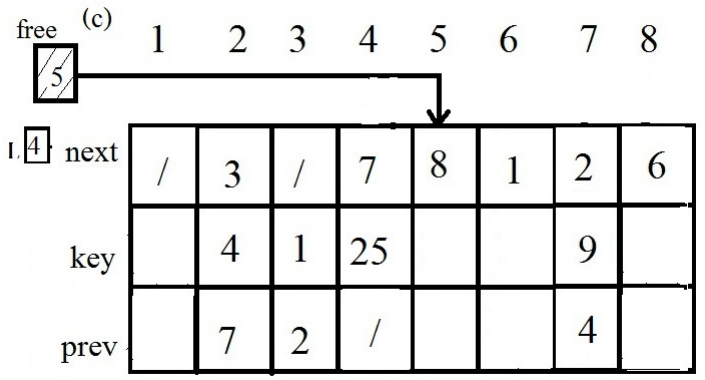
Slobodni (free) objekti su u jednostruko povezanoj listi *slobodna lista*.

Sa slika se vidi da su nam potrebni samo next i head pokazivači da bismo vodili evidenciju slobodne memorije.

Slobodna lista je stek. Na slici 6.10 smo prikazali programe zauzimanja i oslobadjanja memorije u računaru, a zatim smo naveli njihove listinge.

U sljedećem programu je složenost $T(n) = O(1)$.

```
ALLOCATE_OBJECT ()
if free = NIL
     then error "out␣of␣space"
     else x<- free
          free <- next[x]
return x
FREE_OBJECT(x)
     next(x) <- free
     free <- x
```



Slika 6.8: Dodavanje novog elementa na polje sa indeksom četiri

6.4 Binarno drvo

Neka je A skup elemenata koje ćemo zvati $atomi$. Dajemo specifikaciju strukture binarnog drveta sa atomima pridruženim listovima drveta.

Struktura ima dvije sorte:

A - sorta atoma,

T - sorta drveta.

Sorte A i T nisu disjunktne. Podrazumijevamo da je $A \subset T$.

Operacije strukture drveta su sljedeće:

$c : T \times T \to T$,

$e : T \to B_0$,

$a : T \to B_0$,

$l : T \to T$,

$r : T \to T$.

Operacije l, r su parcijalne operacije i nisu definisane ako je argument atom.

Aksiome binarnih drveta su:

TR1 $(\forall t \in T)(a(t) \vee e(t) \vee t = c(l(t), r(t)))$;

TR2 $(\forall t_1, t_2 \in T) l(c(t_1, t_2)) = t_1$;

TR3 $(\forall t_1, t_2 \in T) r(c(t_1, t_2)) = t_2$;

TR4 $(\forall t_1, t_2 \in T) \neg e(c(t_1, t_2)) \wedge \neg a(c(t_1, t_2))$;

TR5 $(\forall t \in T)$**while** $\neg e(t) \wedge \neg a(t)\{$

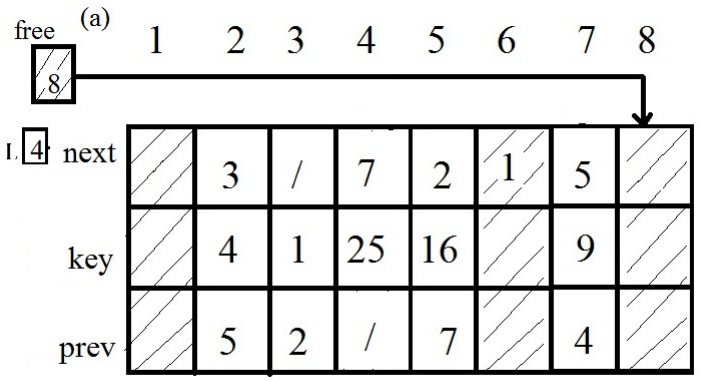
if $e(l(t)) \vee a(l(t))$ **then** $t = r(t)$;

else $t = c(l(l(t)), c(r(l(t)), r(t)))$;

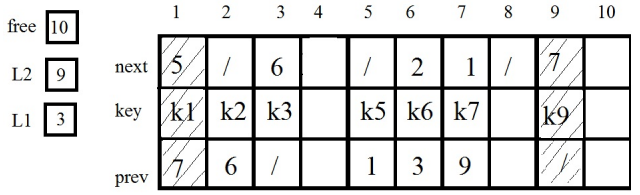
return true;

TR6 $(\forall t_1, t_2 \in T)((e(t_1) \wedge e(t_2)) \Rightarrow t_1 = t_2$.

Standardan model za ove aksiome je skup S-izraza. S-izrazi čine sematičku bazu za programski jezik "čisti"$LISP$.



Slika 6.9: Uklanjanje elementa iz evidencije



Slika 6.10: Oslobadjanje i zauzimanje memorije u računaru

Definicija 6.2. *Skup S-izraza nad skupom A je najmanji skup izraza takav da:*

1. *sadrži skup $A \cup \{nil\}$;*

2. *za svaka dva S-izraza τ_1 i τ_2, izraz $(\tau_1 \cdot \tau_2)$ je takodje u skupu S-izraza.*

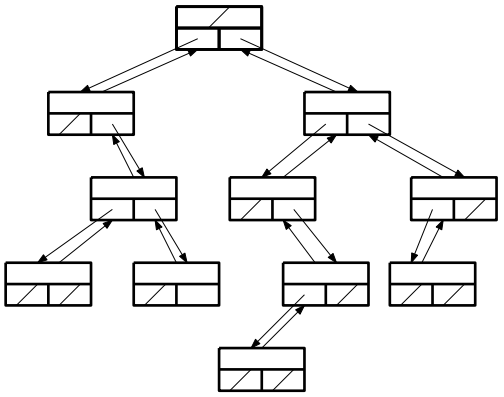
Dajemo i specifikaciju drveta koja se često koristi u teoriji skupova. Struktura podataka

$$\mathcal{A} =< A, f, a_0 >,$$

gdje je $a_0 \in A$, $f : A \to A$, naziva se $drvo$ ako i samo ako zadovoljava aksiom:

$$(\forall a \in A)\, \mathbf{while}\, a \neq a_0\, \{a = f(a)\}; \mathbf{return\ true}.$$

Na slici 6.11 prikazan je implementacija binarnog drveta pomoću pokazivača. Pokazivač označimo sa p. Imamo lijevo i desno poddrvo drveta i pokazivače $left$ i $right$, respektivno. Ako je x korjen (root) drveta, onda je $p[x] = NIL$ i obrnuto. Ako je $root[T] = NIL$, onda je drvo prazno i obrnuto.



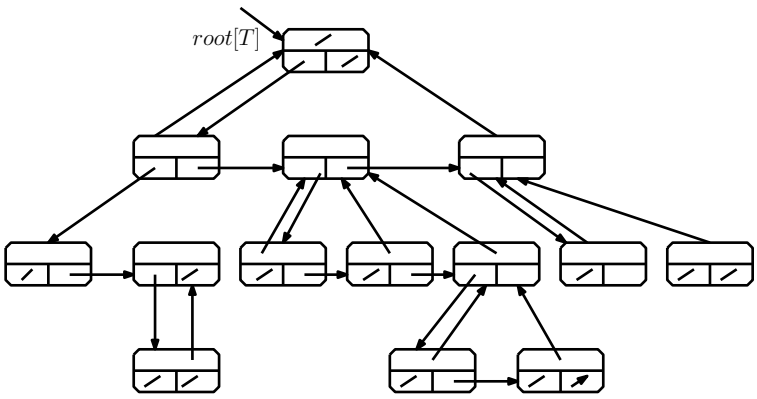
Slika 6.11: Primjer implementacije binarnog drveta pomoću pokazivača

6.5 Neograničeno grananje

Na slici 6.12 data je "$left\text{-}child, right\text{-}sibling$" reprezentacija drveta sa beskonačnim grananjem. Pri tome

1. $left - child[x]$ ($lijevo - dijete[x]$)pokazuje na krajnje lijevo dijete od x
2. $right - sibling[x]$ ($desni - brat$) pokazuje na brata neposredno sa desne strane

Ako je $left - child[x] = NIL$, onda x nema djece. Za $rightmost$ ($krajnje\,desno$) dijete roditelja važi $right - sibling[x] = NIL$.



Slika 6.12: Reprezentacija beskonačnog grananja

Navodimo funkcije za reprezentaciju opisanog beskonačnog grananja drveta.

(a) Funkcija KRAJNJE_LIJEVO_DIJETE

```
vrh function KRAJNE_LIJEVO_DIJETE ( n: vrh; T: DRVO ){
    // vraca krajnje lijevo dijete vrha n na drvetu T
 int L; // kursor na pocetak liste djece vrha n
    {
L = T.header[n];
if L == 0 then // n je list drveta
    return (0);
else
    return (cellspace[L].vrh)
    }; // KRAJNJE_LIJEVO_DIJETE
}
```

(b) Funkcija RODITELJ

```
vrh function RODITELJ ( n: vrh; T: DRVO){
    // vraca roditelja vrha n u drvetu T

 vrh p; // radi za moguce roditelje vrha n
 position i; // radi do dna liste djece vrha p
    {
 for (p = 1;  not maxnode; next_node){
    i = T.header[p];
    while i <> 0 do // ispitaj je li n medju djecom p
if cellspace[i].node == n then
     return (p)
else
     i = cellspace[i].next
}
return (0) //vraca NULL vrh ako roditelj nije nadjen
    }; //RODITELJ
```

Implementacija "$left - right$" $child1, \ldots, childk$, nije pogodna reprezentacija za beskonačno grananje.

Neke druge moguće reprezentacije drveta sa beskonačnim grananjem su:

1. Heap sa jednim nizom (array) i indeksom.

2. Reprezentacija sa samo pokazivačem na roditelja za drveta koja prolazimo samo prema korjenu.

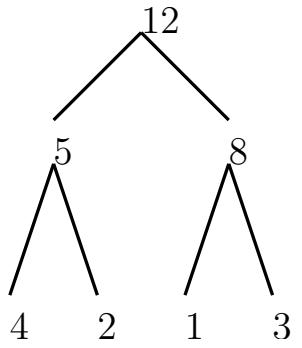
6.6 Heap sort (sortiranje drvetom)

Hip (heap) je binarno drvo koje je

1. Balansirano sa mogućim izuzecima čvorova na dnu drveta
2. Na svakom putu od korjena drveta do nekog lista, vrijednosti u čvorovima su uredjene

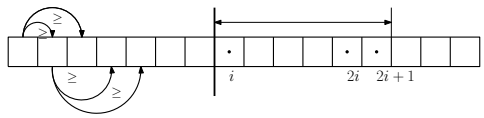
Skrećemo pažnju da struktura podataka binarni hip, koju ovdje razmatramo, nije isto što i specijalna memorija poznata kao hip, iz koje sistem alocira memoriju potrebnu za kreiranje nekog objekta sa new.

Na slici 6.13 dat je primjer hipa.



Slika 6.13: Primjer hipa

Niz (array) možemo interpretirati pomoću hipa. Pri tome su djeca čvora i na pozicijama $2i$ i $2i+1$ i imaju manje vrijednosti nego čvor roditelj (slika 6.14).

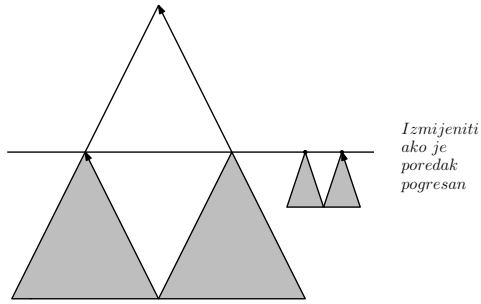


Slika 6.14: Interpretacija linearnog niza pomoću hipa

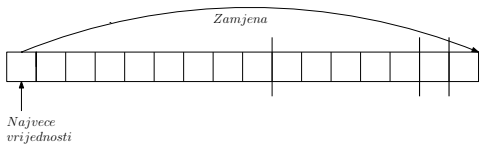
Postavlja se pitanje kako neki niz možemo pretvoriti u hip. To radimo tako što startujemo od listova i pravimo sve veće i veće hipove. Svako pod-drvo koje se sastoji od jednog lista, već formira hip. Pod-drvo visine h pretvaramo u hip razmjenjujući korijen i veći elemenat sinova korjena, ako je korijen manji od nekog od njih. Radeći tako možemo pokvariti hip visine $h-1$, koji u tom slučaju moramo ponovo pretvoriti u hip. U našem programu to radi funkcija $heapify$. Na slici 6.15 data je šema rada procedure $heapify$. Neka je $T(h)$ vrijeme potrebno za izvodjenje $heapify$ na čvoru visine h. Tada je $T(h) < T(h-1) + c$, za neku konstantu c.

Odatle slijedi da je $T(h) = O(h)$. Program poziva $heapify$, jednom, za svaki čvor. Zbog toga je vrijeme potrebno za pravljenje hipa jednako sumi visina čvorova, nad svim čvorovima. Ali najviše $\lceil \frac{n}{2^{i+1}} \rceil$ vrhova su na visini i. Dakle, ukupno vrijeme potrebno našem programu da napravi hip je $T(n) = \sum_{1}^{\log n} i\frac{n}{2^i} = O(n)$.

Kada su elementi niza aranžirani u hip, elemente sa korjena možemo jedan po jedan, uklanjati. To radimo tako što razmijenimo $A[1]$ i $A[n]$, a zatim $A[1], A[2], \ldots, A[n-1]$ aranžiramo u hip. Zatim razmijenimo $A[1]$ i $A[n-1]$, a elemente $A[1], A[2], \ldots, A[n-2]$ aranžiramo u hip. Na kraju će elementi $A[1], A[2], \ldots, A[n]$ biti sortirani (Slika 6.16). Vrijeme izvodjenja procedure $heapify(1, i)$ je $O(i)$, pa je vrijeme izvodjenja programa za sortiranje hipom $T(n) = O(n \log n)$.



Slika 6.15: Pretvaranje linearnog niza u hip



Slika 6.16: Ilustracija sortiranja hipom

U sljedećem listingu dajemo $C++$ kod sortiranja drvetom.

```
template <class T>
void heapify (T*a, int k, int n)
{ T t= a[k];
while (k<n/2)
{ int j=2*k+i; //pretvara j u najstarijeg potomka k
if(j+1<n && a[j]<a[j+1]) ++j;
if(t>a[j]) break;
a[k]=a[j];
k=j;
}
a[k]=t;
}
template <class T>
void sort(T*a, int n)
{for (int i=n/2-1;i>=0;i--)
heapify(a,i,n);
for(i=n-1;i>0;i--)
{swap(a[0],a[i]);
//Invarijanta: elementi a[i:n-1] su u korektnom poretku
heapify(a,o,i);//Invarijanta: podniz a[o:n-1] je heap
}
}
template <class T>
void swap (T&x, T&y)
{
T temp=x;
x=y;
y=temp;
}
```

6.7 Nizovi (Arrays)

Ova često korištena struktura dozvoljava nam da razmatramo konačne nizove elemenata date sorte E zajedno sa operacijama: pristupi i-toj komponenti niza i abdejtuj i-tu komponentu niza. Ideja izgleda jednostavno, ali ipak postoje nezgodne skrivene zamke u vezi sa nizovima.

Pod *strukturom podataka jednodimenzionalnih nizova* podrazumijevamo sistem

$$< E \cup Ar \cup \mathbb{N}, put, get, lower, upper, newar, succ, empty, emptyar, 0, =, \leq >,$$

gdje su $E, Ar, \mathbb{N}$ skupovi struktura podataka. $\mathbb{N}$ je skup prirodnih brojeva, E je neprazan skup elemenata, Ar je neprazan skup nnizova. Operacije na nizovima su:

$put : A \times \mathbb{N} \times E \to Ar$, $get : Ar \times \mathbb{N} \to E$,
$lower : Ar \to \mathbb{N}$, $upper : Ar \to \mathbb{N}$,
$newar : \mathbb{N} \times \mathbb{N} \to Ar$, $empty \in E$,
$emptyar \in Ar$, $succ : \mathbb{N} \to \mathbb{N}$,
$0 \in \mathbb{N}$.

relacija $=$ je relacija identičnosti na nizovima, a relacija $\leq$ je relacija poretka na skupu prirodnih brojeva $\mathbb{N}$.

Pošto ovdje ne izgradjujemo formalnu semantiku struktura izostavićemo aksiome za nizove.

6.8 Heš-tabele

6.8.1 Uvod

U bilo kojoj oblasti primjene računara, upravljanju u ekonomiji, medicinskoj dijagnostici, nauci, tehnici, prevodjenju sa jednog prirodnog jezika na drugi, lako se uvjerimo da efikasnost rješavanja problema u velikoj mjeri zavisi od dobrog pristupa projektanata i programera pitanju organizacije skupova podataka, čuvanju podataka u memoriji i od algoritama za pronalaženje informacije. Posebno su važne metode za efikasno pretraživanje velikih skupova podataka. U ovoj sekciji ćemo razmotriti jednu klasu takvih metoda. Opisujemo skup metoda organizacije i algoritme transformacije skupova podataka koji se zasnivaju na zajedničkom konceptu heš (raspršene) tabele.

Pretpostavimo da treba napraviti spisak stanovnika države koja ima n stanovnika. Ako napravimo spisak u obliku niza, onda je matematičko očekivanje broja pokušaja da se nadje neka osoba $E = \frac{n+1}{2}$. Ako datoteku sortiramo u odnosu na prezimena, tada ćemo moći primijeniti *binarno pretraživanje*. Za pronalaženje neke osobe sada se očekuje $E = \log_2 n$ pokušaja. Pri tome je binarno pretraživanje spor algoritam, a dodatno traži dosta vremena za sortiranje. Ubacivanje novog sloga je takodje nezgodna operacija. Mogli bismo traženi skup organizovati u obliku liste. Takva struktura takodje traži sekvencijalno pregledanje zapisa, ali bi se u nekim drugim strukturama sa pokazivačima, kao što je na primjer, drvo, možda moglo raditi efikasnije. Sa druge strane, manipulisanje strukturama nije ni zgodno ni brzo.

Datoteke zasnovane na indeksu obezbjedjuju brži dostup do svakog sloga. Vrijeme se gubi samo na pretraživanje indeksa, koji može zauzeti značajan dio memorije. Nije teško primijetiti, da je indeks neka funkcija definisana pomoću tabele. Argumenti su ključevi slogova, a vrijednosti su adrese memorije. Ako uspijemo funkciju definisati algoritamski, onda možemo napustiti indekse. U ovoj sekciji će biti govora baš o takvim funkcijama. Fragment primjera naše datoteke predstavljen je na slici 6.18. Skupovi podataka koji su organizovani na dati način nazivaju se *heš (raspršene)* tabele.

Heš tabele su univerzalna metoda programiranja i primjenjuju se u bazama podataka, u sistemima pretraživanja informacije za identifikaciju relevantnih dokumenata, u konstruisanju kompajlera kao efikasno organizovana tablica identifikatora, u numeričkim problemima sa razrijedjenim matricama i slično. Heš tabele su veoma pogodan način organizacije raznih skupova podataka, koji obezbjedjuje brz dostup do podataka u situacijama u kojima se traže pojedinačni elementi sa jedinstvenim identifikatorom. Ako se traže slogovi koji zadovoljavaju niz uslova, onda neke druge strukture podataka mogu biti efikasnije.

U formalnom zasnivanju struktura podataka heš tabela sastoji se od pet sorti:

$$\mathbb{N}, E, Q, Ar, HT.$$

Jezik teorije heš tabela je unija jezika redova za čekanje i jezika nizova (arrays). Dodatno postoji funktor $h : E \to \mathbb{N}$. U teoriji se razmatraju redovi za čekanje elemenata iz skupa E i nizovi redova za čekanje. Na aksiome redova za čekanje i nizova redova za čekanje dodaju se aksiome koje definišu operacije rječnika.

6.8.2 Rasporedjivanje ključeva po slotovima

Ako razmještamo n slogova u n slotova memorije, tada će vjerovatnoća da nema ni jednog konflikta biti $P = \frac{n!}{n^n}$. Ta je vjerovatnoća vrlo mala i iznosi *0,000363*, za $n = 10$. Ako povećamo broj slotova memorije na $p > n$, tada, po teoriji vjerovatnoće, imamo $P = \frac{p(p-1)...(p-n+1)}{p^n}$. Ako za deset slogova obezbijedimo trinaest slotova memorije, tada će vjerovatnoća bezkonfliktnog smještanja u heš tabeli biti *0,0074*. I dalje je mala, ali je preko dvadeset puta veća od odgovarajuće vjerovatnoće u prethodnom primjeru.

U praksi se uvijek primjenjuje višak slotova od 5% do 20%. Taj višak ćemo izražavati kroz "faktor opterećenja" ("load factor")

$$\alpha = \frac{n}{p} = \frac{\text{Broj zapisa u heš tabeli}}{\text{Broj slotova u tabeli}}.$$

Pred programerom koji želi implementirati heš tabele, postavljaju se dva zadatka:

1. Izbor odgovarajuće funkcije heširanja

2. Izbor metode rješavanja konflikta

Funkcije heširanja mogu biti nezavisne od ključa (Randomizacija, Izdvajanje sredine kvadrata po Fon Nojmanovoj metodi, Metoda sabiranja ključa, Metoda sabiranja dijelova ključa po modulu 2, metoda dijeljenja, Metoda množenja i slično) i mogu biti funkcije heširanja zavisne od ključa.

Metoda randomizacije generiše slučajan broj od ključa k, kao argumenta, to jest $rand(k)$ i u dobivenu adresu smješta slog sa ključem k. Metoda Fon Nojmana uzima brojeve koji odgovaraju sredini ključa, diže ih na kvadrat, pa slog smješta na adresu koja se dobije uzimanjem nekoliko brojeva iz dobivenog kvadrata. Metode dijeljenja i množenja ćemo obraditi u ovoj sekciji, kasnije. Za sada ćemo reći da je srednji broj kolizija koje nastaju metodom dijeljenja i metodom množenja, u principu znatno manji od odgovarajućeg srednjeg broja kolizija koje nastaju primjenom metoda randomizacije, Fon Nojmana i sabiranja dijelova ključa.

U metodi zavisnoj od ključa, najprostija funkcija toga tipa se nalazi analizom znakova ključa. Pretpostavimo da se ključ sastoji od znakova $k = c_1, c_2, \ldots, c_n$. Može se izračunati vjerovatnoća pojavljivanja svakog znaka na pozicijama ključa.

Zatim izračunamo entropiju H_j za svaku poziciju j ključa

$$H_j = -\sum_{i=1}^{w} p_{ij} \cdot \log(p_{ij}), \text{za} j = 1, 2, , \ldots, s$$

pri čemu je:

s dužina ključa

p_{ij} vjerovatnoća pojavljivanja znaka c_i na j-toj poziciji ključa

c_i je elemenat alfabeta ključa $\{c_1, c_2, \ldots, c_w\}$.

Eliminišemo iz ključa znakove na pozicijama na kojima je entropija najmanja, jer što je entropija veća to se raspodjela vjerovatnoća približava uniformnoj raspodjeli, do čega nam je i stalo. Na osnovu preostalih znakova formiramo adresu slota u koji smještamo slog.

Praksa pokazuje da analiza znakova ne daje dobre rezultate, jer eliminacija znaka na nekoj poziciji ključa ne garantuje ravnomjernu raspodjeluključeva u tabeli. Bolji rezultati mogu se postići ako se funkcija heširanja konstruiše na osnovu empiričke funkcije $F_K(k)$, koja opisuje raspodjelu skupa ključeva tabele u prostoru svih mogućih ključeva. Ta funkcija je funkcija raspodjele slučajne promjenljive K, koja uzima vrijednosti iz prostora ključeva S, a realizuje ključeve $k_1, k_2, \ldots, k_n$. Pri tome se funkcija raspodjele slučajne promjenljive K definiše kao

$$F_K(k) = P(K \leq k),$$

gdje je $P(y)$ vjerovatnoća dogadjaja y.

Slučajna promjenljiva K uzima vrijednosti iz skupa ključeva heš tabele, a $0 \leq F_K(k) \leq 1$. Treba naći heš funkciju h koja daje uniformnu raspodjelu adresa tako da je $P(h(k)) = d) = \frac{1}{p}$, za $0 \leq d \leq p-1$. Razmotrimo $F_K(K)$. Ako se svi ključevi razlikuju, imamo $P\left(F_K(K) \leq \frac{r}{n}\right)$, za rn. Zbog toga $F_K(K)$ ima ravnomjernu raspodjelu na skupu $\{\frac{1}{n}, \ldots, \frac{n-1}{n}, 1\}$, pa $pF_K(K)$ ima ravnomjernu raspodjelu na skupu $\{\frac{p}{n}, \frac{2p}{n}, \ldots, p\}$. Zato je $\lceil pF_K(K) - 1 \rceil$ približno ravnomjerno rasporedjeno na skupu adresa $\{0, 1, \ldots, p-1\}$. Zbog toga možemo uzeti heš funkciju u obliku $h(k) = \lceil pF_K(K) \rceil - 1$. U specijalnom slučaju ravnomjerne raspodjele ključa, heš funkcija se može uzeti u obliku $h(k) = \left\lceil \frac{p \cdot k}{k_{max}} \right\rceil - 1$.

6.8.3 Rad sa heš tabelama

Heš tabele podržavaju operacije nad heš tabelama: $INSERT$, $SEARCH$ i $DELETE$.

1. Tabele sa direktnim adresiranjem

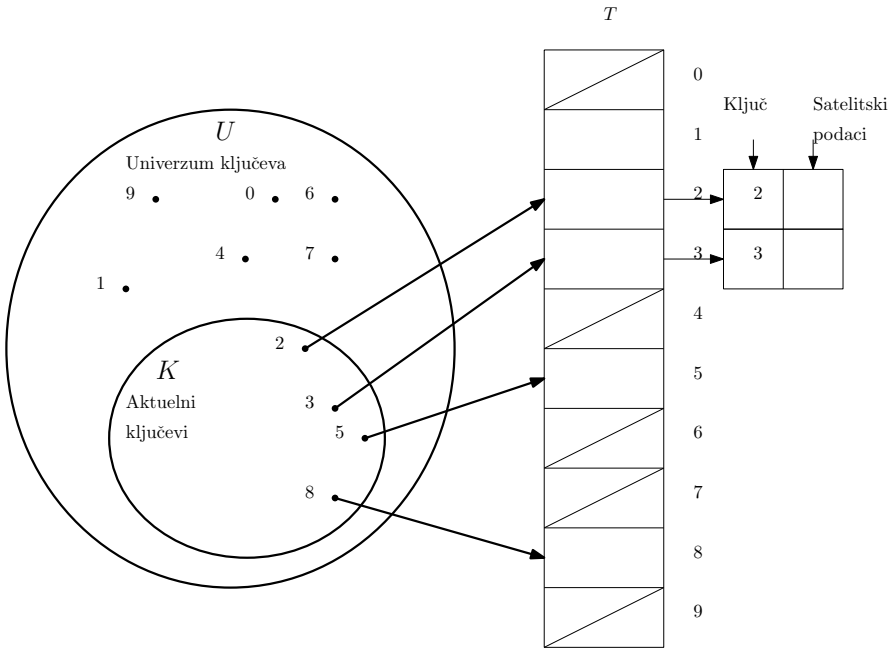
Ključ (Key) je iz univerzalnog skupa ključeva $\in U = \{0, ..., m-1\}$.

Svaka dva elementa podataka imaju različite ključeve.

podatke smještamo u

$$\text{Array(niz) = DirectAddressTable T}\lceil 0 \square m - 1 \rceil,$$

kao što je prikazano na slici 6.17



Slika 6.17: Primjer direktnog adresiranja

Pozicija ili slot odgovara jednom ključu iz univerzum U.

Na slici 6.17 dajemo primjer smještanja ključeva u slotove, pri direktnom adresiranju, pri čemu je $U = [0, 1, ..., 9]$ i $K = [2, 3, 5, 8]$.

Operacije su date programima:

```
DIRECT_ADDRESS_SEARCH (T,k)
    return T[k]
DIRECT_ADDRESS_INSERT(T,x)
    T[key[x]] <- x
DIRECT_ADDRESS_DELETE (T,x)
    T[key[x]] <- NIL
```

2. Heš tabele i kolizije

Direktnim adresiranjem element sa ključem k stavlja se u slot k. Heširanjem se on stavlja u $h(k)$ gdje je h heš funkcija koja računa slot za k. Tako imamo da $h = U\{0, 1, ..., m-1\}$, k hešira slot $h(k)$ i $h(k)$ je heš vrijednost od k.

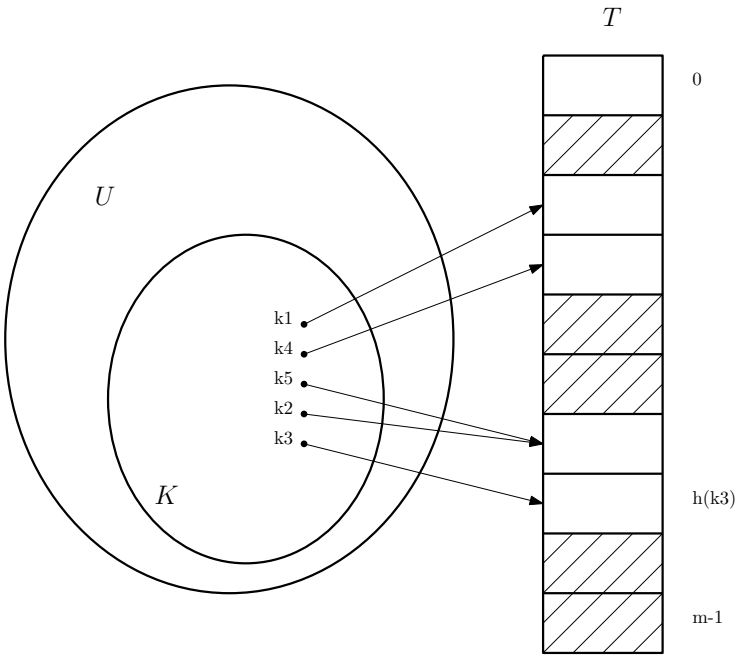
Kada se, kao na slici 6.18, ključevi $k2$ i $k5$ heširaju u isti slot, onda govorimo o **koliziji**.

Zahtijeva se da h mora biti deterministička funkcija i da je $|U| > m$.

Iako dobro dizajnirana random funkcija može minimizirati broj kolizija, mi ipak trebamo neki metod za rješavanje kolizija.

Rješavanje kolizija uvezivanjem

Na slici 6.19 prikazano je rješavanje kolizije uvezivanjem za slučaj $h(k1) = h(k4)$ i $h(k5) = h(k2) = h(k7)$.



Slika 6.18: Heš tabela

Operacije opisujemo sljedećom tabelom:

```
CHAINED_HASH_INSERT (T,x)
    Insert x of the head of list T[h(key[x])]
CHAINED_HASH_SEARCH (T,k)
    Tra\v{z}i jedan element sa klju\v{c}em k u listi T[h[k]]
CHAINED_HASH_DELETE (T,x)
    Delete x iz liste T[h[key[x]]]
```

Analiza heširanja sa ulančavanjem

Neka je dato n elemenata koje treba smjestiti u m slotova. Definišemo faktor opterećenja $\alpha = \frac{n}{m}$. α može da bude manje, veće ili jednako 1.

Worst-case je neprihvatljivo.

Iz jednostavnog računa dobivamo da je

$$\text{Lista + funkcija heširanja} \sim \theta(n).$$

Average-case

Razmotrimo prosto uniformno heširanje.

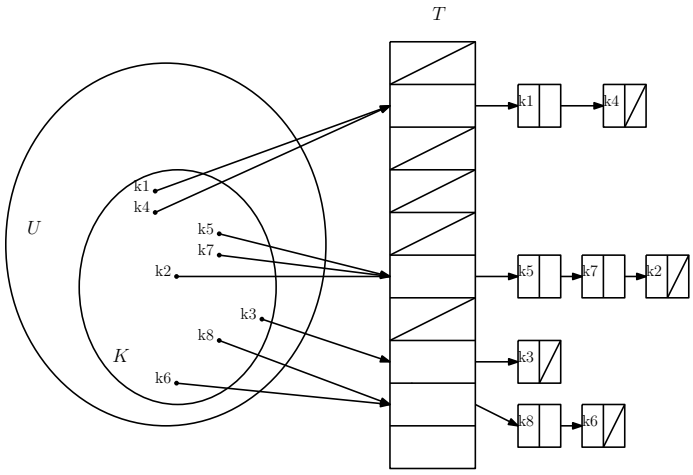
Neka je $length(T[j]) = n_j$, $j = 0, \ldots, m-1$.

Onda je $n = n_0 + ... + n_{m-1}$.

Srednja vrijednost n_j je tada $E[n_j] = \alpha = n/m$.

Neka $h(k)$ možemo izračunati u $O(1)$ vremenu i pristupiti slotu $h(k)$. Odatle slijedi da pretraživanje zavisi linearno od dužine $n_{h(k)}$ liste $T[h(k)]$.

Sada su moguća dva slučaja:



Slika 6.19: Rješavanje kolizije uvezivanjem

1. Nijedan element tabele nema ključ k
2. Pretraživanje uspješno nalazi element sa ključem k

Teorema 6.1. *U heš tabeli, u kojoj su kolizije riješene uvezivanjem, neuspješno pretraživanje zahtijeva $T = \theta(1 + \alpha)$, uz pretpostavku uniformnog heširanja.*

Dokaz
Očekivano vrijeme je dužina liste $T[h(k)]$, a ta ima očekivanu dužinu $E[n_{h(k)}] = \alpha$. Broj ispitivanih elemenata je α, pa slijedi da je $T = \theta(1 + \alpha)$. □

Teorema 6.2. *U heš tabeli, u kojoj se kolizije rješavaju uvezivanjem, uspješno pretraživanje, u srednjem, zahtijeva $O(1 + \alpha)$ vrijeme, uz pretpostavku uniformnog heširanja.*

Dokaz
Označimo $k_i = key[x_i]$, $i = 1, 2, ..., n$

$$X_{ij} = I\{h(k_i) = h(k_j)\} = \frac{1}{m} \Rightarrow E[X_{ij}] = \frac{1}{m}$$

Očekivani broj ispitanih elemenata je
$E[\frac{1}{n}\sum_{i=1}^{n}(1 + \sum_{j=i+1}^{n} X_{ij})]$
$= \frac{1}{n}\sum_{i=1}^{n}(1 + \sum_{j=i+1}^{n} E[X_{ij}])$
$= \frac{1}{n}\sum_{i=1}^{n}(1 + \sum_{j=i+1}^{n} \frac{1}{m}) = 1 + \frac{1}{nm}\sum_{i=1}^{n}(n - i) = 1 + \frac{1}{nm}(\sum_{i=1}^{n} n - \sum_{i=1}^{n} i) = 1 + \frac{1}{nm}(n^2 -$

$\frac{n(n+1)}{2}) = 1 + \frac{n-1}{2m} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} \Longrightarrow T = \Theta(2 + \frac{\alpha}{2} - \frac{\alpha}{2n}) = \Theta(1 + \alpha).$ □

6.8.4 Heš funkcije

U 6.8.1 smo dali pregled heš funkcija. Ovdje ćemo detaljnije razmotriti funkcije heširanja iz dvije klase:

- **Heurističke:** Heširanje dijeljenjem i heširanje množenjem
- **Stohastičke:** Univerzalno heširanje

Napominjemo definiciju uniformnog heširanja: Svaki ključ je podjednako moguće heširati u bilo koji slot od m, nezavisno od toga gdje su heširani drugi ključevi.

Teškoće

1. Ne znamo distribuciju vjerovatnoća sa kojima će ključevi biti izvučeni.
2. Ključevi mogu biti vučeni u zavisnosti jedan od drugog.

Ponekad znamo distribucije: ključevi su k slučajnih brojeva nezavisno i uniformno distribuirani u $0 \leq k < 1 \Rightarrow h(k) = \lfloor km \rfloor$ zadovoljava uslov prostog uniformnog heširanja. Praksa:

1. pt i pts slati u različite slotove,
2. zahtijevaju se ponekad oštriji uslovi: daleke vrijednosti heširati u bliske slotove.

Ključevi kao prirodni brojevi

Uzimamo $U = \mathbb{N} = \{0, 1, 2, \ldots\}$. Ako ključevi nisu prirodni brojevi, onda ih možemo tako interpretirati.

Primjer 6.2. *pt=(112,116), (112*128)+116=14452 (cio broj u osnovi 128)*

□

Metod dijeljenja.

Uzimamo $h(k) = k \mod m$

Primjer 6.3. $m = 12, k = 100, h(k) = 4$
⇒ mora biti m≠ 2^p, jer je $h(k) = p$ najnižih bitova.
Bolje je napraviti heš funkciju zavisnu od svih bitova!
$m = 2^p - 1$ *je loš izbor, jer permutovanje znakova k*
(interpretirano u radix 2^p) ne mijenja slot. Dokazati!
Prost broj daleko od 2^p je dobar izbor.

□

Primjer 6.4. *Uvezivanje za kolizije $n = 2000$ stringova znakova (char ima 8 bitova).*

Ako nam, u slučaju neuspješnog pretraživanja, ne predstavlja problem pretraživati prosječno 3 elementa, onda možemo izabrati $m = 701,\ h(k) = k \mod 701$, gde je k prirodan broj!

Izabrali smo $m = 701$, jer je prost broj, $701 \approx 2000/3$ i daleko je od 2^n!

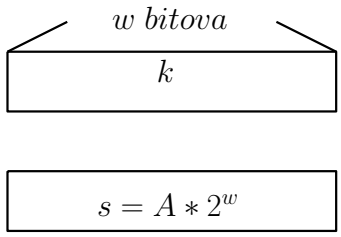
Metod množenja.

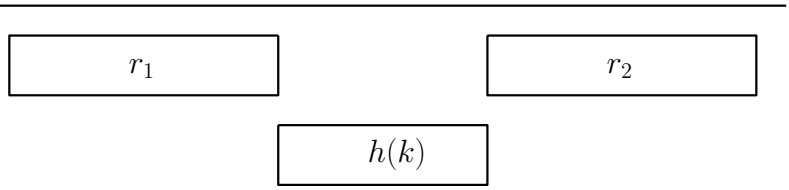
1. k množimo konstantom $A,\ 0 < A < 1$ i uzimamo funkcijski dio od kA
2. množimo sa m i uzimamo floor kao rezultat

$\Rightarrow h(k) = \lfloor m(kA \mod 1) \rfloor,$
sa $kA \mod 1 \equiv kA - \lfloor kA \rfloor$

Tipično, m nije kritičan, pa uzimamo $m = 2^p,\ p \in \mathbb{Z}$, jer je tada implementacija na kompjuteru jednostavnija.

1. Dužina riječi mašine je w bita.
2. k staje u jednu riječ.
3. A je decimalni dio od $\frac{s}{2^w},\ 0 < s < 2^w$.





Slika 6.20: Metod množenja

Rezultat je $r_1 2^w + r_0$.

4. Uzimamo p bitova najveće težine u r_0 (slika 6.20).

D.Knuth sugeriše $A \approx \dfrac{\sqrt{5}-1}{2} = 0.6180339887$, jer se onda ključevi približno uniformo distribuiraju u slotove.

Primjer 6.5. $k = 123456,\ p = 14,\ m = 2^{14} = 16384, w = 32.$

A neka bude decimalni dio $\frac{s}{2^{32}}$ *najbliže* $\dfrac{\sqrt{5}-1}{2}$

$\Rightarrow A = \frac{2654435769}{2^{32}}$

$k \cdot s = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$

i $r_1 = 76300,\ r_0 = 17612864 \Rightarrow h(k) = 67.$

6.8.5 Univerzalno heširanje

Da ne bi svi ključevi bili heširani u isti slot, treba **slučajno** izabrati heš funckiju. Ako je funkcija izabrana nezavisno od vrijednosti ključeva, to je onda **univerzalno heširanja**.

Izbor heš funkcije se vrši iz unaprijed pažljivo dizajniranog skupa funkcija.

$\mathscr{H}$ - konačna kolekcija heš funkcija.

$h : U \to \{0, 1, \ldots, m-1\}$

U je univerzalna kolekcija akko $\forall k, l \in U$, broj $h \in \mathscr{H}$ za koje je $h(k) = h(l)$ najviše je $\frac{|\mathscr{H}|}{m}$, tj. šanse da bude kolizija $h(k)$ i $h(l)$ su $\leq \frac{1}{m}$. Pri tome su $h(k)$ i $h(l)$ nezavisno i slučajno izabrani iz $\{0, 1, \ldots, m-1\}$.

Teorema 6.3. *Neka je h izabrana iz univerzalne kolekcije $\mathscr{H}$ heš funkcija. h se koristi za heširanje n ključeva u tabelu T dužine m, sa uvezivanjem za razrješavanje kolizija.*

Ako k nije u tabeli, tada je očekivana dužina $E[n_{h(k)}]$ liste u koju se hešira k, najviše α.

Ako je k u T, tada je očekivana dužina $E[n_{h(k)}]$ liste sa ključem k najviše $1 + \alpha$.

Dokaz

Za $\forall k, l$ definišemo $X_{kl} = I\{h(k) = h(l)\}$

k, l kolidiraju sa vjerovatnoćom $\leq \frac{1}{m}$.

$\Rightarrow Pr\{h(k) = h(l)\} \leq \frac{1}{m} \Rightarrow E[X_{kl}] \leq \frac{1}{m}$.

Definišemo za $\forall k$

$$Y_k = \sum_{\substack{l \in T \\ l \neq k}} X_{kl}$$

$$\Rightarrow E[Y_k] = E[\sum_{\substack{l \in T \\ l \neq k}} X_{kl}] = \sum_{\substack{l \in T \\ l \neq k}} E[X_{kl}] \leq \sum_{\substack{l \in T \\ l \neq k}} \frac{1}{m}$$

- $k \notin T \Rightarrow n_{h(k)} = Y_k$ i

  $| \{k \mid l \in T \& l \neq k\} | = n$

  $\Rightarrow E[n_{h(k)}] = E[Y_k] \leq \frac{n}{m} = \alpha$

- $k \in T \& k \in T[h(k)] \&$ k nije u računu Y_k

  $\Rightarrow n_{h(k)} = Y_k + 1 \wedge | \{l \mid l \in T \& l \neq k\} | = n - 1$

$$\Rightarrow E[n_{h(k)}] = E[Y_k] + 1 \leq \frac{n-1}{m+1} = 1 + \alpha - \frac{1}{m} < 1 + \alpha.$$

□

Korolar 6.1. *Korištenje univerzalnog heširanja i rješavanja kolizija uvezivanjem u tabeli sa m slotova, zahtijeva $T = \Theta(n)$ za bilo koju sekvencu od n $INSERT$, $SEARCH$, $DELETE$ operacija od kojih su njih $\Theta(m)$ $INSERT$.*

Dokaz
$| Ins |= O(m) \Rightarrow n = O(m) \Rightarrow \alpha = O(1)$

$INSERT$, $DELETE$ trebaju $O(1)$ vrijeme, a takodje i $SEARCH$. Zbog linearnosti E za čitavu sekvencu (niz) treba $T = O(n)$ koraka. □

6.8.6 Otvoreno adresiranje

Svi elementi su u samoj heš tabeli, tj. svaki unos sadrži ili element ili NIL. Kada tražimo element, pretražujemo samo tabelu dok ne nadjemo element ili dok ne bude jasno da tabela ne sadrži element. Tabela može biti popunjena do kraja bez mogucnosti daljeg unosa. "Load faktor" (faktor popunjenosti) nikada ne može biti veći od 1.
Umjesto pointera računamo (slijedimo) nizove slotova koje treba ispitati. Memorija koju bi koristili pointeri daje veći broj slotova, potencijalno vodeći ka smanjenju kolizija i bržem pretraživanju.
INSERT $h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$
Zahtijevamo: probni niz $h(k,0), h(k,1), \ldots, h(k,m-1)$ je permutacija $\{0, 1, \ldots, m-1\}$. □

```
HASH_INSERT(T,k)
        i <- 0
        repeat j <- h(k,i)
                    if T[j]=NIL
                          then T[j]<-k
                                  return j
                          else i<-i+1
        until i=m
        error "hash table overflow"
HASH_SEARCH (T,k)
        i<-0
        repeat j<- h(k,i)
                    if T[j]=k
                          then return j
                          i<-i+1
        until T[j]=NIL or i=m
        return NIL
```

$DELETE$ će imati poteškoća ako stavimo NIL u izbrisani slot i. Nećemo moći izbrisati element za koji je u postavljanju i nadjen kao popunjen. Izlaz iz problema: umjesto NIL stavimo $DELETE$.

6.8.7 Uniformno heširanje

Za svaki element probni niz ima jednaku vjerovatnoću da bude jedan od m! permutacija. Tačno uniformno heširanje je teško za implementaciju.

Umjesto pravog UH koristimo: dvostruko heširanje, linearnu probu i kvadratnu probu.

Linearna proba.

$h' : U \to \{0, 1, \ldots, m-1\}$ neka je pomoćna heš funkcija. Linearna proba koristi $h(k,i) = (h'(k) + i) \mod m$.

Samo je m različitih probnih nizova odredjeno sa $h(k, 0)$.

Jednostavna je implementacija, ali imamo problem *primarno klasterovanje*! To nam povećava average time.

KLASTEROVANJE: iza i popunjenih, vjerovatnoća popunjavanja slota $i+1$ da bude sledeći popunjen je $\frac{i+1}{m} - klasterovanje.$

Kvadratna proba. $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$

Dvostruko heširanje.

Kod kvadratne probe se pojavljuje $h(k_1, 0) = h(k_2, 0) \Rightarrow h(k_1, i) = h(k_2, i)$, tzv. sekundarno klasterovanje, tj. k_1 i k_2 imaju iste nizove.

Zbog toga uvodimo dvostruko heširanje $h(k,i) = (h_1(k) + h_2(k))$, pri čemu $h_2(k)$ mora biti relativno prost sa dužinom table.

Primjer 6.6. $h_1(k) = k \mod$ *13*
$h_2(k) = 1 + (k \mod$ *11)*
Za vježbu ubaciti 79,69,98,72,14,50.

6.8.8 Komentari

Postoji više pristupa problemima struktura podataka. Tri najčešće upotrebljavana su *identifikacija domena, algebarska specifikacija i konstrukcija domena.* U pionirskim radovima C.A.R. Hoare je istaknuto da posao programera treba biti podijeljen u dvije faze:

1. specifikacija i implementacija struktura podataka;
2. dizajn i verifikacija abstraktnog programa u strukturama podataka.

Uspješna realizacija ovih principa je postignuta u savremenim objektno orijentisanim jezicima.

Osim razmotrenih struktura podataka imamo i druge koje se pojavljuju u programerskoj praksi, na primjer u kompjuterskoj grafici ili u procesiranju podataka za upravljanje kao što su bankarstvo ili aplikacije u realnom vremenu. U svakom slučaju se može izgraditi teorija odgovarajuće strukture. Teorija treba zadovoljiti zahtjeve identifikacije domena, verifikacije korektnosti tvrdnji o programima, testiranje saglasnosti implementacije i specifikacije i što je najvažnije treba dati odgovarajuće razumijevanje problema strukture podataka.

Glava 7

Linearno programiranje

7.1 Uvod

Linearno programiranje se može definisati kao dio matematike koji se bavi pronalaženjem minimalne odnosno maksimalne vrijednosti linearne funkcije uz ograničenja koja su data u obliku linearnih (ne)jednacina.

Problem linearnog programiranja u opštem obliku glasi:

Optimizovati (maksimizovati ili minimizovati) linearnu funkciju

$$F(x_1, x_2, ..., x_n) = \sum_{i=1}^{n} c_i x_i$$

uz sljedećih m uslova:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n &\leq b_2 \\ ... & \\ a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n &\leq b_m \end{aligned}$$

Ovdje će biti izložen samo problem nalaženja maksimuma linearne funkcije uz zadate uslove. Problem pronalaženja minimuma je sličan.

Ako označimo sa $x = [x_1\ x_2\ ...\ x_n]^T$, sa $c = [x_1,\ x_2\ ...\ x_n]$ i $b = [b_1\ b_2\ ...\ b_m]^T$ i sa A matricu koeficijenata a_{ij}, dobijamo kanonski oblik problema linearnog programiranja:

Pronaći maksimum linearne funkcije

$$F = cx$$

uz ograničenja

$$Ax \leq b$$

U praksi se najčešce pojavljuje i dodatni uslov, da sve promjenljive $x_1, x_2, ..., x_n$ moraju biti pozitivne.

Linerano programiranje predstavlja jednu od osnovnih tehnika operacionih istraživanja i ima veliku primjenu u mnogim oblastima: proizvodnja, transport, ekonomija, marketing i sl.

7.2 Graficka metoda za rješavanje problema linearnog programiranja

Jedna od metoda za rješavanje prostijih problema linearnog programiranja je graficka metoda (optimizacije funkcija manjeg broja promjenljivih). Ideja je sljedeća: za svaki od uslova nacrta se poluravan u kojoj važi taj uslov. Presjek svih poluravni daje područje u kojem važe svi uslovi. To područje se naziva **područje izvodljivosti**. To je, ustvari, jedan poligon i rješenje problema je jedno od njegovih tjemena. Ako pretpostavimo da F ima konstantnu vrijednost, grafik jednačine $c_1x_1 + c_2x_2 = C$ je prava. Mijenjajući vrijednosti promjenljive C, dobijamo paralelne prave. Ako maksimum funkcije F iznosi C_{max}, prava $c_1x_1 + c_2x_2 = C_{max}$ prolazi kroz jedno od tjemena i to je upravo tačka koja je rješenje problema.

Primjer 7.1. *Pekara svaki dan dobija 120kg bijelog brašna, 45kg crnog brašna, 2 kg kvasca i 3 kg soli. Vekna bijelog hljeba iznosi 1 KM a polubijelog 1,20 KM. Za bijeli hljeb je potrebno 800g bijelog brašna, 10g kvasca i 10g soli, a za polubijeli 600g bijelog brašna, 300g crnog brašna 10g kvasca i 20g soli. Koliko bijelih a koliko polubijelih vekni pekara treba da napravi tako da zarada bude maksimalna?*

Rješenje

Označimo sa x_1 i x_2 traženi broj vekni bijelog odnosno crnog hljeba. Tada funkcija čiji maksimim tražimo glasi

$$F(x_1, x_2) = x_1 + 1,2x_2$$

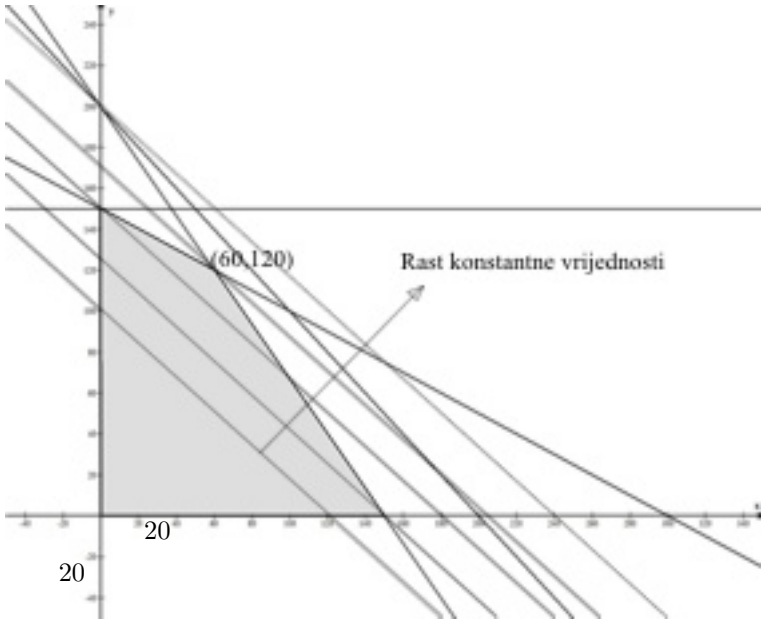
a uslovi su sljedeći

$$\begin{aligned} 0,8x_1 + 0,6x_2 &\leq 120 \\ 0,3x_2 &\leq 45 \\ 0,01x_1 + 0,01x_2 &\leq 2 \\ 0,01x_1 + 0,02x_2 &\leq 3 \end{aligned}$$

Jasno je mora da važi $x_1 \geq 0$ i $x_2 \geq 0$.

Na slici 1. osjenčeni dio predstavlja područje gdje su sva četiri uslova ispunjena. Paralelne linije su grafici funkcija $x_1+1,2x_2 = C$ za različite vrijednosti promjenljive C. Najveća vrijednost promjenljive C čiji grafik ima zajedničkih tačaka sa osjenčenom površi je rješenje problema. Maksimalna vrijednost je $F_{max} = 204$ za $x_1 = 60$ i $x_2 = 120$.

Grafička metoda, je relativno prosta, ali u praksni nije upotrebljiva jer je komplikovana za funkcije koje imaju više od dvije promjenljive. U tom slučaju se dobije više tjemena i nije lako provjeriti u kojem funkcija ima ekstremum.



Slika 7.1: Graficko rješenje primjera 1.

7.3 Pivot operacija

Neka je dat sljedeci sistem jednacina

$$\begin{aligned} 2x_1 + 3x_2 + x_3 &= s_1 \\ x_1 - 2x_2 + 3x_3 &= s_2 \\ x_2 - x_3 &= s_3 \end{aligned} \tag{7.1}$$

Promjenljive s_1, s_2 i s_3 nazivaju se zavisne a x_1, x_2 i x_3 nezavisne promjenljive. Pretpostavimo da želimo da neku nezavisnu promjenljivu transformišemo u zavisnu. Npr. želimo da promjenljive s_1, s_2 i x_2 budu zavisne a x_1, s_3 i x_3 budu nezavisne. To je moguce postici tako što s_1, s_2 i x_2 izrazimo preko x_1, s_3 i x_3:
Iz treće jednakosti izrazimo x_2 preko x_3 i s_3

$$x_2 = s_3 + x_3$$

i dobijenu vrijednost uvrstimo u prve dvije jednakosti

$$\begin{aligned} 2x_1 + 3(s_3 + x_3) + x_3 &= s_1 \\ x_1 - 2(s_3 + x_3) + 3x_3 &= s_2 \end{aligned}$$

Nakon sredivanja dobijamo

$$\begin{aligned} 2x_1 + 3s_3 + 4x_3 &= s_1 \\ x_1 - 2s_3 + x_3 &= s_2 \\ s_3 + x_3 &= x_2 \end{aligned}$$

i to je upravo ono što smo željeli: promjenljive s_1, s_2 i x_2 su zavisne a x_1, s_3 i x_3 nezavisne.

Ova operacija se naziva **pivotiranje**, a element a_{ij} oko kojeg pivotiramo naziva se **pivot**.

U opštem slučaju, imamo n linearnih funkcija sa m promjenljivih

$$Ax = s$$

gdje su $x = [x_1\ x_2\ ...\ x_n]^T$ i $s = [s_1\ s_2\ ...\ s_n]^T$ i

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \ddots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Nakon pivotiranja promjenljivih x_i i s_j dobićemo

$$A'x' = s'$$

gdje su $x' = [x_1\ x_2\ ...\ s_j\ ...\ x_n]^T$ i $s = [s_1\ s_2\ ...\ x_i\ ...\ s_n]^T$ i

$$\begin{bmatrix} a'_{11} & a'_{12} & \cdots & a'_{1n} \\ a'_{21} & a'_{22} & \cdots & a'_{2n} \\ \cdots & \cdots & \ddots & \cdots \\ a'_{m1} & a'_{m2} & \cdots & a'_{mn} \end{bmatrix}$$

Relacije izmedju a_{ij} i a'_{ij} su sljedeće:

$$\begin{aligned} a'_{ij} &= \frac{1}{a_{ij}} \\ a'_{hj} &= \frac{a_{hj}}{a_{ij}}\ za\ h \neq i \\ a'_{ik} &= \frac{-a_{ik}}{a_{ij}}\ za\ k \neq j \\ a'_{hk} &= a_{hk} - \frac{a_{ik}a_{hj}}{a_{ij}}\ za\ h \neq i\ i\ k \neq j \end{aligned}$$

Šematski,

$$\begin{bmatrix} a & b \\ c & \boxed{p} \end{bmatrix} \rightarrow \begin{bmatrix} a - \frac{bc}{p} & \frac{b}{p} \\ -\frac{c}{p} & \frac{1}{p} \end{bmatrix}$$

Primjenjujući opisani metod, pivotiranje na sistemu 7.1 se može riješiti na sljedeći nacin

	s_1	s_2	s_3
x_1	2	1	0
x_2	3	-2	$\boxed{1}$
x_3	1	3	-1

$\rightarrow$

	s_1	s_2	x_2
x_1	2	1	0
s_3	3	-2	1
x_3	4	1	1

Ako nastavimo s pivotiranjem i pivotiramo sve elemente, zatim poredamo kolone i redove u pravi poredak, dobićemo inverznu matricu početne matrice.

	s_1	s_2	x_2
x_1	2	$\boxed{1}$	0
s_3	3	-2	1
x_3	4	1	1

$\rightarrow$

	s_1	x_1	x_2
s_2	2	1	0
s_3	7	2	1
x_3	$\boxed{2}$	-1	1

$\rightarrow$

	x_3	x_1	x_2
s_2	-1	2	-1
s_3	$-\frac{7}{2}$	$\frac{11}{2}$	$-\frac{5}{2}$
s_1	$\frac{1}{2}$	$-\frac{1}{2}$	$\frac{1}{2}$

Odnosno, inverzna matrica početne matrice je

$$A^{-1} = \begin{bmatrix} -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ 2 & -1 & -1 \\ \frac{11}{2} & -\frac{5}{2} & -\frac{7}{2} \end{bmatrix}$$

Ova osobina je pogodna za provjeru pivotiranja.

7.4 Simpleks metoda

Kao što je već opisano, grafička metoda nije pogodna za optimizaciju funkcija koje imaju više od dvije promenljive. Pored toga nije pogodna za implementaciju na računaru.

Simpleks metoda efikasno pretražuje tjemena područja izvodljivosti i pronalazi tjeme u kojem je vrijednost funkcije optimalna. Neka je dat problem linearnog programiranja u kanonskom obliku:

Pronaći maksimum linearne funkcije

$$F = cx$$

uz ograničenja

$$Ax \leq b$$

gdje su $x = [x_1\ x_2\ ...\ x_n]^T$, $c = [c_1\ c_2\ ...\ c_n]$, $b = [b_1\ b_2\ ...\ b_m]^T$ i A matrica koeficijenata a_{ij}.

Simpleks matrica je matrica sljedećeg oblika

x	
A	b
$-c$	0

Ova matrica ima $m+1$ redova i $n+1$ kolonu. Prvih m redova su koeficijenti matrice A a u $m+1$-om redu su koeficijenti c_i kojima se promijeni znak. Prvih n kolona popunjavaju koeficijenti matrice A a u $n+1$-oj koloni se nalaze koeficijenti b_i. "Nulti" red je pomoćni red i on sadrži promjenljive x_i

x_1	x_2	$\dots$	x_n	
a_{11}	a_{12}	$\dots$	a_{1n}	b_1
a_{21}	a_{22}	$\dots$	a_{2n}	b_2
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
a_{m1}	a_{m2}	$\dots$	a_{mn}	b_m
$-c_1$	$-c_2$	$\dots$	$-c_n$	0

Ukoliko vršimo pivotiranja sve dok ne dobijemo nenegativne elemente u zadnjem redu (gdje se nalaze koeficijenti c_i) i zadnjoj koloni (gdje se nalaze koeficijenti b_j) rezultat optimizacije nalaziće se u donem desnom uglu.

Simpleks metoda nam govori kako da biramo elemente za pivotiranje.

Slucaj 1. *Svi elementi b_i su nenegativni*

Izaberemo bilo koju kolonu u kojoj je posljednji član negativan. Neka je to kolona sa indeksom j_0. Zatim izaberemo red i_0 takav da je $a_{i_0,j_0} > 0$ i da je vrijednost $b_{i_0}/a_{i_0,j_0}$ minimalna. Nakon izbora elementa a_{i_0,j_0} vršimo pivotiranje oko njega.

Proces ponavljamo sve dok ne dodjemo do slučaja da su svi elementi posljednjeg reda nenegativni (u tom slučaju je problem riješen) ili dok ne ne dodjemo do slučaja da su svi a_{i,j_0} negativni (u tom slučaju problem je neizvodljiv, odnosno područje izvedivosti je neograničeno).

Primjer 7.2. *Maksimizovati funkciju $x_1 + x_2 + 2x_3$ uz uslove*

$$\begin{aligned} x_2 + 2x_3 &\le 3 \\ -x_1 + 3x_3 &\le 2 \\ 2x_1 + x_2 + x_3 &\le 1 \end{aligned}$$

Rješenje

	x_1	x_2	x_3	
	0	1	2	3
	−1	0	3	2
	2	1	1	1
	−1	−1	−2	0

→

	x_1		x_3	
	−2	−1	1	2
	−1	0	3	2
x_2	2	1	1	1
	−1	1	−1	1

→

	x_1			
				$\frac{4}{3}$
x_3				$\frac{2}{3}$
x_2				$\frac{1}{3}$
	$\frac{2}{3}$	1	$\frac{1}{3}$	$\frac{5}{3}$

Maksimalna vrijednost funkcije je $\frac{5}{3}$, a optimalni vektor je $x_1 = 0$, $x_2 = \frac{1}{3}$ i $x_3 = \frac{2}{3}$.

Slučaj 2. *Neki element b_i je negativan*

Izaberimo prvi negativni element b_k. Pronadjimo bilo koji negativni element a_{k,j_0} u k-tom redu. Uporedimo $\frac{b_k}{a_{k,j_0}}$ i $\frac{b_i}{a_{i,j_0}}$ za svako $b_i \geq 0$ i $a_{i,j_0} > 0$ i izaberimo i_0 takvo da je ovaj količnik najmanji. Pivotirajmo a_{i_0}, j_0.

Primjer 7.3. *Maksimizovati funkciju $x_1 + x_2 + 5x_3$ uz uslove*

$$\begin{aligned} x_2 + 2x_3 &\leq 3 \\ -x_1 - 3x_3 &\leq -2 \\ 2x_1 + x_2 + 7x_3 &\leq 5 \end{aligned}$$

Rješenje

	x_1	x_2	x_3	
	0	1	2	3
	$\boxed{-1}$	0	-3	-2
	2	1	7	5
	-1	-1	-5	0

$\rightarrow$

		x_2	x_3	
	0	1	2	3
x_2	-1	0	3	2
	2	1	1	1
	-1	-1	-2	2

Problem je sveden na slučaj 1. koji znamo riješiti. Dalja pivotiranja dovode do rješenja

			x_1	
				$\frac{4}{3}$
x_3				$\frac{2}{3}$
x_2				1
	$\frac{2}{3}$	1	$\frac{1}{3}$	$\frac{11}{3}$

Maksimalna vrijednost funkcije je $\frac{11}{3}$ za vektor $x_1 = 0$, $x_2 = 1$ i $x_3 = \frac{2}{3}$

Do sada su obradjivani samo slučajevi traženja maksimuma, ukoliko se traži minimalna vrijednost funkcije $F = cx$ sa ograničenjima $Ax \geq b$ simpleks matrica će imati oblik

x	A	c
	$-b$	0

a metoda je identična kao kod pronalaženja maksimuma.

7.5 Implementacija

Ulaz programa je simpleks matrica dimenzija $(m+1) \times (n+1)$ a izlaz je optimalno rješenje funkcije. U kodu su m i n pisani velikim slovima, radi preglednosti.

Algoritam koji rješava problem linearnog programiranja je opisan u prethodnim odjeljcima. Pivot se pronalazi u linearnom vremenu, a pivotiranje svaki put vrši mn izračunavanja.

Pretpostavlja se da je problem izvodljiv i da je simpleks matrica ispravno zadana.

Procedura koja vrši pivotiranje oko elementa $a_{p,q}$

```
void pivot(int p, int q)
{
    int i, j;

    for(i = 0; i < M+1; i++)
    {
        if(i == p)
            continue;
        for(j = 0; j < N+1; j++)
        {
            if(j == q)
                continue;
            a[i][j] -= a[p][j]*a[i][q] / a[p][q];
        }
    }

    for(i = 0; i < M+1; i++)
        if(i != p)
            a[i][q] /= -a[p][q];

    for(j = 0; j < N+1; j++)
        if(j != q)
            a[p][j] /= a[p][q];

    a[p][q] = 1 / a[p][q];
}
```

Procedura koja izračunava optimalnu vrijednost funkcije simpleks metodom

```
double simplex()
{
    int i, j, k;
    int simplex_case;

    while(1)
    {
        simplex_case = 1;
        for(i = 0; i < M; i++)
            if(a[i][N] < 0)
                simplex_case = 2;

        if(simplex_case == 1)
        {
            for(j = 0; j < N && a[M][j] >= 0; j++);
            for(i = 0; i < M && a[i][j] <= 0; i++);
```

```
            if(i >= M || j >= N)
                break;

            for(k = i; k < M; k++)
                if(a[k][j] > 0)
                    if(a[k][N] / a[k][j] < a[i][N] / a[i][j])
                        i = k;
        }
        else
        {
            for(i = 0; i < M && a[i][N] >= 0; i++);
            for(j = 0; j < N && a[i][j] >= 0; j++);

            for(k = 0; k < M; k++)
                if(a[k][j] > 0 && a[k][N] >= 0)
                    if(a[k][N] / a[k][j] < a[i][N] / a[i][j])
                        i = k;
        }
        pivot(i,j);
    }
    return a[M][N];
}
```

7.6 Konvergencija i vremenska složenost LP problema

Simpleks metoda se zasniva na principu da se optimalna vrijednost nekog linearnog programa, ako postoji, uvijek dobiva iz baznih rješenja. Ako pretpostavimo da su sva dostižna bazna rješenja nedegenerisana, onda simpleks metoda daje optimalno rješenje nakon konačnog broja iteracija, pošto je broj mogućih baza konačan i nijedna se ne ponavlja dva puta. Za linearan program kažemo da je nedegenerisan ako je $B^{-1}\mathbf{b} > 0$. Ovo je ekvivalentno sa tvrdnjom da $\mathbf{b}$ ne može biti predstavljeno kao linearna kombinacija nekog skupa kolona matrice A koji ima kardinalnost manju ili jednaku $m - 1$. Ako je LP problem nedegenerisan, onda svako bazno dostižno rješenje ima jedinstvenu pridruženu bazu. U slučaju degenerisanog rješenja može postojati niz iteracija koje generišu niz baza $B_1, B_2, \ldots, B_p$, od kojih svaka odgovara istom baznom dostižnom rješenju i istoj vrijednosti funkcije. Može se takodje desiti da je $B_i = B_p$, pa simpleks metoda može da udje u beskonačnu petlju. Beskonačna petlja nije ozbiljan problem, jer se gotovo i ne pojavljuje u praktičnnim implementacijama. LP programi uglavnom nemaju neki kod koji rješava problem degenerisanja. Ipak, pojava beskonačne petlje je ozbiljan problem u nekim cjelobrojnim linearnim problemima koji se rješavaju proširenjem simpleks metode, pa u takvim slučajevima mora biti spriječena programerski. Dakle, polazeći od dostižne baze, nakon konačnog broja koraka, simpleks metoda daje optimalno rješenje, ili daje dostižnu bazu koja omogućava da odredimo da je LP problem neograničen.

Pojavljuje se važno praktično pitanje o broju iteracija potrebnih za rješenje LP problema sa n varijabli i m ograničenja. Za takav problem postoji najviše

$$\left(\begin{array}{c} n \\ m \end{array} \right) = \frac{n!}{m!(n-m)!}.$$

iteracija. Pokazuje se da je ovo veoma slaba ocjena, jer simpleks metoda ne ispituje sva bazna rješenja, kojih je mnogo čak i u problemima osrednje veličine. Praktično se ispituje samo mali broj baznih, dostižnnih rješenja. Iz prakse proizilazi da se broj iteracija kreće izmedju m i $3m$. Postoje i tvrdnje da je za slučajno izabran problem, sa fiksiranim m, broj iteracija proporcionalan sa n. Prihvatljiva ocjena broja iteracija je $2(n+m)$.

Pitanje o maksimalnom mogućem broju dostižnih rješenja je druga stvar. Algoritam se u pivot koraku kreće od jedne ekstremne tačke skupa dostižnih rješenja do neke susjedne ekstremne tačke. Vrijednost funkcije monotono raste kako se put povećava. Ovaj put baznih dostižnih rješenja naziva se $izotoni\ put$ u odnosu na objektivnu funkciju $z = \mathbf{c}^T\mathbf{x}$. Dužina puta l se definiše kao broj tačaka na putu, isključujući polaznu tačku. Klee i Minty [1972] su riješili sljedeći problem. Neka je dat LP problem $min\{\mathbf{c}^T : A\mathbf{x} \geq \mathbf{b}, \mathbf{x} \geq 0\}$, gdje je A matrica reda $m \times n$. Kolika je maksimalna dužina nekog izotonog puta? Selektovanjem odgovarajućih baznih varijabli u iterativnim koracima, može se postići da simpleks metoda slijedi takav put. Pomenuti autori su konstruisali primjer koji pokazuje da maksimalna dužina puta može biti veličine čak $2^k - 1$, gdje je $k = \frac{m}{2} = n$. Broj nije ograničen polinomom po m i n, što znači da je moguće naći varijantu simpleks metode koja slijedi neki izotoni put i taj put nije polinomijalno ograničen. Naša varijanta simpleks metode rješava problem Klee-Minty - ja u polinomijalnom vremenu. Khachian [1980] je prezentovao LP problem sa n redova i $2n$ kolona, koji za rješenje problema treba 2^{n-1} iteracija.

Sa teorijskog aspekta, simleks metoda ima nedostatak da njena vremenska složenost nije polinomijalno ograničena. Umjesto toga, granica je eksponencijalna, što karakteriše algoritme koji se mogu primijeniti samo na relativno male probleme. Ostaje interesantno pitanje: zbog čega simpleks metoda pokazuje dobre performanse pri rješavanju praktičnih problema proizvoljne dimenzije?

Khachian [1979, 1980] je našao polinomijalan algoritam za rješavanje LP problema. Njegov algorita nije bio praktičan. Karmarkarov algoritam, koji je uslijedio, takodje je polinomijalan, ali ima i praktičnu vrijednost.

7.7 Problemi ekvivalentni linearnom programiranju

Korisno je identifikovati klasu problema koji su ekvivalentni LP problemu u smislu da su jednako teški za rješavanje. Preciznije, neki problem je ekvivalentan LP problemu ako se u polinomijalnom vremenu može svesti na LP problem. Ovaj pojam ekvivalentnosti dozvoljava da se bilo koji LP-ekvivalentan problem posmatra kao LP problem i da se u njegovom rješavanju primijeni simpleks algoritam. Može se pokazati da su mnogi matematički problemmi LP-ekvivalentni. Neki

interesantni slučajevi uključuju i sljedeće probleme.

Linearne nejednačine
Neka je data cjelobrojna matrica A, tipa $m \times n$ i cjelobrojni m-vektor $\mathbf{b}$. Treba ispitati da li postoji racionalan vektor $\mathbf{x}$ takav da je $A\mathbf{x} \leq \mathbf{b}$. Geometrijska formulacija problema linearnih nejednačina traži da se za date zatvorene poluravni $H_i = \{x : \mathbf{a}_i^T\mathbf{x} \leq b_i\}, i = 1, 2, \ldots, n$, odredi je li neprazan presjek $H_1 \cap H_2 \cap \ldots \cap H_n$. Napominjemo da se u problemu linearnih nejednačina traži konstatovanje postojanja rješenja, dok se nalaženje samog rješenja ne traži.

Zadatak 7.1. *Dokazati da su problem LP i problem linearnih nejednačina ekvivalentni.*

Relevantnost Skup ograničenja je $redudantan$ ako postoji ograničenje koje je zadovoljeno čim su ostala ograničenja zadovoljena. Korisno je znati je li neki skup ograničenja u LP problemima redudantan. Ovo nas dovodi do pojma $relevantnosti$. Neka je zadat skup ograničenja $\{\mathbf{a}_i^T\mathbf{x} \leq b_i\}$, $i = 1, 2, , \ldots, m$. Odrediti da li zadovoljenje zadnjih $m-1$ ograničenja povlači zadovoljenje svih m ograničenja.

Primijetimo da je sljedeći problem sličan, ali je mnogo teži i nije LP-ekvivalentan: Neka je dat skup linearnih ograničenja $\{\mathbf{a}_i^T\mathbf{x} \leq b_i\}$, $i = 1, 2, , \ldots, m$ i cijeli broj k. Odrediti neki skup od k ograničenja tako da njegova zadovoljivost povlači zadovoljivost svih ograničenja.

Ekstremne tačke Neka je dat skup tačaka u Euklidovom prostoru E^n. *Konveksni omotač* je najmanji konveksan skup koji sadrži ove tačke.

Tačka Q je *ekstremna tačka* u odnosu na tačke $P_1, P_2, \ldots, P_n$ ako i samo ako je Q ekstremna za konveksni omotač tačaka $P_1, P_2, \ldots, P_n$. Problem ekstremne tačke traži da se za dati skup tačaka $P_0, P_1, \ldots, P_n$ u E^n, odredi je li P_0 ekstremna u odnosu na $P_1, P_2, \ldots, P_n$.

Pošto je P_0 unutrašnja za konveksan omotač tačaka $P_1, P_2, \ldots, P_n$ ako i samo ako

$$\sum_{i=1}^{m} x_i P_i = P_0, x \geq 0, \text{i } \sum_{i=1}^{n} x_i = 1$$

to ovaj problem može biti riješen simpleks metodom.

Razdvajanje skupova tačaka Dva skupa tačaka su *razdvojivi* ako i samo ako postoji hiper-ravan H, tako da sve tačke jednog skupa leže sa jedne strane H, a sve tačke drugog skupa sa druge strane$H$. Problem separacije (razdvajanja) glasi: Dati su skupovi tačaka $P_1, P_2, \ldots, P_n$ i $Q_1, Q_2, \ldots, Q_n$. Odrediti da li su ova dva skupa razdvojivi.

Glava 8

Brza Furijeova transformacija

Furijeova transformaciija se prirodno pojavljuje u mnogim inžinjerskim i naučnim oblastima i zbog toga je potrebno potražiti algoritam koji računa Furijeovu transformaciju. Pokazuje se da Furiijeova transformacija omogućuje dizajn drugih efektivnih algoritama, npr. algoritma za efikasno množenje dva polinoma. Na vektore koeficijenata se prvo primijeni linearna transformacija, zatim se na slikama koeficijenata primijeni operacija prostija od konvolucija i na kraju inverzna transformacija rezultata daje traženi rezultat. Linearna transformacija pogodna za ovu situaciju je diskretna Furijeova transformacija.

8.1 Kompleksni korjeni jedinice

Definišemo n-ti korjen jedinice, gdje je $n = 1, 2, ...$ kompleksni broj z takav da je

$$z^n = 1$$

Za n-ti korjen iz jedinice kažemo da je prost ako je

$$z^k \neq 1 \ (k = 1, 2, ..., n-1)$$

Prost korjen jedinice je oblika

$$e^{\frac{2\pi i}{n}}$$

jer je

$$(e^{\frac{2\pi i}{n}})^k = e^{\frac{2\pi ik}{n}} \neq 1 \ (k = 1, 2, ..., n-1)$$
$$(e^{\frac{2\pi i}{n}})^n = e^{2\pi i} = 1$$

Da bismo u praksi koristili ovu formulu, koristimo jednakost

$$e^{iu} = \cos u + i \sin u$$

8.2 Diskretna Furijeova transformacija

Neka je dat polinom

$$P(x) = \sum_{j=0}^{n-1} a_j x^j$$

stepena n. Pretpostavimo da želimo da izračunamo vrijednosti datog polinoma P koji je dat svojim vektorom koeficijenata $a = (a_0, a_1, ..., a_{n-1})$ u tačkama $z^0, z^1, ..., z^{n-1}$, pri čemu je z n-ti korjen iz jedinice.

Sa y_k $(k = 0, 1, ..., n-1)$ označimo

$$y_k = P(z^k) = \sum_{j=0}^{n-1} a_j z^{kj}$$

Vektor $y = (y_0, y_1, ..., y_{n-1})$ naziva se **Diskretna Furijeova transformacija (DFT)** vektora koeficijenata $a = (a_0, a_1, ..., a_{n-1})$. Takodje se i piše $y = DFT_n(a)$.

8.3 Brza Furijeova transformacija

Korišcenjem metode koja se naziva Brza Furijeova transformacija (FFT - Fast Fourier Transform) možemo izračunati $DFT_n(a)$ za vrijeme $O(n \log n)$, naspram standardnog izračunavanje koje uzima $O(n^2)$. FFT koristi specijalne osobine kompleksnih korjena iz jedinice i tehniku "podijeli i vladaj".

Ideja je da se dati polinom podijeli na dva manja polinoma, stepena $\frac{n}{2}$ koji sadrže koeficijente sa parnim, odnosno neparnim koeficijentima. Označićemo ih sa $P_1(x)$ i $P_2(x)$ sljedece polinome

$$P_1(x) = a_0 + a_2(x) + a_4(x^2) + ... + a_{n-2}(x^{\frac{n}{2}-1})$$
$$P_2(x) = a_1 + a_3(x) + a_5(x^2) + ... + a_{n-1}(x^{\frac{n}{2}-1})$$

Sada polinom $P(x)$ možemo predstaviti kao

$$P(x) = P_1(x^2) + xP_2(x^2)$$

Problem izračunavanja vrijednosti polinoma stepena n, $P(x)$ u tačkama $z^0, z^1, ..., z^{n-1}$ sveden je na dva potproblema izračunavanja vrijednosti polinoma $P_1(x)$ i $P_2(x)$ stepena $\frac{n}{2}$ u tačkama $(z^0)^2, (z^1)^2, ..., (z^{n-1})^2$.

Broj koraka za polinom veličine n je $T(n) = T(\frac{n}{2}) + n$, tj. kompleksnost $FFT_n(a)$ je $O(n) = n \log n$.

Primjena brze Furijeove transformacije je široka. Najznačajnija primjena je u digitalnog obradi zvuka. Kasnije će biti prikazana primjena kod množenja polinoma.

8.4 Implementacija

Ovdje će biti prikazana implementacija brze Furijeove transformacije za polinome realnih koeficijenata.

Pretpostavlja se da je implementirana struktura cn za rad sa kompleksnim brojevima, metode koje vrše osnovne aritmetičke operacije sa kompleksnim brojevima - sabiranje (`complex_plus_complex(cn, cn)`),
oduzimanje (`complex_minus_complex(cn, cn)`) i
stepenovanje (`complex_power(cn, double)`) kao i metoda koja kreira kompleksan broj od njegovog realnog i imaginarnog dijela
(`make_complex_number(double, double)`). Dalje, a je vektor koeficijenata polinoma, y je vektor DFT(a), n dužina vektora i z korjen jedinice za koji se računa vrijednost. Na početku je $z = e^{\frac{2\pi}{n}}$.

```
void FFT(double *a, cn *y, int n, cn z)
{
    if(n == 1)
    {
        y[0] = make_complex_number(a[0], 0);
        return;
    }
    int i, j;

    double *aeven = (double*) malloc(sizeof(double)*n/2);
    double *aodd = (double*) malloc(sizeof(double)*n/2);

    for(i = 0, j = 0; i < n; i=i+2)
        aeven[j++] = a[i];
    for(i = 1, j = 0; i < n; i=i+2)
        aodd[j++] = a[i];

    cn* T = (cn*) malloc(sizeof(cn) * n/2);
    cn* Q = (cn*) malloc(sizeof(cn) * n/2);

    FFT(aeven, T, n/2, complex_power(z,2));
    FFT(aodd, Q, n/2, complex_power(z,2));

    for(i = 0; i < n/2; i++)
    {
        y[i] = complex_plus_complex(T[i],
            complex_multiply_complex(complex_power(z, i),
            Q[i]));
        y[i+n/2] = complex_minus_complex(T[i],
            complex_multiply_complex(complex_power(z, i),
            Q[i]));
    }

    free(T);
    free(Q);
```

```
    free(aodd);
    free(aeven);
}
```

8.5 Množenje polinoma u vremenu $O(n \log n)$

Neka su dati polinomi $P(x)$ i $Q(x)$ stepena n

$$P(x) = \sum_{j=0}^{n-1} a_j x^j$$

$$Q(x) = \sum_{j=0}^{n-1} b_j x^j$$

Potrebno je izračunati njihov proizvod $P(x)Q(x)$ koji ima stepen $2n-2$. "Klasičnim" postupkom, množimo svaki od koeficijenata iz vektora a sa svakim koeficijentom iz vektora b i imamo n^2 množenja, odnosno kompleksnost takvog množenja je $O(n^2)$.

Ukoliko polinom predstavimo na drugačiji način, množenje polinoma može biti efikasnije.

Polinom se može predstaviti, umjesto vektorom svojih koeficijenata, vektorom svojih vrijednosti u n različitih tačaka. Te vrijednosti jednoznačno odredjuju polinom. Ako su polinomi P i Q predstavljeni vrijednostima u n različitih tačaka tada je njihov proizvod PQ odredjen u $2n-1$ tačaka, tj. njihov proizvod se može izračunati u $2n-1$ množenja, tj. u vremenu $O(n)$. Medutim, polinomi su najčešće zadati vektorom svojih koeficijenata, a i predstavljanje polinoma vrijednostima nije pogodno za mnoge primjene. Postavlja se pitanje da li postoji efikasan algoritam koji prevodi polinome iz jednog zapisa u drugi? Odgovor je upravo brza Furijeova transformacija.

Ukoliko nam je polinom dat svojim vektorom koeficijenata i izabaremo n proizvoljnih tačaka, za predstavljanje vrijednostima potrebno je n^2 operacija - za svaku od n tačaka potrebno je n množenja (Hornerova šema). Obrnuti postupak, prelazak sa predstavljanja vrijednostima u n tačaka na vektor koeficijenata naziva se interpolacija. Ona takodje zahtijeva n^2 operacija. Da bismo poboljšali algoritam, nećemo birati n proizvoljnih tacaka, već n korjena iz jedinice - računaćemo vrijednosti polinoma u tačkama $e^{\frac{2\pi}{n}}$.

Brzom Furijeovom transformacijom možemo polinom koji je dat pomoću vektora koeficijentima prevesti u vektor vrijednosti u tačkama $e^{\frac{2\pi}{n}}$ za vrijeme $O(n \log n)$.

Da bi algoritam bio u potpunosti efektivan, potrebno je konstruisati algoritam interpolacije koji ima kompleksnost $O(n \log n)$. Taj postupak se naziva **Inverzna Furijeova transformacija**.

Prisjetimo se da je vektor $y = DFT_n(a)$ dat sljedećim jednakostima

$$y_k = P(z^k) = \sum_{j=0}^{n-1} a_j z^{kj}$$

Inverzna Furijeova transformacija data je jednakostima (Više detalja može se naći u [9] ili [18])

$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} P(z^k) z^{-kj}$$

Primjenom osobina kompleksnih brojeva, lako se dokazuje da ako je z primitivni korjen iz jedinice, tada je to i z^{-1}. To znači da vektor a možemo izračunati primjenom brze Furijeove transformacije, tj. za vrijeme $O(n \log n)$.

Time je dokazano da se polinomi P i Q mogu pomnožiti za vrijeme $O(n \log n)$.

Napominjemo da možemo raditi i u modularnoj aritmetici umjesto u $\mathbb{C}$. Neka je $\mathbb{Z}_m$ skup cijelih brojeva po modulu m. Sljedeća teorema opravdava naš plan.

Teorema 8.1. *Ako su n, ω stepeni 2 $(\neq 1)$, tada je ω prost korijen iz jedinice, po modulu $m = \omega^n + 1$.*

Dokaz. Treba biti $\omega^{\frac{n}{2}} + 1 = 0 (mod\ m)$, ili $\omega^{\frac{n}{2}} = -1 (mod\ m)$. Dokaz kompletiramo koristeći $\sum \omega^{p \cdot i} = 0$, za $0 < p < n - 1$. □

Zadatak 8.1. *Izračunati DFT od (0,1,2,3).*

Zadatak 8.2. *Opisati algoritam koji ima kompleksnost $O(n \log^2 n)$ i koji nalazi polinom $p(x)$ čije su nule tačno $z_1, z_2, \cdots, z_{n-1}$ (dozvoljena su ponavljanja).*

Zadatak 8.3. *Opisati efikasan algoritam za*

- *množenje dvije Toeplitz-ove matrice*
- *za množenje Toeplitz-ove matrice vektorom*

Glava 9

Pronalaženje uzorka u tekstu

9.1 Uvod

Svaki program za editovanje teksta ima mogućnost pretrage teksta i pronalaska unesenog niza karaktera. U ovom poglavlju biće opisani algoritmi koji rješavaju taj problem. Efikasan algoritam koji rješava ovaj problem može se primjeniti i u druge svrhe, ne samo za pretragu nizova teksta, već proizvoljnih podataka, recimo za pronalaženje odgovarajucih nizova u DNK.

Formalno, ovaj problem glasi: Neka je data konačna azbuka Σ i riječi $T[1...n]$ dužine n i $P[1...m]$ dužine m, $m \leq n$ nad datom azbukom. Nizove karaktera T i P takodje nazivamo stringovima. Kažemo da se uzorak P pojavljuje sa pomakom s u tekstu T (ili, ekvivalentno, da se uzorak P pojavljuje u tekstu T sa početkom na poziciji $s+1$) ako za $0 \leq s \leq n-m$ važi $T[s+1...s+m] = P[1...m]$ (tj. da je $T[s+i] = P[i], 1 \leq i \leq m$). Ukoliko se uzorak P pojavljuje sa pomakom s u T, tada s nazivamo ispravan pomak. U suprotnom, za s kažemo da je neispravan pomak. Problem pronalaska uzorka u tekstu je problem pronalaženja svih ispravnih pomaka datog uzorka P u tekstu T.

Najjednostavniji algoritam koji rješava ovaj problem je prosti algoritam grube sile (Brute force). Dovoljno je uporediti sve moguće podnizove niza T sa uzorkom P i tako otkriti da tekst T sadrži uzorak P, tj. posmatrati sve podnizove $t_k, t_{k+1}, ..., t_{k+m-1}$ za $k = 0, 1, ...n-m$ [1] teksta T i uporedjivati ih karakter po karakter sa uzorkom P. Uporedjivanje se vrši sve dok se ne ustanovi da su svi znaci jednaki ili dok se ne naidje na neslaganje. U tom slučaju se podniz pomjera jedno mjesto u desno i ponovo se primjenjuje isti algoritam.

Broj uporedjivanja je manji od mn, pa je složenost ovog algoritma $O((n-m+1)m)$ u najgorem slučaju.

U narednim redovima je prikazan kod algoritma.

[1] Zbog indeksiranja nizova u programskom jeziku C, podrazumijevaće se da prvi element niza ima indeks 0, a ne 1

```
void brute_force(char* pattern, char* text, int m, int n)
{ // m - duzina uzorka, n - duzina teksta
    int i = 0; // indeks uzorka
    int j = 0; // indeks teksta

    while(i < m && j < n)
    {
        if(pattern[i] == text[j])
        {
            i++; j++;
        }
        else
        {
            j = j - i + 1;
            i = 0;
        }
        if(i == m)
        {
            printf("Pojavljivanje na indeksu: %d\n", j - i);
            i = 0;
        }
    }
}
```

U ovom algoritmu, indeks se vraća unazad svaki put kada se naidje na neslaganje. To je upravo ono što ovaj algoritam čini neefikasnim. U narednim poglavljima će biti prikazani optimalniji algoritmi za rješavanje ovog problema.

9.2 Rabin-Karp algoritam

Ovaj algoritam su razvili Michael O. Rabin i Richard M. Karp, 1987. godine, a zasniva se na primjeni heš funkcije koja izračunava dekadnu vrijednost za dati niza cifara.

Pretpostavimo da je data azbuka $\Sigma = 0, 1, 2, ..., 9$, tj. da su karakteri decimalne cifre. Ukoliko to nije slučaj, karaktere možemo posmatrati kao cifre u brojnom sistemu sa osnovom d, gdje je $d = |\Sigma|$. Tada string dužine k možemo posmatrati kao k-tocifreni broj. Npr. ako je data azbuka $\Sigma = 0, 1, 2, ..., 9$ string "12345" možemo posmatrati kao dekadni broj 12345. String "abcdef" možemo predstaviti kao dekadni broj tako što svakom karakteru dodijelimo jednu cifru, npr. $a = 0$, $b = 1$, ..., $z = 26$ i slično. Tako bi string abc u azbuci $\Sigma = a, b, c, ..., z$, $d = |\Sigma| = 26$ imao dekadnu (heš) vrijednosti $a \cdot 26^2 + b \cdot 26^1 + c = 0 * 26^2 + 1 \cdot 26 + 3 = 29$. Ovo je samo jedan od načina heširanja. Za heširanje se može izabrati bilo koja heš funkcija $h : a[\,] \rightarrow \mathbf{R}$.

Sa p ćemo označavati dekadni broj koji odgovara uzorku P a sa t_s dekadne brojeve koji odgovaraju stringovima $T[s...s + m - 1]$ za $s = 0, 1, ..., n - m$.

Vrijednost dekadnog broja u sistemu sa osnovom d možemo izračunati u vre-

menu $O(m)$, pomoću Hornerove sheme:

$$p = P[m-1] + d(P[m-2] + d(P[m-3] + ... + dP[0])))$$

Vrijednost t_0 se na isti način može izračunati u vremenu $O(m)$, a svaka sljedeca vrijednost $t_1, ..., t_{n-m}$ se može ozračunati u konstantnom vremenu, tako što vrijednosti t_i dobijemo pomoću vrijednosti t_{i-1} na sljedeći način:

$$t_i = d(t_{i-1} - d^{m-1}T[i-1]) + T[i+m-1]$$

Prostije rečeno, potrebno je ukloniti cifru najveće težine, i dodati sljedeću cifru iz niza. Npr. ako je $T = 123456$, $d = 10$ i $m = 5$, tada je:

$$t_0 = 12345$$
$$t_1 = 10(12345 - 10000 \cdot 1) + 6 = 23456$$

Ako je vrijednost d^{m-1} poznata tada svako izračunavanje troši konstantan broj aritmetičkih operacija. Da bi riješili problem, potrebno je pronaći pomake s (ako postoje) za koje je $t_s = p$. Medutim i to ne rješava problem u potpunosti, jer $t_s = p$ ne povlači da je $P[0...m-1] = T[s...s+m-1]$, ali zato $t_s \neq p$ odredjuje da $P[0...m-1] \neq T[s...s+m-1]$. Do konačnog rješenja, potrebno je još provjeriti koji od pomaka s za koje važi $t_s = p$ zadovoljava kriterijume, a to se može provjeriti prostim uporedivanjem nizova $P[0...m-1]$ i $T[s...s+m-1]$.

Za male nizove karaktera ovakav pristup je prihvatljiv. Problem može nastati ako su nizovi karaktera veliki i samim tim, ako su vrijednosti p i t_s jako velike, takve da se sa njima ne može manipulisati na uobičajen način (npr. za vrijednosti p i t_s se ne može koristiti osnovni tip podatka za cijele brojeve). Taj problem se može prevazići pamćenjem ostataka pri dijeljenju brojeva p i t_s sa odgovarajućim brojem q, umjesto samih brojeva p i t_s. Ostatak pri dijeljenju broja p sa brojem q može se izračunati u vremenu $O(m)$ a svi ostatci pri djeljenju brojeva t_s sa q u vremenu $O(n-m+1)$. Broj q se bira tako da bude prost i da vrijednost dq može stati u standardni tip podataka kojeg želimo koristiti. U tom slučaju algoritam je identičan, samo što posmatramo ostatke pri dijeljenju sa q:

$$t_i = (d(t_{i-1} - hT[i-1]) + T[i+m-1]) \mod q$$

pri čemu je $h = d^{m-1} \pmod q$.

Ulazni podaci za algoritam Rabin-Karp su tekst T, uzorak P, broj slova azbuke $|\Sigma| = d$, i prost broj q. U primjeru koji slijedi, azbuka je $\Sigma = a, b, c, ..., z$ a vrijednost karaktera je $a = 0$, $b = 1$, ..., $z = 26$.

```
int value(char* s, int n, int d)
{ // racuna vrijednost niza karaktera s,
  // dužine d u sistemu s osnovom d
    int i, result = 0;
    for(i = 0; i < n; i++)
```

```
        result = d*result + s[i] - 'a';
    return result;
}

void RabinKarp(char* T, char* P, int d, int q)
{
    int i, j;
    int n = strlen(T);
    int m = strlen(P);
    int h = power(d, m-1) % q;
    int* t = (int*) malloc(sizeof(int) * (n-m));
    int p = value(P, m, d);
    t[0] = value(T, m, d);

    for(i = 1; i < n-m; i++)
    {
        t[i] = (d*(t[i-1] - (T[i-1]-'a')*h) + T[i+m-1]-'a') % q;
        printf("%d, %d\n", i, t[i]);
    }

    for(i = 0; i < n-m; i++)
        if(p == t[i])
        { // p = t[i], potrebno je ispitati da li su nizovi identicni
            for(j = 0; j < m; j++)
                if (T[i+j] != P[j])
                    break;
            if(j == m)
                printf("Pojavljivanje na indeksu: %d\n", i);
        }

    free(t);
}
```

Algoritam Rabin-Karp u najgorem slučaju troši $O((n-m+1)m)$ vremena, kao i algoritam Brute force, ali u prosjecnom slučaju troši samo $O(n+m)$.

9.3 Boyer-Moore algoritam

Boyer-Moore algoritam je jedan od najčešce upotrebljavanih algoritama u aplikacijama koje imaju funkciju pretrage teksta, a razvili su ga Bob Boyer i J Strother Moore, 1977. godine.

Ovaj algoritam u datom tekstu traži uzorak tako što uporedjuje karaktere uzorka sa desna ulijevo (od kraja uzorka ka početku). Kada se naidje na nepoklapanje, vrši se pomjeranje udesno što je dalje moguće. Postoje dvije vrste pomjeranja - pomjeranje lošeg karaktera (*bad character shift*) i pomjeranje dobrog nastavka (*good suffix shift*).

Prvo pomjeranje omogućuje da se, ukoliko se u tekstu naidje na karakter kojeg nema u uzorku, narednih $m-1$ karaktera može preskočiti, pri čemu je m dužina uzorka.

Na primjer, ako je dat uzorak "AABE" i tekst "BABCAABEABEBBAE" uporedjivanje se vrši na sljedeći način:

$$
\begin{aligned}
&AABE\\
&4 \leftleftarrows BAB\mathbf{C}AABEABEBBAE\\
&1 \leftleftarrows BABCAAB\mathbf{E}ABEBBAE\surd\\
&2 \leftleftarrows BABCAABE\mathbf{A}BEBBAE\\
&2 \leftleftarrows BABCAABEAB\mathbf{E}BBAE\\
&3 \leftleftarrows BABCAABEABEB\mathbf{B}AE\\
&BABCAABEABEBB\mathbf{A}E\_
\end{aligned}
$$

Za ovo pomjeranje potrebno je napraviti tabelu pomjeranja uzorka - za svaki karakter treba odrediti vrijednost pomjeranja udesno.

U prethodnom primjeru, za uzorak "AABE" tabela pomjeranja izgleda

A	B	E	ostali karakteri
3	2	1	4

što znači, kada se u tekstu naidje na karakter A a nema poklapanja, vrši se pomjeranje uzorka udesno za 3 mjesta. Kada se naidje na B, pomjeranje je 2 mjesta itd.

Ideja drugog pomjeranja je da, ukoliko se u tekstu dogodi nepoklapanje a uporedjuje se karakter koji se nalazi negdje u tekstu, pomjeranje se vrši tako da se dogodi poklapanje. Tabela ovih pomjeranja je proširenje prve - umjesto da se posmatraju pomaci za pojedine karaktere, posmatraju se pomaci za podstringove datog uzorka. Ukoliko je uzorak oblika $P[0...m-1]$ posmatraju se podstringovi $P[m-1]$, ... $P[2...m-1]$ i $P[1...m-1]$ i za svaki od njih se računa pomak takav da dodje do poklapanja na mjestima sa desna ulijevo. Ova tabela se naziva tabela pomjeranja dobrog nastavka.

Za uzorak "AEBEAAE", ova tabela izgleda

E⊗	A⊗E	A⊗AE	E⊗AAE	B⊗EAAE	A⊗BEAAE	A⊗ABEAAE
1	3	5	7	7	7	7

Pomoću ove dvije tabele moguće je izvesti pretraživanje na opisani način. Uzorak se u tekstu traži tako što se sa desna ulijevo uporedjuju karatkeri uzorka sa karakterima u tekstu, a ukoliko se dogodi nepoklapanje, pomjeranje se vrši pomoću opisanih tabela.

Algoritam Boyer-Moore, za pronalazak svih pojavljivanja uzorka u tekstu, zahtijeva približno $3n$ uporedjivanja, pa je njegova kompleksnost $O(n)$. Ovo je 1991. godine dokazao Richard Cole (Vd. [7]).

9.3.1 Implementacija Boyer-Moore algoritma

Boyer-Moore algoritam je zgodan za ilustraciju metode transformacionog razvoja programa. U transformacionoj metodi se polazi od jednostavnog, često rješenja "grube sile", pa se kroz niz iteracija rješenje poboljšava, dok se ne dobije rješenje koje nas zadovoljava. Zbog nedostatka prostora, ovdje ćemo dati samo elemente gotovog rješenje problema i oznake i definicije relevantne za preprocesiranje i razumijevanje programa. Više detalja dizajna i analize složenosti algoritma, mogu se naći u knjizi [13] i originalnom radu [6]. U radu [6] je dokazano da BM algoritam nije brz algoritam u smislu "najgoreg slučaja", ali je brz u smislu statističke ("average case") složenosti.

Prilikom poboljšanja algoritma "grube sile" za pronalaženje uzorka u tekstu, sa ciljem da se dobije linearni algoritam, koristi se preprocesiranje. Pri tome se polazi od invarijante $P[0...j] = T[i...i+j]$ i broja simbola koji se slažu u uzorku i tekstu pri pomaku. Uvode se oznake $simbol = T[i+j+1]$, i $p = max(0, j - pomak)$. Tu je $pomak$ najmanji prirodan broj $k \geq 1$ koji zadovoljava odredjeni uslov povezan sa potpunom informacijom (znamo vrijednost j i $simbol$) ili nepotpunom informacijom (znamo vrijednost j i svojstvo $T[i+j+1] \neq P[j+1]$). Za potpunu informaciju imamo uslov:

$$P[0...p] \text{ je sufiks riječi } P[0..j]$$

i

$$((P[p+1] = simbol) \text{ ili } (pomak = j+1)).$$

Uvodi se funkcija $P_0(j) = max\{0 \leq p < j | uslov_0(p, j)\}$, gdje $uslov_0$ označava nepotpunu informaciju. Drugim riječima, $P_0(j)$ je broj simbola koji se slažu u uzorku i u tekstu nakon pomaka pri nepotpunoj informaciji. Funkcija $P_1(j, simbol)$ se uvodi kao broj simbola koji se slažu u uzorku i u tekstu nakon pomaka pri potpunoj informaciji. Funkcija $P1$ se rekurzivno zapisuje

```
P1(j,simbol)=if j=0 then 0;
             else
               if P[P1(j-1,P[j])+1]=simbol then P1(j-1,P[j])+1;
               else P1(P1(j-1,P[j]),simbol))
```

Probajmo naći vezu izmedju P_0 i P_1.

$$P_1(j, simbol) = max(0 \leq k \leq j : k = 0 \text{ ili}$$

$$(P[0...j-1] \text{ je sufiks } P[0...j] \text{ i } P[k] = simbol)).$$

Zbog toga za $j \geq 1$ važi $P_0(j) = P1(j-1, P[j])$.
Iz algoritma koji je predstavljen navedenom rekurzivnom vezom može se računati P_1, ali taj algoritam nema linearnu složenost. Da bi se taj algoritam ubrzao, funkcije P_0 i P_1 se tabuliraju kao nizovi vrijednosti funkcija respektivno i koristi se metod dinamičkog programiranja. Prije početka računanja polja niza $P_0[\ ]$ se inicijaliziraju sa "nije računato".
Neka je s vrijednost pomaka takva da je moguća pojava uzorka počevši od pozicije $i+s$. Definišemo sljedeća tri uslova:

- **uslov 1:** $j - s \leq pozicija(T[i+j])$, gdje je $pozicija(simbol)$ broj prve pojave simbola $simbol$ u uzorku P, sa desne strane uzorka. Na primjer, $pozicija(b) = 3$ za $P = aaba$. Ako se simbol ne pojavljuje u uzorku, onda je $pozicija(simbol) = 0$.
- **uslov 2:** za svaki $j < k \leq m\,(s \geq m)$ ili $(P[k-s] = P[k])$. Posebno, ako je $j+s \leq m$, onda je $P[j...m] = P[j-s...m-s]$. Ovaj uslov izražava saglasnost pomaknutog dijela teksta P sa dijelom koji je posljednji pasovao do teksta T.
- **uslov 3:** $(s \geq j$ ili $(P[j-s] \neq P[j])$. Ovaj uslov slijedi iz nesaglasnosti posljednjih čitanih simbola teksta T i uzorka P.

Ako uvedemo oznake

$$d_1(j) = min\{s \geq 1 | \text{važi } \mathbf{uslov2}\},$$

$$d_2(j) = min\{s \geq 1 | \text{važe } \mathbf{uslov2} \text{ i } \mathbf{uslov3}\},$$

onda možemo napisati sljedeću varijantu Boyer-Moore algoritma:

```
function BM(){
int i=0;
int l=0;
while (i<=n-m){
    j=m;
    while(j>l && P[j]=T[i+j])
        j=j-1;
    if (j=l) then   //Pojava uzorka. Zapamtiti informaciju!
        {printf("%d",i);
        l = m-k;//k=shift(P), pa je k=d2(0)
        j=0; }
    else l=0;
    i=i+d2(j);
        }
    }
```

Algoritam za računanje funkcije $d2$ nije trivijalan i njegova početna verzija data u radu [17] nije bila ispravna, jer je ispušten jedan mogući slučaj [23]. Kod koji navodimo pokazuje način na koji funkciju $d2$ računamo u linearnom vremenu.

```
function racunanjeTabeled2(){
for(j1=1; j1 < m; j1++)
    d2[j1]=m;
P1[0]=1;
for(j=1< m; j++){
    t1=P1[j-1];
    while(P[m-j+1] != P[m-t]){
        s=j-t-1;
        j1=m-t;
        d2[j1]=min(d2[j1],s);
```

```
        t=P1[t];
        };
    if (P[m-t]=P[m-j+1]) then
                            P1[j]=t+1;
                        else{
                        d2[m]=min(d2[m],j-1);
                        P1[j]=0;
                        };
};
\\Racunanje d2[j] za j koje zadovoljava d2[j]>=j
t=P0[m]; q=1; //P0 je izracunat u preprocesiranju
while (t > 0){
    s=m-t;
    for (j=q; j<=s;j++)
        d2[j]=min(d2[j], s);
    q=s+1;
    t=P0[t];
    }
}
```

9.4 Knuth-Morris-Pratt

Knuth, Morris i Prat su 1977. godine objavili algoritam koji je pronalazio uzorak u datom tekstu u linearnom vremenu. Osnovna ideja algoritma je da se obrazac prvo analizira, što omogućava da se izbjegne testiranje svake pozicije teksta.

Algoritam prvo učitava uzorak i formira tabelu h, koja odredjuje koliko znakova treba pomjeriti uzorak udesno u slučaju neslaganja, a zatim se prelazi na traženje uzorka u tekstu.

Pretpostavimo da je do neslaganja došlo na poziciji i u uzorku, tj. za neki indeks j u tekstu je $p_i \neq s_j$ i $p_1p_2...p_{i-1} = s_{j-i+1}s_{j-i+2}...s_{j-1}$. Postavlja se pitanje: koliko uzorak najmanje treba pomjeriti udesno, odnosno koji znak p_l uzorka treba postaviti ispod znaka s_j teksta, tako da se (manji) dio uzorka $p_1p_2...p_{l-1}$, $l < i$, i dalje slaže sa dijelom teksta $s_{j-l+1}s_{j-l+2}...s_{j-1}$, i da pri tome bude $p_l \neq p_i$? Pošto je $l < i$, biće $p_{i-l+1}p_{i-l+2}...p_{i-1} = s_{j-l+1}s_{j-l+2}...s_{j-1}$. Zbog toga je jednakost $p_1p_2...p_{l-1} = s_{j-l+1}s_{j-l+2}...s_{j-1}$ ekvivalentna sa slaganjem $p_1p_2...p_{l-1} = p_{i-l+1}p_{i+l-2}...p_{i-1}$ prefiksa uzorka dužine $l-1$ sa dijelom uzorka koji se završava sa p_{i-1} što je uslov koji ne zavisi od teksta, nego samo od uzorka. Pošto i uslov $p_l \neq p_i$ zavisi samo od uzorka, jasno je da indeks l zavisi samo od indeksa i i znakova uzorka $p_1, p_2, ..., p_i$. Moguće je dakle najpre obraditi uzorak, tako da se u poseban vektor h na poziciju $h[i]$ smjesti izračunati indeks l, $i = 1, 2, ..., m$. Ako ni za jedno $l < i$ nije istovremeno $p_1p_2...p_{l-1} = p_{i-l+1}p_{i-l+2}...p_{i-1}$ i $p_l \neq p_i$, onda se stavlja $h[i] = 0$: p_1 se postavlja ispod s_{j+1}.

Ukoliko je prvo neslaganje na $p_i \neq s_j$, tada se očitava vrijednost $l = h[i]$ i uzorak se pomjera udesno za $i - h[i]$, pa se (ako je $l > 0$) provjerava da li je $p_l = s_j$. Ako jeste, onda se uporedjuju p_{l+1} i s_{j+1}, a ako nije, onda se postupa na već poznat način: očitava vrijednost $h[l]$ a uzorak se pomjera dalje udesno za

$l - h[l]$, itd. Ukoliko je $l = 0$, onda se napreduje u tekstu, tj. p_1 se postavlja ispod s_{j+1}.

9.4.1 Implementacija Knuth-Morris-Pratt algoritma

Implementirana je varijanta Knuth-Morris-Pratt algoritma u kojem se koristi originalno sp' preprocesiranje. Naš kod je adaptacija koda iz paketa $STRMAT$ D. Gusfielda i J. Knight-a i oslanja se na originalan kod iz rada [17]. Detaljna objašnjenja programa mogu se naći u knjizi [13]. Nekoliko novijih metoda za preprocesiranje teksta razvijeno je na osnovu originalne metode KMP i zbog toga smatramo da je poznavanje originalne metode potrebno i korisno.

```
/*
*
*KMP algoritam
*
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#endif
#include "kmp.h"
/*
 *
 * Originalno Knuth-Morris-Pratt preprocesiranje
 * koristenjem spprim vrijednosti.
 *
 */
KMP_STRUCT *kmp_spprime_orig_prep(char *P, int M, int copyflag)
{
  int i, v, *F, *sp, *spprime;
  char x, *buf;
  KMP_STRUCT *vrh;

  P--;            /* Shift da se napravi niz P[1],...,P[M] */

  /*
   * Alocirati sve.
   */
  if ((vrh = malloc(sizeof(KMP_STRUCT))) == NULL)
    return NULL;

  memset(vrh, 0, sizeof(KMP_STRUCT));
  vrh->M = M;
  vrh->copyflag = copyflag;

  if ((F = vrh->F = malloc((M + 2) * sizeof(int))) == NULL) {
    free(vrh);
    return NULL;
```

```
  }
  memset(F, 0, (M + 2) * sizeof(int));

  if (!copyflag)
    vrh->P = P;
  else {
    if ((buf = malloc(M + 2)) == NULL) {
      kmp_free(vrh);
      return NULL;
    }

    buf[0] = buf[M+1] = '\0';
    memcpy(buf + 1, P + 1, M);
    vrh->P = buf;
  }

  if ((sp = malloc((M + 1) * sizeof(int))) == NULL ||
      (spprime = malloc((M + 1) * sizeof(int))) == NULL) {
    if (sp != NULL)
      free(sp);
    kmp_free(vrh);
    return NULL;
  }
  memset(sp, 0, (M + 1) * sizeof(int));
  memset(spprime, 0, (M + 1) * sizeof(int));

  /*
   * Racunaj sp vrijednosti, a zatim spprim vrijednosti.
   */
  sp[1] = 0;
  for (i=1; i <= M - 1; i++) {
    x = P[i+1];
    v = sp[i];
    while (v != 0 && P[v+1] != x) {
      v = sp[v];



    if (P[v+1] == x)
      sp[i+1] = v + 1;
    else
      sp[i+1] = 0;


  }

  spprime[1] = 0;
  for (i=2; i <= M; i++) {
    v = sp[i];
    if (P[v+1] != P[i+1])
```

```
      spprime[i] = v;
    else
      spprime[i] = spprime[v];


  }

  /*
   * Racunaj vrijednosti F.
   */
  F[1] = 1;
  for (i=2; i <= M + 1; i++) {
    F[i] = spprime[i-1] + 1;

  }

  free(spprime);
  free(sp);

  return vrh;
}


/*
 *
 * Knuth-Morris-Pratt algoritam pretrazivanja.
 *
 */
char *kmp_search(KMP_STRUCT *vrh, char *T, int N, int initmatch)
{
  int p, c, M, *F;
  char *P;

  T--;            /* Shifta da se napravi niz T[1],...,T[N] */

  P = vrh->P;
  M = vrh->M;
  F = vrh->F;

  /*
   * Ranuj Knuth-Morris-Pratt algoritam na nizu.
   * Stop na prvoj poziciji koja je saglasna sa
   * kompletnim uzorkom.
   */
  if (!initmatch) {
    c = 1;
    p = 1;
  }
  else {
    c = M + 1;
```

```
    p = F[M+1];

  }

  while (c <= N) {
    /*
     * Uporedi sufiks od P[p],...,P[M] sa T[c],...,T[c+M-p].
     *
     * Ako se zavrsava na kraju uzorka, javi poklapanje.
     * Ako se zaustavio zbog neslaganja na pocetku uzorka,
     * povecaj c.
     * Inace, primijeni funkciju greske na p.
     */

    else {
      vrh->total_shifts += p - F[p];
      p = F[p];
      vrh->num_shifts++;
    }


    while (p <= M && c <= N && P[p] == T[c]) {
      c++;
      p++;
    }

    if (p == M + 1)
      return &T[c-M];
    else if (p == 1)
      c++;
    else
      p = F[p];

#endif
  }

  return NULL;
}


/*
 * kmp_free
 *
 * Oslobodi alociranu strukturu KMP_STRUCT.
 *
 * Parametri:   vrh  -  KMP_STRUCT structura
 *
 * Return:  nista.
 */
void kmp_free(KMP_STRUCT *vrh)
```

```
{
  if (vrh == NULL)
    return;

  if (vrh->copyflag && vrh->P != NULL)
    free(vrh->P);
  if (vrh->F != NULL)
    free(vrh->F);
  free(vrh);
}
```

Zadatak 9.1. *Daunloadovati i instalirati paket* $STRMAT$*, (Vd. [13]), izučiti četiri varijante programa* KMP *navedene u* $STRMAT$*, a zatim ih testirati na računaru.*

□

Dodatak A

A.1 Primjer programa koji ispisuje sopstveni kod

U inženjerstvu bi samorepliciranje moglo omogućiti sniženje cijena proizvodnje. Sljedeći $C++$ program replicira samog sebe:

```
# include <iostream.h>
main()
{
   char *s= "\# include <iostream.h> \%c
     main()
{
 char *s=\%c\%s\%c;
 count.form(s,10,34,s,34,10);}
 \%c";
 count.form(s,10,34,s,34,10);}
```

Autor je lapinski@utexas.edu.
Sljedeći C program veličine 800 bitova, objavljen u časopisu Byte, Avgust 1980.g. - 74.strana, takodje replicira samog sebe.

```
main()
{
 char q=34, n=10,
 *a="main () { char q=34, n=10,  *a= \%c\%s\%c;
 printf(a,q,a,q,n);} \%c";
 printf(a,q,a,q,n);}
```

U svakom Turing kompletnom jeziku, to jest jeziku u kome se može implementirati mašina Turinga, takodje je moguće napisati program koji ispisuje sopstveni kod.

Neki drugi samoreplicirajući mehanizmi su puno složeniji. Za poredjenje, Morisov internet crv ima oko 500 000 bitova, bakterija Esherichia coli ima 8 000 000 bitova, Drekslerov assembler ima 100 000 000 bitova, a čovjek oko 64 000 000 000 bitova.

Gornji primjeri samoreproducirajućih programa daju nadu da će u budućnosti biti moguće napraviti jednostavne samoreproducirajuće mašine koje će pomagati u procesu proizvodnje u nanotehnologijama.

Više detalja i reference mogu se naći u [19].

A.2 Stohastički modeli

Algoritam se u stohastičkoj analizi algoritama tretira kao slučajna promjenljiva i ocjenjuje se koštanje toga algoritma. Za stohastički prostor, na kojem je definisana takva stohastička promjenljiva, uzimamo prostor podataka toga algoritma. Ocjenjivaćemo samo vremensku složenost, to jest broj koraka koje algoritam izvede. Stohastički prostor treba da je skup svih mogućih podataka algoritma, pa za svaki konkretan slučaj moramo precizno definisati stohastički model definišući precizno skup svih mogućih podataka zajedno sa raspodjelom vjerovatnoća.

Pošto razmatramo univerzalne algoritme u smislu da rade sa različitim skupovima podataka, proizvoljne dimenzije, cijena izvodjenja algoritma će zavisiti i od dimenzije podataka i od konkretnog tipa tih podataka. Neka je dimenzija prirodan broj n i neka može biti više tipova podataka za dati algoritam. Ako tip podataka označimo sa d i njegovu dimenziju sa $|d|$, onda će prostor podataka za fiksirano n biti skup

$$D_n = \{d : |d| = n\}.$$

Ako sa $P_n(d)$ označimo vjerovatnoću pojavljivanja u D_n tipa d dimenzije $|d|$, onda za konačan skup D_n imamo da je

$$\sum_{|d|=n} P_n(d) = 1.$$

Cijena algoritma je funkcija n i d, sa $|d| = n$. Ako je svaki D_n jednoelementni skup, onda je cijena funkcija jedne promjenljive n. Ako je D_n netrivijalno i ima odredjenu raspodjelu vjerovatnoće $P_n(d)$, onda je cijena, za fiksirano n, slučajna promjenljiva definisana na D_n.

Cijena izvodjenja algoritma je definisana kao broj koraka koje algoritam izvodi na podacima d, pa za fiksirano n imamo posla za slučajnom promjenljivom T_n koja uzima ne-negativne vrijednosti. Za $|d| = n$ je $T_n(d) \geq 0$.

Pošto je T_n diskretna slučajna promjenljiva, možemo za izračunavanje $ave(T_n)$ (average), $var(T_n)$ (varijanca) i $dev(T_n)$ (devijacija), koristiti tehniku formalnih redova. Označimo sa P_{nk} vjerovatnoću da slučajna promjenljiva T_n ima vrijednost$k \geq 0$. Neka je $P_n(z)$ funkcija generatrisa za T_n. Definišemo je kao red oblika

$$P_n(z) = \sum_{k \leq 0} P_{nk} z^k.$$

Važi sljedeća lema:

Lema A.1.

$$P_n(1) = 1,\ ave(T_n) = P_n^{'}(1),\ var(T_n) = P_n^{''}(1) + P_n^{'}(1) - (P_n^{'}(1))^2$$

Dokaz
Pošto je $\sum_{k\geq 0} P_{nk} = 1$, slijedi da je $P_n(1) = 1$. Formalnim diferenciranjem $P_n(z)$ dobivamo

$$P_n^{'}(z) = \sum_{k\geq 0} kP_{nk}z^{k-1}$$

Sa druge strane, na osnovu definicije matematičkog očekivanja

$$ave(T_n) = \sum_{k\geq 0} kP_{nk}$$

ili $ave(T_n) = P_n^{'}(1)$.
Ponovnim diferenciranjem P_n dobivamo

$$P_n^{''}(z) = \sum_{k\geq 0} k(k-1)P_{nk}z^{k-2}$$

Na osnovu definicije varijance slijedi

$$\begin{aligned}
var(T_n) &= \\
&= \sum_{k\geq 0}(k - ave(T_n))^2 P_{nk} = \\
&= \sum_{k\geq 0}(k^2 P_{nk} - 2kP_n^{'}(1)P_{nk} + P_n^{'}(1)^2 P_{nk}) = \\
&= \sum_{k\geq 0} k^2 P_{nk} - 2P_n^{'}(1)\sum_{k\geq 0} kP_{nk} + P_n^{'}(1)^2 \sum_{k\geq 0} P_{nk} = \\
&= \sum_{k\geq 0} k(k-1)P_{nk} + \sum_{k\geq 0} kP_{nk} - P_n^{'}(1)^2 = \\
&= P_n^{''}(1) + P_n^{'}(1) - (P_n^{'}(1))^2
\end{aligned}$$

□

Ako uspijemo izraziti generatrisu $P_n(z)$ u obliku koji nije formalan red, onda na osnovu dokazane leme možemo izračunati matematičko očekivanje $ave(T_n)$, varijancu $var(T_n)$ i devijaciju $dev(T_n) = \sqrt{var(T_n)}$ i bez računanja vjerovatnoća P_{nk}. *Asimptotske vrijednosti* tih funkcija karakterišu ponašanje algoritma u slučaju slučajnih podataka velikih dimenzija.

Primjer A.1. *1. Ako je $ave(T_n) = O(n)$, $dev(T_n) = O(1)$, onda će očekivano ponašanje algoritma biti linearno u odnosu na dimenziju podataka, a slučajna promjenljiva koja odgovara cijeni algoritma je dobro rasporedjena oko očekivane vrijednosti (pošto je standardno odstupanje konstantno i nezavisno od dimenzije podataka). Može se desiti da se algoritam ne ponaša u skladu sa našim očekivanjima, ali će za slučajne promjenljive najčešće raditi u linearnom vremenu.*

2. *Ako dobijemo $ave(T_n) = O(n \log n)$ i $dev(T_n) = O(n)$, to znači da iako standardno odstupanje nije konstantno, ipak sa rastom n, slučajna promjenljiva koja karakteriše cijenu algoritma sve bolje se grupiše oko vrijednosti $O(n \log n)$.*

3. *Ako je $ave(T_n) = O(n)$ i $dev(T_n) = O(n)$, onda to znači da iako je očekivano vrijeme linearno, ipak slučajna promjenljiva nije dobro grupisana oko svoje očekivane vrijednosti.*

Analizu funkcija $ave(T_n)$ i $dev(T_n)$ treba raditi u slučaju kada se naš algoritam primjenjuje na više raznih slučajno promjenljivih podataka koji imaju veliku dimenziju, a očekujemo da će se maksimalna vrijednost T_n znatno razlikovati od očekivane vrijednosti te promjenljive.

U stohastičkoj analizi raznih algoritama često je korisna sljedeća lema:

Lema A.2. *Neka su $R_n(z) = \sum_{k \geq 0} R_{nk} z^k$, $Q_n(z) = \sum_{k \geq 0} Q_{nk} z^k$ i $P_n = R_n Q_n$ generatrise slučajnih promjenljivih U_n, V_n i T_n. Tada je $ave(T_n) = ave(U_n) + ave(V_n)$ i $var(T_n) = var(U_n) + var(V_n)$.*

Dokaz
Dokaz se izvodi korištenjem Cauchy-jevog množenje redova i diferenciranjem. □

Više detalja o primjeni teorije vjerovatnoće u teoriji algoritama može se naći u [9].

A.3 Problemi zaustavljanja: Izbor sekretara

Neka n kandidata koji su rangirani po vrijednostima od 1 (najgori) do n (najbolji), nastupaju pred nama, jedan po jedan, u slučajnom poretku, pri čemu je svaka od $n!$ permutacija jednako vjerovatna. Kada se pojavi i-ti kandidat, moramo ga prihvatiti (selektovati) ili odbaciti na osnovu observiranog ranga kandidata u odnosu na prethodne kandidate. Pri tome je $1 < i < n$. Ako prije n-tog kandidata nismo nikoga prihvatili, onda zadnji kandidat mora biti prihvaćen. Pretpostavljamo da izabrani kandidat sigurno prihvata ponudu.

U odnosu na kriterije optimalnosti, problem izbora sekretara se često razdvaja na dva standardna problema: problem minimizacije ranga, u kojem je cilj minimizovati očekivani (apsolutni) rang selektovanog kandidata i problem maksimizacije vjerovatnoće, u kojem je cilj da se maksimizira vjerovatnoća selektovanja najboljeg kandidata.

Za rješavanje zadatka biramo sljedeću strategiju. Odrediti f-ti dio intervala od 1 do n, iz kojeg nećemo izabrati kandidata. Zapamtimo najveći rang koji se pojavio u tom intervalu. Nakon tog intervala izabiramo prvog kandidata koji ima veći rang od zapamćenog. Ako se takav kandidat ne pojavi do kraja, onda gubimo tu igru. Kako izabrati f tako da maksimiziramo vjerovatnoću da će izabrani kandidat imati maksimalan rang?

Ako podijelimo sa n brojeve na x-osi i tako skaliramo koordinatni sistem u kojem su na x-osi upisani brojevi kandidata, od 1 do n, a na y-osi rangovi kandidata,

onda u intervalu $(0, f)$ preskačemo kandidate, a van toga intervala, u intervalu $(f, 1)$, izaberemo prvog kandidata koji ima veći rang nego bilo koji kandidat iz intervala $(0, f)$. Ako u $(f, 1)$ ne nadjemo takvog kandidata, onda gubimo igru.

Za uspješan završetak igre najveći rang se mora naći u intervalu $(f, 1)$. Ako položaj kandidata sa tim rangom označimo sa x, onda jedan od sljedećih povoljnih dogadjaja treba da se desi:

- Drugi po veličini rang se nalazi u intervalu $(0, f)$
- Drugi po veličini rang se nalazi u intervalu $(x, 1)$, a treći u intervalu $(0, f)$
- Drugi i treći po veličini rang se nalaze u intervalu $(x, 1)$, a četvrti u intervalu $(0, f)$
- Drugi, treći i četvrti po veličini rang se nalaze u intervalu $(x, 1)$, a peti po veličini rang se nalazi u intervalu $(0, f)$
- itd

Lako se provjeri da su vjerovatnoće opisanih dogadjaja f, $f \cdot (1-x)$, $f \cdot (1-x)^2$, $f \cdot (1-x)^3$, i tako dalje, respektivno. Zbog toga je vjerovatnoća uspjeha, da će se najveći rang naći na poziciji x, jednaka zbiru

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

Pošto je to tačno za svaki x iz intervala $(f, 1)$, možemo pisati

$$P(uspjeh) = P(f) = f \cdot \int_f^1 \frac{dx}{x} =$$

$$= f \cdot (\ln 1 - \ln f) = f \cdot \ln(\frac{1}{f}).$$

Pošto je $P^{'}(f) = \ln \frac{1}{f} - 1$, dobivamo da se maksimalna vjerovatnoća postiže u $f = \frac{1}{e}$.

Vjerovatnoća uspjeha za $f = \frac{1}{e}$ je

$$p\left(\frac{1}{e}\right) = \left(\frac{1}{e}\right) \cdot \ln e = \frac{1}{e} \approx 0.37.$$

Navedena igra daje korisan metod rješavanja problema dinamičke raspodjele memorije. Više detalja o problemima zaustavljaja pri izboru sekretara može se naći u knjizi [9]. Strog matematički dokaz da je interval $(0, f)$ optimalan za sve moguće strategije, dao je E. B. Dynkin koristeći teoriju lanaca Markova. Navedena analiza i rješenje problema zaustavljanja dati su prema knjizi [20], koja se poziva na originalan rad L. Moser-a.

A.4 Teorija složenosti i 3SAT

Moderna era teorije složenosti počela je od teoreme Cook-a. Prikaz konkretnog problema koji je kompletan za **NP** klasu bio je kritičan korak za kasniji razvoj teorije računanja. Ovdje opisujemo neke pojmove iz opšte teorije računanja, a zatim dokazujemo $3SAT$ teoremu Cook-a. Naše izlaganje se bazira na predavanjima Prof. M. Sipsera, koje je držao na University of California, Berkeley. Više detalja se može naći u [27].

Teorija složenosti obuhvata i sljedeće pojmove:

- **Mjere složenosti:** Postoje razne mjere složenosti. U našem izlaganju najčešće će biti korišteni prostor i vrijeme.
- **Modeli računanja:** Postoji čitav niz različitih modela računanja, koji uključuje i deterministički, nedeterministički, alternirajući, paralelni, probabilistički (stohastički) i neuniformni model računanja.
- **Klasifikacione šeme:** Kombinujući različite mjere složenosti sa različitim modelima računanja dolazimo do različitih šema klasifikacije. Ako startujemo od $P \subseteq NP$, možemo produžiti klasifikaciju na obje strane:

  $$L \subseteq NL \subseteq N \subseteq P \subseteq NP \subseteq PH \subseteq PSPACE.$$

  Partikularni problemi se takodje klasifikuju. Na primjer, može se lako dokazati da je $MATCHING$ u $NP \cap co\text{-}NP$. Edmonds je dokazao da je i u P. Dokazano je i da polinomijalan broj stohastičkih procesora može vrlo brzo provjeriti ima li graf prefektno mečiranje. Takodje su interesantne relacije izmedju klasa. Na primjer, može se pokazati da je $BPP \subseteq PH$, što znači da je klasa problema koji mogu biti brzo riješeni uz bacanje novčića, sadržana u polinomijalnoj hijerarhiji.
- **Slučajni nizovi:** Teorija vjerovatnoće igra fundamentalnu ulogu u teoriji složenosti, jer je na neki način šlučajno" opozit od "jednostavno". Izučavaju se razni načini definisanja slučajnosti i generisanja slučajnih nizova.
- $P \stackrel{?}{=} NP$: Veliki broj radova je posvećen pokušajima da se da odgovor na pitanje $P = NP$. To je jedno od osnovnih pitanja savremene kompjuterske nauke.

Determinističke mašine Turinga

Kao model računanja uzimamo mašinu Turinga sa više traka. Medju trakama je jedna koja je ulazna i može se samo čitati (read oly). Za determinističku mašinu Turinga pretposavljamo da ima konačnu kontrolu.

Definicija A.1. $Time(f(n)) = \{L | L$ *je jezik koji je odlučiv na determinističkoj mašini Turinga za vrijeme* $O(f(n))\}$.

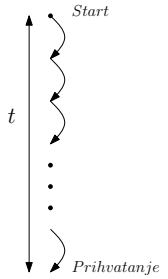
Primjer A.2. *Jezik* $\{a^n b^n | n \geq 0\} \in Time(n)$ *na determinističkoj mašini Turinga sa više traka. Na* DTM *sa jednom trakom je u* $Time(n \log n)$.

Ne-determinističke mašine Turinga

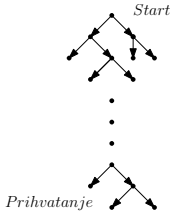
Deterministička mašina Turinga ima u svakom stanju samo jednu tranziciju. Ne-deterministička mašina Turinga ima u nekim stanjima više mogućih tranzicija. $NDTM$ M prihvata ulaz ω ako neka komputacija mašine M prihvata ω.

Definicija A.2. $NTime(f(n)) = \{A | A$ *je jezik koji je odlučiv na* $NDTM$ *koja radi u vremenu* $O(f(n))$*.*$\}$

Na slici A.1 prikazana je komputacija na modelu DTM, a na slici A.2 je prikazano drvo komputacije na modelu $NDTM$. Ne-deterministička mašina Turinga prihvata neku riječ ako neki put komputacije na drvetu vodi do finalnog stanja $NDTM$.



Slika A.1: Niz tranzicija na DTM do finalnog stanja



Slika A.2: Niz tranzicija na $NDTM$ do finalnog stanja

Klase P i NP

Definicija A.3.

$$P = \bigcup_k Time(n^k)$$

$$NP = \bigcup_k NTime(n^k)$$

Klasa P je važna najmanje iz dva razloga:

1. **Invarijantnost.** Klasa je invarijantna u bilo kom prihvatljivom modelu komputacije. Svaki prihvatljiv deterministički model komputacije može simulirati svaki drugi deterministički model komputacije u polinomijalnom vremenu.

2. **Praktičnost.** Polinomijalno vrijeme odgovara klasama problema koji mogu biti stvarno riješeni. Algoritmi sa golemom polinomijalnom složenošću, kao na primjer n^{100}, rijetko se sreću u praksi.

Intuicija. Intuitivno, klasa P odgovara problemima u kojima pripadnost odgovarajućem skupu može biti *odredjena* brzo. Klasa NP odgovara problemima u kojima pripadnost odgovaajućem skupu može biti brzo *verifikovana (provjerena)*.

Primjer A.3.
- **Problem Path je u klasi P:** *dat je graf i dva njegova vrha x i y. Ispitati da li postoji put od x do y.*
- *Problem odredjivanja je li broj $n \in \mathbf{N}$ perfektan kvadrat, takodje je u klasi P.*
- **Problem HPath je u klasi NP:** *dat je graf i dva njegova vrha x i y. Ispitati da li postoji neki Hamiltonov put od x do y.*
- *U klasi NP je i problem odredjivanja je li prirodan broj n prost.*

Zadovoljivost formula

Definicija A.4. *Formula ϕ je* **zadovoljiva** *Booleova formula ako je tačna za neku dodjelu istinitosnih vrijednosti.*

Definicija A.5. $SAT = \{\phi \mid \phi$ *je zadovoljiva*$\}$.

SAT je u NP pošto je jednostavno provjeriti da li data Booleova formula zadovoljava datu dodjelu istinitosnih vrijednosti. S. Cook je dokazao 1971. da je SAT u klasi P ako i samo ako je $P = NP$. Glavna ideja je *redukcija*. Cook je dokazao da svaki problem iz NP može biti sveden na problem zadovoljivosti.

Definicija A.6. *A je u* **polinomijalnom vremenu svodljivo** *na B (piše se $A \leq_p B$), ako postoji funkcija takva da u polinomijalnom vremenu $f : \Sigma^* \to \Sigma^*$ i da je $(\forall x \in \Sigma^*)(x \in A \, akko \, x \in B$.*

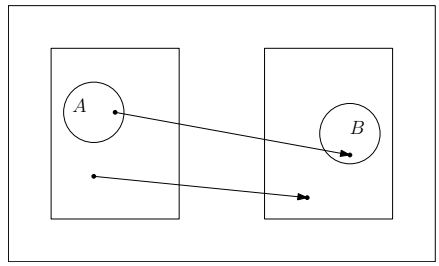
Svodljivost ilustrujemo na slici A.3.
Činjenica. Ako je $A \leq_p B$ i $B \in P$, tada je $A \in P$.

Definicija A.7. *Problem B je* **NP kompletan** *ako je*

1. $B \in NP$
2. $(\forall A \in NP)(A \leq_p B)$.

Teorema A.1. *SAT je NP-kompletan problem.*

U sljedećem odjeljku dokazaćemo ovu teoremu Cook-a.



Slika A.3: Polinomijalna svodljivost

A.4.1 Teorema Cook-a: SAT je NP-kompletan

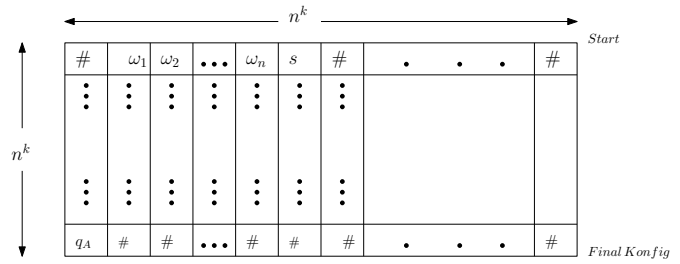
Teorema Cook-a označava početak novog doba u teoriji složenosti računanja. Nalaženje prirodnog konkretnog problema koji je kompletan za NP klasu bio je odlučujući korak za dalji razvoj teoorije računarstva.

Počećemo od

Teorema A.2. *Za svako $A \in NP$ je $A \leq_p SAT$.*

Za dokaz teoreme treba opisati proceduru koja u polinomijalnom vremenu konvertuje pitanje o pripadnosti u A na pitanje o pripadnosti u SAT. Za dato ω, konstuisaćemo string ϕ_ω tako da je $\omega \in A$ ako i samo ako je $\phi_\omega \in SAT$. konstrukcija mora biti efektno izračunljiva.

Pošto je $A \in NP$, postoji ne-deterministička Turingova mašina ($NDTM$) M koja odlučuje A u vremenu n^k. Treba konstruisati formulu ϕ_ω koja je zadovoljiva ako i samo ako M prihvata ω. Znači, formula ϕ_ω u Boole-ovoj algebri izražava činjenicu da M prihvata.



Slika A.4: Tabela koja odgovara računanju $NDTM$

Odredjivanje tabele mašine M za ulaz ω.

Konstruišemo ϕ_ω tako što ćemo opisati tabelu mašine M sa jednom trakom, za riječ ω. Tabela je način da predstavimo komputaciju koja prihvata ω na mašini M. Prvi red tabele predstavlja startnu konfiguraciju mašine M, sa n^k kvadratića na

traci. Svaki od n^k redova u tabeli opisuje konfiguraciju mašine M prikazivanjem sadržaja svakog kvadratića trake. Pri tome je aktuelno stanje mašine opisano simbolom u kvadratiću trake koji je neposredno lijevo od kvadratića koji se skenira. Prema tome, prvi red tabele sadrži string $\#\omega_1\omega_2\ldots\omega_n s\#$. Drugi red predstavlja konfiguraciju mašine nakon jedne tranzicije. Pretpostavljamo da mašina M prihvata kada se nadje u finalnom stanju q_A.

Koliko postoji tabela komputacije mašine M na riječi ω? Lako se vidi da postojanje tabele znači dokaz da M prihvata ulaz ω. Zbog toga je broj različitih tabela jednak broju komputacija koje prihvataju.

Konstrukcija formule ϕ_ω

Formula ϕ_ω tvrdi da tabela od M na riječi ω postoji. Opisaćemo kako se konstruiše ta formula.

Ispitajmo ij ćeliju tabele. Ćelija sadrži neki simbol nad alfabetom M ili oznaku stanja. Uvedimo varijable $x_{ij\sigma_1},\ldots,x_{ij\sigma_l}$, za $\sigma_l \in \Sigma_M \cup Q_M$, gdje je Q_M alfabet koji opisuje stanja M. Posebno, iz $x_{ij\sigma_l}=1$ slijedi da ćelija ij sadrži σ_l. Sada možemo uspostaviti korespondenciju izmedju formula i praznih tabela. Takodje ćemo definisati odnos izmedju dodjela istinitosnih vrijednosti i popunjavanja tabele.

Konstruisanje dijelova formule ϕ_ω

Razložićemo formulu ϕ_ω u dijelove

$$\phi_\omega = \phi_0 \wedge \phi_{start} \wedge \phi_{racunanje} \wedge \phi_{prihvatanje}$$

Boole-ova formula ϕ_0 izražava trivijalnu činjenicu da svaka ćelija u dobro definisanoj tabeli sadrži tačno jedan simbol. Dakle, možemo pisati

$$\phi_0 = \bigwedge_{1\le i,j\le n^k} \left[(x_{ij\sigma_1} \vee x_{ij\sigma_2} \ldots \vee x_{ij\sigma_l}) \bigwedge_{\sigma\neq r} (x_{ij\sigma} \to \overline{x_{ijr}}) \right]$$

Formula ϕ_{start} tvrdi da je prvi red tabele korektan:

$$\phi_{start} = x_{11\#} \wedge x_{12\omega_1} \wedge \ldots \wedge x_{1,n+1,\omega_n} \wedge x_{1,n+2,s} \wedge x_{1,n+3,\#} \wedge \ldots \wedge x_{1,n^k,\#}$$

Sada treba konstruisati formulu $\phi_{racunanje}$. Treba ispitati "okolinu i,j" ćelije i,j, koja po definniciji sadrži šest ćelija hv sa $i \le h \le i+1$ i $j-1 \le v \le j+1$.

Tvrdnja A.1. *Ako je svaka ij okolina ispravna u odnosu na tranzicije mašine, onda je čitava tabela korektna.*

Drugim riječima,

$$\phi_{racunanje} = \bigwedge_{i,j} i,j\ okoline\ su\ korektne\ u\ odnosu\ na\ tranzicije\ masine\ M.$$

Sada samo treba numerisati ispravne okoline. Jedna ispravna okolina ima identične simbole u gornjem i donnjem redu, respektivno. Takva konfiguracija odgovara tranzicijama koje ne utiču na ćelije u okolini. Druge okoline mogu opisivati konsekvence tranzicija u M na povezanu grupu unosa tabele.

Konačno imamo i specifikaciju

$$\phi_{prihvatanje} = x_{n^k,1,q_{finalno}}.$$

3SAT je NP-kompletan

Definišemo $3SAT$ kao skup formula u konjunktivnoj normalnoj formi sa tri literala po klauzuli. Na primjer

$$(a \vee b \vee \overline{c}) \wedge (\overline{a} \vee d \vee e) \wedge \cdots .$$

Primijetimo da formule izvedene u našem dokazu teoreme Cook-a mogu biti prevedene u 3konjunktivnu normalnu formu $(3KNF)$ i da pri tome bude sačuvana zadovoljivost. Formula koja odgovara komputaciji mašine M na riječi ω, u n^k koraka, imaće dužinu $O(n^{2k})$, što je i dalje polinomijalno u odnosu na $|\omega|$.

Pošto znamo da je $3SAT$ u NP, treba još pokazati da je $SAT \leq_p 3SAT$. Za datu $\phi \in SAT$ treba konstruisati $\hat{\phi} \in 3SAT$.

Uzmimo, na primjer, formulu

$$\phi = (a \vee b) \rightarrow c.$$

Konstruišimo standardnu reprezentaciju Boole-ove formule drvetom i uvedimo nove varijable koje odgovaraju neterminalnim vrhovima drveta. Mi možemo, na primjer, definisati y_1 akko $a \vee b$. Onda možemo konstruisati novu formulu

$$(y_1 \leftrightarrow a \vee b) \wedge (y_2 \leftrightarrow (y_1 \rightarrow c))$$

koja odgovara obilasku drveta. Pretvaranje u 3konjunktivnu normalnu formu je sada trivijalno.

Konstrukcija polinomijalne redukcije iz $3SAT$ (ili nekog drugog problema za koji je ppoznato da je NP-kompletan) na proizvoljni problem A je standatdna tehnika za dokazivanje NP-kompletnosti A.

Znamo da je SAT NP-kompletan, ali dosadašnje metode za dokazivanje "težine" problema nisu uspjele pokazati $SAT \notin P$. Malo je vjerovatno da će postojeće tehnike eventualno riješiti pitanje $P \stackrel{?}{=} NP$.

Zadatak A.1. *Problem $2SAT$ je specijalan slučaj problema SAT u kojem svaka klauza sadrži jedan ili dva literala. Dokazati da je $2SAT$ problem polinomijalan.*

A.4.2 Teoreme o vremenskoj hijerarhiji

Parafraziraćemo teoremu koja, u suštini, kaže da postoje proizvoljno složeni problemi.

Teorema A.3. *Za svaku rekurzivnu funkciju $f(n)$, postoji rekurzivan jezik $A \notin Time(f(n))$.*

Dokaz. Neka je $M_1, M_2, \ldots$ numeracija Turingovih mašina. Definišemo mašinu M_A na sljedeći način:

Na ulaz ω, pokrenimo mašinu $M_\omega(\omega)$ za $f(|\omega|)$ koraka. Ako $M_\omega(\omega)$ prihvata u $f(|\omega|)$ koraka, mi odbacujemo, inače prihvatamo. □

Teorema A.3 pokazuje da M_i ne odlučuje A u vremenu $f(n)$. Medjutim, teorema A.3, u gornnjoj formulaciji je netačna. Naime, postoje super brzo rastuće funkcije (slično Busy-Beaver funkcije) tako da su svi rekurzivno nabrojivi skupovi odlučivi u okviru takvih vremenskih granica. Zbog toga treba postaviti restrikcije na funkciju f da bismo dobili tačnu teoremu.

Uvodimo sljedeću definiciju:

Definicija A.8. *Funkcija $f(n)$ je konstruktibilna sa potpunim vremenom ako postoji mašina Turinga koja koristi vrijeme $f(n)$ na ulazima dužine n.*

Sada možemo korektno preformulisati našu teoremu.

Teorema A.4. *Ako je $f(n)$ vremenski konstruktibilna, $F(n) \geq n$ i $g(n)\log g(n) = o(f(n))$, tada je $Time(g(n)) \subset Time(f(n))$.*

Primijetimo da je $g(n)\log g(n) = o(f(n))$ skraćenica za

$$\inf_{n\to\infty} \frac{g(n)\log g(n)}{f(n)} = 0.$$

Dodatni faktor $\log g(n)$ dolazi od simulacije mašine M_ω mašinom M_A.

Zaključak je da je moguće povećati sposobnost Turingove mašine da raspoznaje jezike dajući joj više vremena, sa najmanje logaritamskim povećanjem.

Korolar A.1. *Za svako $1 < \alpha < \beta$, je $Time(n^\alpha) \subset Time(n^\beta)$.*

Slično je $P \subset Time(2^n)$. postoje problemi koji su dokazivo izvan P.

A.4.3 Klase prostorne složenosti

Definicija A.9. $Space(f(n)) = \{A|$ *A je odlučivo pomoću DTM koja koristi najviše $f(n)$ kvadratića na traci*$\}$.

Slično se definiše $NSpace(f(n))$. Ulaz se daje na read-only traku, tako da read-write trake samostalno doprinose prostornoj složenosti.

Definišemo klasu složenosti $L = Space(\log n)$ i $NL = NSpace(\log n)$. Na primjer, problem palindroma

$$\left\{\omega\omega^R | \omega \in 0, 1^*\right\} \in L.$$

Ako je zadat orijentisan ili neorijentisan graf G, onda je problem $PATH \in NL$, gdje je $PATH$ skup trojki (G, a, b) tako da postoji put iz a u b. Logaritam radi

u logaritamskom vremenu tako što koristi pokazivač u graf, ne-deterministički birajući neki put i prateći da li je to cilj b.

Konačno, teoremu o hijerarhiji prostora, možemo formulisati na sljedeći način:

Teorema A.5. *Ako je $f(n)$ prostor-konstruktibilna, $g(n) = o(f(n))$, a $f(n) \geq \log n$, tada je $Space(g(n)) \subset Space(f(n))$.*

A.5 Algoritmi polinomijalne vremenske složenosti za grupe permutacija

Svaka grupa permutacija nad n slova može uvijek biti predstavljena malim brojem generatora. Ovdje pokazujemo kako takva reprezentacija može biti iskorištena za konstruisanje odgovarajućih algoritama koji u polinomijalnom vremenu od n ispituju članstvo u grupi, jednakost grupa i inkluziju grupa. Na osnovu reprezentacije generatorima se takodje može, u polinomijalnom vremenu, izračunati normalno zatvorenje podgrupe, a procedura se može iskoristiti za ispitivanje razrješivosti grupe. Iz reprezentacije generatorima izvodimo i algoritam za računanje presjeka dvije grupe.

Razvoj nabrojanih računarskih algoritama započet je u radu [26]. Problemi složenosti tih algoritama razvijeni su u radovima M. Fursta *et al.*, a razvijene tehnike su primijenjene za popravljanje rezultata o izomorfizmu grafova. Mi ćemo opisati primjene navedenih rezultata u rješavanju problema u vezi sa Rubikovom kockom.

A.5.1 Uvod

Neka je G grupa permutacija nad $\{1, 2, \ldots, n\}$, $n \in \mathbb{N}$. Tipična pitanja o grupi G su:

1. Koliko je velika grupa G?
2. Dato je g. Je li $g \in G$?
3. Je li zadata grupa permutacija jednaka grupi G?
4. Je li zadata grupa permutacija podgrupa grupe G?
5. Je li G razrješiva?

Složenost odgovora na ova pitanja zavisi od reprezentacije grupe G. Broj permutacija od $n \in \mathbb{N}$ je $n!$ Većina grupa ne može biti efektivno izlistana. Grupa G može se definisati preko generatora $g_1, \ldots, g_k$, $k \in \mathbb{N}$. Svi drugi elementi u G mogu se izraziti kao konačni proizvodi nad $\{g_i, g_i^{-1}\}$, $i \in \{1, \ldots, k\}$.

Primjer A.4. *Neka je S_n grupa svih permutacija od n elemenata. Sljedeća dva generatora generišu grupu S_n :*

$$J_1 = \begin{pmatrix} 1 & 2 & \ldots & n \\ 2 & 3 & \ldots & 1 \end{pmatrix}, J_2 = \begin{pmatrix} 1 & 2 & \ldots & n \\ 2 & 1 & \ldots & n \end{pmatrix}.$$

Definicija A.10. *Dužina $g \in G$ nad skupom generatora $\{g_1, \ldots, g_k\}$ grupe G je dužina najkraćeg niza koji daje g, a sastavljen je od g_i^{-1}, g_i, $g_i \in \{g_1, \ldots, g_k\}$. Dijametar grupe G je dužina najdužeg elementa $g \in G$, to jest $max_g min_{rep}|word|$.*

Zadatak A.2. *Dokazati ili opovrgnuti: svaki dijametar grupe S_n, u odnosu na generatore g_1, g_2 je $\Theta(n^3)$.*

Lema A.3. *Svaka grupa $G \subset S_n$ može biti predstavljena pomoću $O(\log_2 n!) = O(n \log n)$ generatora.*

Dokaz. Počinjemo sa proizvoljnim $g_1 \in G$. Označimo sa $G_1 =< g_1 >$ grupu generisanu sa g_1. Ako je $G - G_1 \neq \emptyset$, uzmimo $g_2 \in (G - G_1)$, $G_2 =< g_1, g_2 >$.

Želimo koristiti kosete grupe $\mathcal{C}G$, sa $g_i \cdot G_i$, $i \in \{1, \ldots, n\} \subset \mathbb{N}$ tako da konstruišemo

$$G = G_0 \supseteq G_1 \ldots \supseteq G_n = I = \{e\}.$$

Za dato $G_i \supseteq G_{i+1}$, definišemo G_i/G_{i+1} kao grupu koseta G_{i+1} u odnosu na elemente iz G_i,

$$g \cdot G_{i+1} = \{gh | h \in G_{i+1}\}.$$

Ako sa $|G|$ označimo indeks grupe G, onda možemo formulisati sljedeću

Tvrdnja A.2. *1. Dva koseta su identični ili disjunktni,*

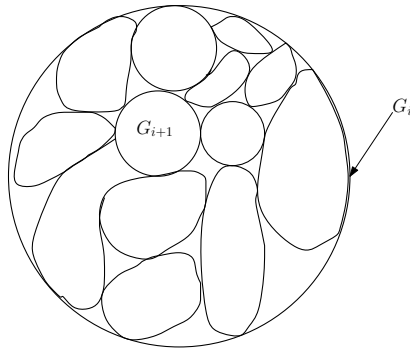
2. Svi koseti imaju isti kardinalni broj,

3. Koseti definišu grupu koju označavamo sa G_i/G_{i+1}, pri čemu je

$$|G_i/G_{i+1}| = \frac{|G_i|}{|G_{i+1}|}.$$

Dokaz. Poznat iz teorije grupa. □

Na slici A.5 prikazana je struktura grupe koseta.



Slika A.5: Struktura grupe koseta

Naš cilj je da predstavimo grupu G nizom u kojem grupe imaju što manje indekse. Definišemo takav niz za grupe permutacija

$$G = G_0 \supseteq G_1 \supseteq G_2 \supseteq \ldots \supseteq G_n = \{e\},$$

pri čemu grupa G_1 fiksira elemenat 1, grupa G_2 fiksira sve elemente iz skupa $\{1, 2\}, \ldots$,a grupa G_n fiksira sve elemente iz skupa $\{1, 2, \ldots, , n\}$.

Neka je g partikularna permutacija koja preslikava 1 u neko ε. Možemo pisati

$$G = G_1^1 \cup G_1^2 \cup G_1^3 \cup \ldots \cup G_1^n,$$

pri čemu je $G_1^i = g^i \cdot G_1$. Jasno je da je svaki indeks ograničen sa n.

Sada možemo pisati

$$G = G_0 = (G_0/G_1) \cdot (G_1/G_2)(G_2/G_3) \ldots (I),$$

gdje smo sa $I = \{e\}$ označili jediničnu grupu. Dakle, svaki elemenat grupe G može se realizovati kao permutacija koja šalje 1 u korektnu vrijednost, množena permutacijom koja fiksira 1, a šalje 2 u korektnu vrijednost, množena sa permutacijom koja fiksira 1 i 2, a šalje 3 u korektnu vrijednost, i tako dalje. Kolekciju elemenata iz G_i/G_{i+1}, $i \in \{0, 1, 2, \ldots, n-1\}$, nazivamo **strogi generatori**.

	1	2	.	.	.	n
G_0/G_1	e					
G_1/G_2	$\emptyset$	e				
		$\emptyset$	e			
			$\emptyset$	e		
	⋮	⋮	⋮	⋮	⋮	⋮
G_{n-1}/G_n						e

Slika A.6: Tabela T strogih generatora grupe G

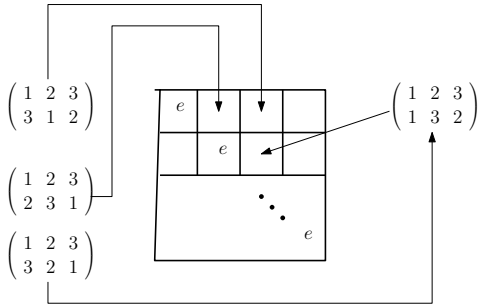
Maksimalna vrijednost koju može imati $|G_i/G_{i+1}|$ je n, pošto se koseti razlikuju samo po tome u šta preslikavaju $i + 1$. Konstruisaćemo tabelu T sa n redova,

labelisanih od 0 do $n-1$, odnosno sa G_i/G_{i+1} i n kolona, kao na slici A.6. U i-tom redu se nalaze predstavnici desnih koseta za G_i/G_{i+1}. Tabela će biti organizovana na takav način da permutacija na poziciji (i, j) fiksira $1, 2, \ldots, i-1$ i preslikava i u j. Odatle izlazi da se unos u tabelu nalazi iznad dijagonale.

Nakon popunjavanja tabela treba imati osobinu da je $g \in G$ ako i samo ako je $g = g_0 g_1 \ldots g_{k-1}$, gdje je g_i član i-tog reda. Ovu reprezentaciju g nazivamo **kanonska reprezentacija**.

Inicijalno u tabelu T na dijagonalu upisujemo identičnu permutaciju, a ostale pozicije u T ostavljamo prazne. Za $x \in G$ treba modifikovati tabelu unošenjem najviše jednog novog predstavnika koseta tako da x može biti napisano u kanonskom obliku. Sljedeći postupak ponavljamo od $i = 0$ i nastavljamo dok je $i < n-1$. Ako u i-tom redu postoji y takvo da y i x preslikavaju $i+1$ u isto slovo, tada inkrementiramo i na $i+1$, a x uzima novu vrijednost $y^{-1}x$. Ako x nije član reda i, tada unosimo x u red i.

Popunjavanje tabele za x ilustrujemo slikom A.7.



Slika A.7: Popunjavanje tabele T strogih generatora grupe G

Primjećujemo da su svi predstavnici koseta nadjeni ako i samo ako

1. svi originalni generatori mogu biti predstavljeni kao proizvod strogih generatora,
2. matrica strogih generatora je zatvorena u odnosu na proizvod.

Algoritam sada možemo opisati na sljedeći način:

1. Predstavimo sve generatore grupe,
2. Zatvorimo tabelu T tako da svaki proizvod xy, za svaki par (x, y) u tabeli, može biti zapisan u kanonskom obliku.

□

Za složenost algoritma treba uzeti u obzir sljedeće:

1. potrebna je jedna permutacija da bismo u tabeli išli jedan nivo na doli,

2. potrebno je $O(n)$ proizvoda da bismo prošli na doli cijelu matricu,

3. Da bismo napunili matricu inicijalno, potrebno je $k \cdot O(n)$ proizvoda, pri čemu je k broj generatora,

4. za zatvaranje matrice u odnosu na parove potrebno je $O(n^4)$.

Slijedi da je ukupna složenost $O(kn + n^5)$ proizvoda permutacija.

Tvrdnja A.3.

$$G =< g_1, \ldots, g_k >=<< g_{ij} >>,$$

pri čemu su $g_1, \ldots, g_k$, originalni generatori, a g_{ij} su proizvodi strogih generatora, po jedan ulaz iz svakog reda.

Dokaz. Neka je, na primjer,

$$G \ni g = g_1 g_3 g_2 = (g^{'}_{i_1}, g^{'}_{2i_2}, \cdots, g^{'}_{ni_n})(g_{1j_1} g_{2j_2} \cdots g_{nj_n})(g_{1k_1} g_{2k_2} \cdots g_{nk_n})$$

Tada treba redom zamijeniti susjedne proizvode, iz raznih zagrada, proizvodom strogih generatora. U navedenom primjeru možemo zamijeniti $g^{'}_{ni_n} g_{1j_1}$ proizvodom $(g_{1l_1} g_{2l_2} \cdots g_{nl_n})$. □

Zadatak A.3. *Koristeći dobiveni algoritam pokazati da je*

$$S_5 = \left\langle \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 3 & 4 & 5 \end{pmatrix} \right\rangle$$

A.5.2 Primjene

Nakon što je nadjena reprezentacija pomoću koseta, ispitivanje kardinalnog broja grupe G i članstva u grupi G nije teško.

1. Data je grupa permutacija G njenim generatorima. Koliki je kardinalni broj grupe G? Za odgovor treba samo izračunati proizvod broja nepraznih ulaza u raznim redovima nadjene tabele. Drugim riječima, kardinalni broj G je proizvod kardinalnih brojeva od G_i/G_{i+1}.

2. Neka je dato neko g. Ispitati je li $g \in G$. Za odgovor treba g dati na ulaz navedenom algoritmu i koristiti izračunatu tabelu. Ako g možemo napisati bez uvodjenja novog elementa u tabelu, onda g možemo napisati kao proizvod generatora. Ako g ne može biti napisano u kanonskom obliku, onda g nije u G.

3. Grupa $G =< g_1, \ldots, g_k >$ sadrži $H =< h_1, \ldots, h_s >$ ako i samo ako je svako h_i član grupe G. Dvije grupe su jednake ako svaka od njih sadrži onu drugu. Dakle, mogućnost ispitivanja članstva u grupi u polinomijalnom vremenu daje polinomijalno vrijeme za ispitivanje inkluzije i jednakosti grupa.

Zadatak A.4. *Neka je G grupa permutacija na $\{1, \ldots, n\}$ generisana sa $g_1, \ldots, g_k$. Neka je H podgrupa u G, prepoznatljiva u polinomijalnom vremenu i sa indeksom koji je najviše polinomijalan po n. Dokazati da generatori za H mogu biti nadjeni u polinomijalnom vremenu.*

Zadatak A.5. *Neka je $H =< h_1, \ldots, h_r >$ podgrupa grupe $G =< g_1 \ldots, g_k >$. Dokazati da se generatori normalnog zatvaranja od H u G mogu izračunati u polinomijalnom vremenu.*

Izvedena podgrupa $G^{'}$ grupe G je grupa generisana svim proizvodima koji imaju oblik $a^{-1}b^{-1}ab$, pri čemu su $a, b \in G$.

Grupa G je rješiva ako se niz $G \supset G^{'} \supset \ldots \supset G^{(i)} \supset \ldots$ završava jediničnom grupom I. Razrješive grupe igraju važnu ulogu u teoriji raširenja polja i direktno su povezane sa uslovima pod kojim polinomijalne jednačine imaju rješenja. Koristeći algoritam o normalnom zatvorenju može se dobiti algoritam koji u polinomijalnom vremenu ispituje razrješivost grupa.

Zadatak A.6. *Dokazati da ako je $G =< g_1, \ldots, g_k >$ grupa permutacija nad $\{1, \ldots, n\}$, onda se razrješivost grupe G može ispitati u polinomijalnom vremenu.*

Neka su svojim generatorima zadate dvije grupe $G =< g_1, \ldots, g_k >$ i $H =< h_1, \ldots, h_r >$. Nije nam poznato da li se za polinomijalno vrijeme, u opštem slučaju, mogu izračunati generatori njihovog presjeka. Ipak, za grupe sa *polinomijalno dostupnim lancima,* odgovor na pitanje je pozitivan.

Definicija A.11. *Neka je $G_0 \supset \ldots \supset G_r = I$ i neka su faktor grupe označene sa G_i/G_{i+1}. Lanac se naziva* **polinomijalno dostupan** *ako zadovoljava sljedeće uslove:*

1. *broj grupa u lancu je polinomijalan po n,*
2. *generatori za grupu G_0 su poznati,*
3. *kardinalni broj grupa G_i/G_{i+1} je ograničen polinomom po n,*
4. *postoji algoritam koji u polinomijalnom vremenu ispituje da li $a, b \in G_i$ pripadaju istom kosetu G_i/G_{i+1}.*

Lako je provjeriti da se za grupe sa *polinomijalno dostupnim lancima* može dokazati analog leme A.3.

Zadatak A.7. *Neka su G i H dvije grupe, prepoznatljive u polinomijalnom vremenu. Neka je S grupa za koju su poznati generatori. Ako S sadrži obje grupe G i H i ako postoje dva* **polinomijalno dostupna lanca***, jedan od S preko G do I, a drugi od S preko H do I, onda generatori za $G \cap H$ mogu biti nadjeni u polinomijalnom vremenu. Dokazati!*

Uputa. Koristiti analog leme A.3 za grupe sa *polinomijalno dostupnim lancima*.

Iz rješenja zadatka A.7 može se izvesti zaključak da se problem presjeka za grupe G i H koje zadovoljavaju

1. G i H se nalaze u zajedničkoj grupi S,

2. postoji lanac grupa izmedju S i G i lanac grupa izmedju S i H čiji indeksi su polinomijalno ograničeni,

nalazi u $NP \cap coNP$.

A.5.3 Algoritmi planiranja, Rubikova kocka i algoritmi učenja

Neka su date bazne operacije. Treba ih komponovati na što efikasniji način pa da se ostvari odredjeni cilj, kao što, na primjer, radi robot prilikom planiranja akcije. Ovakve algoritme nazivamo *algoritmima planiranja*. Algoritme planiranja iskoristićemo za rješavanje problema Rubikove kocke.

Nad permutacijama možemo razmatrati i *algoritme učenja*: Data je složena akcija, a treba pokušati naći tačan efekat individualnih akcija.

Algoritmi planiranja

Neka je data permutacija $g \in G$. Treba odrediti da li se g može predstaviti kao riječ $w \in (\Sigma \cup \Sigma^{-1})^*$, dužine n, a zatim naći takvu reprezentaciju, ako postoji. Ovdje je Σ skup formalnih simbola $\{g_1, \ldots, g_k\}$, a Σ^{-1} je skup njihovih formalnih inverza $\{g_1^{-1}, \ldots, g_k^{-1}\}$. Dužinu riječi i dijametar G definišemo kao i ranije. Daćemo opis algoritma pomoću kojeg možemo naći najkraću reprezentaciju g, raditi sa nepotpuno specifikovanim g i naći stohastičku gornju granicu za dijametar G.

Odgovarajući model formalizujemo kao tranzicioni sistem koji se sastoji od skupa $S = \{1, 2, \ldots, q\}$ stanja okruženja i skupa $\Sigma \cup \Sigma^{-1}$ generatora i njihovih inverza. Tranzicija $i\omega = j$ označava da ako se u stanju i primijeni permutacija ω prelazi se u stanje j.

Teorema A.6 (S. Even i O. Goldreich, 1981)**.** *Odredjivanje vrijednosti $|g|$ je NP-težak problem.*

Dokaz. Redukovati NP-težak problem prekrivanja skupa na problem odredjivanja dužine permutacije. □

Kasnije je M. R. Jerrum, u Theoretical Computer Science, April, 1985, dokazao da je problem odredjivanja dužine reprezentacije neke permutacije *P-space* težak.

Zadatak A.8. *Naći grupu G nad $\{1, \ldots, n\}$, generatore za G i g čija je dužina (u odnosu na generatore) nepolinomijalna po n.*

Ukupan broj riječi dužine n nad $(\Sigma \cup \Sigma^{-1})^*$ je

$$R = \begin{cases} (2k)^n, \text{ ako su dozvoljeni susjedni inverzi,} \\ (2k)(2k-1)^{n-1}, \text{ ako nisu dozvoljeni susjedni inverzi} \end{cases}$$

Problem nalaženja dužine reprezentacije permutacije pomoću generatora može biti riješen iscrpnim pretraživanjem u vremenu $O(R)$ i prostoru $O((k+n)q)$. Pošto je memorija skupa, ideja za rješavanje problema je u razmjeni vremena za prostor (povećavamo vrijeme rada algoritma, a smanjujemo prostor) da se dobije $O(R^{1/2})$

vrijeme i $O(R^{1/4})$ prostor. Uz savremenu tehnologiju algoritam može riješiti probleme predstavljanja do $R = 2^{80}$ mogućih riječi. Algoritam je nepolinomijalan, ali je efikasan za rješavanje problema planiranja, uključujući i problem Rubikove kocke.

Da bismo razmijenili vrijeme za prostor podijelićemo nepoznatu riječ ω, koja reprezentuje permutaciju g, u t podriječi $\omega = \omega_1\omega_2 \dots \omega_t$. Pretpostavljamo da $t|n$ i da sve podriječi ω_i imaju istu dužinu. Sve originalne generatore zamijenimo strogim generatorima koji se sastoje od svih permutacija koje se mogu dobiti komponovanjem n/t generatora iz $\Sigma \cup \Sigma^{-1}$, a onda tražimo reprezentaciju g kao riječi dužine t nad strogim generatorima. Drugim riiječima, imamo problem

Problem A.1. Problem t listi. *Neka je data permutacija g i t listi $L_1, L_2, \dots, L_t$ po m permutacija u svakoj listi. Odrediti kada je $g \in L_1L_2 \dots L_t$.*

Originalni problem reprezentacije može biti posmatran kao problem n listi u kojem svaka lista sadrži originalne generatore i njihove inverze. Iscrpno pretraživanje može biti posmatrano kao problem jedne liste L_1, u kojoj L_1 sadrži sve permutacije iz G čija je dužina n.

U slučaju tri liste $g \in L_1L_2L_3$. Svako L_i sadrži listu svih permutacija generisanih $1/3$ dužine riječi. Koliko je vrijeme? Za $g = w_1w_2w_3$ imamo $w_2^{-1}w_1^{-1}g = w_3$, a odatle bi trebalo $\#^{2/3}$ za vrijeme i $\#^{1/3}$ za prostor.

Zadatak A.9. *Naći opštu vrijeme/memorija razmjenu koja se dobiva za problem tri liste.*

Koristićemo verziju algoritma sa četiri liste za problem reprezentacije. Tada je dužina svake liste $m = (2k)^{n/4} = R^{1/4}$, čime je odredjena prostorna složenost algoritma.

Modifikovanjem permutacija u listama L_i, može se dokazati:

Lema A.4. *Problem odredjivanja kada je $g \in L_1L_2L_3L_4$ može se svesti na problem odredjivanja kada $L_1^{'}L_2^{'} \cap L_3^{'}L_4^{'}$ nije prazan, sa $|L_j| = |L_{i_j}^{'}|$.*

Dokaz. Neka je $L_1^{'} = L_1$, $L_2^{'} = L_2$, $L_3^{'} = \{g\tau^{-1} | \tau \in L_4\}$ i $L_4^{'} = \{\tau^{-1} | \tau \in L_3\}$. Ako je $\tau_1\tau_2 = g\tau_4^{-1}\tau_3^{-1}$, $|tau_i \in L_i$, tada je $g \in L_1L_2L_3L_4$. □

Za nalaženje permutacije h zajedničke za $L_1^{'}L_2^{'}$ i $L_3^{'}L_4^{'}$, treba generisati ulaze u leksikografski rastućem poretku, a onda pretražiti dvije sortirane liste u vremenu koje je linearno u odnosu na njihovu dužinu. Glavni problem je kako generisati m^2 permutacija u $L_1^{'}L_2^{'}$ ili u $L_3^{'}L_4^{'}$ u leksikografski rastućem poretku a da pri tome ne moramo čuvati u memoriji tako mnogo permutacija. Sljedeći "on-the-fly"algoritam nam omogućuje da za date $L_1^{'}$ i $L_2^{'}$ liste permutacija generišemo njihove proizvode $L_1^{'}L_2^{'}$ u rastućem leksikografskom poretku:

1. Sortirati $L_1^{'}$ u leksikografski rastućem poretku, sa najmanjim elementom x_1.
2. Kreirati red sa prioritetom Q koji inicijalno sadrži proizvode x_1y, za sve $y \in L_2^{'}$.

3. Ponavljati sljedeći postupak dok se Q ne isprazni: Obrisati iz Q par xy sa najmanjim proizvodom, štampati proizvod i zamijeniti ga sa $x'y$, gdje je $x^{'}$ sljedbenik od x u $L_1^{'}$. Ako takav sljedbenik ne postoji, dodati ništa.

Na žalost, kompozicija permutaciija je nemonotona operacija, ne samo za leksikografski poredak, već za bilo koji totalan poredak, tako da naš algoritam ne može biti direktno primijenjen. Ako želimo dobivati neprekidno rastući niz proizvoda xy, onda je glavna teškoća pri korištenju ovog algoritma da moramo prolaziti x u $L_1^{'}$ u različitom poretku za svaki $y \in L_2^{'}$. Ovu teškoću možemo prevazići tako što ćemo permutacije u $L_1^{'}$ organizovati kao drvo T, umjesto kao linearnu listu. Drvo T će imati visinu q i svaki čvor drveta ima q sinova, tako da putu od korjena do lista drveta možemo dati ime $< i_1, \ldots, i_q >$, gdje je $i_j \in S = \{1, 2, \ldots, q\}$. Permutacije u $L_1^{'}$ odgovaraju podskupovima listova drveta T. Nekorištene listove odsiječemo, zajedno sa njihovim neiskorištenim precima. Time obezbjedjujemo da je veličina drveta proporcionalna veličini $L_1^{'}$. Prirodan poredak na preostalim listovima je leksikografski poredak imena puteva od korjena drveta do njih. Šta više, za svaku permutaciju y možemo definisati leksikografski poredak $<_y$ na listovima drveta T tako što ćemo preurediti sinove $\{1, 2, \ldots, q\}$ svakog unutrašnjeg čvora V tako da je $1y^{-1} < 2y^{-1} < \ldots < qy^{-1}$. Posjećivanjem listova drveta T u novom poretku $<_y$ možemo dokazati:

Lema A.5. *Za dato drvo T, koje može biti i eksponsncijalno veliko po q, i dvije permutacije x i y, možemo, u vremenu $O(q^2)$ naći, $x^{'}$ u T za koje je, u leksikografskom poretku, $x^{'}y$ najmanja permutacija veća od xy.*

Algoritam iz A.5 možemo sada koristiti za efektivno nalaženje novog para $x^{'}y$ koji treba ubaciti u Q. Pošto svakom y u $L_2^{'}$ pridružujemo sve moguće $x \in L_1^{'}$ i dobivamo neopadajući niz izlaza iz Q, dokazali smo:

Teorema A.7. *Pomoću algoritma četiri liste, problem reprezentacije u grupama permutacija može biti riješen u $O(R^{1/2})$ vremenu i $O(R^{1/4})$ prostoru.*

Rubikova kocka

U suštini je problem Rubikove kocke problem reprezentacije u partikularnoj grupi permutacija koja je generisana sa 18 osnovnih permutacija (rotacije svake od šest strana za 90, 180 ili 270 stepeni), koja operiše na 48 pokretnih podstrana kocke. Jednostavnim prebrojavanjem se može pokazati da je potrebno najmanje 17 poteza za rješavanje većine instanci kocke. U knjizi [4] je pokazano da postoje instance koje zahtijevaju najmanje 18 poteza. Singmaster je postavio hipotezu da je moguće bilo koju instancu Rubikove kocke riješiti u 20 poteza. Najbrži poznat nam algoritam je algoritam dat prema Thislethewaite, koji zahtijeva 52 poteza.

Broj neredundantnih nizova od 20 poteza, koji ne sadrže uzastopne rotacije iste strane, prelazi 2^{78}. Naš algoritam može naći rješenja, za instance Rubikove kocke koje se rješavaju u 20 poteza, u vremenu $O(2^{40})$ i prostoru $O(2^{20})$. To radimo tako što kreiramo listu L od 911250 permutacija koje mogu biti generisane neredundantnim kompozicijama rotacija 5 strana, a zatim koristimo algoritam četiri liste

za nalaženje date instance g u $LLLL$. Nije teško pokazati da će naš algoritam dati kraće rješenje kad god takvo postoji, pa ako je tačna hipoteza Singmastera, naše rješenje predstavlja najbolju dostižnu implementaciju.

Koristeći Simsovu reprezentaciju, možemo sa uniformnom distribucijom u polinomijalnom vremenu izabrati dovoljan broj permutacija $g \in G$, pa iz dužine permutacija probati izračunati dijametar grupe G. Ne može se koristiti najveća dobivena dužina, jer u G mogu postojati samo nekolike permutacije maksimalne dužine. Ipak, lako se dokazuje:

Zadatak A.10. *Ako više od polovine permutacija $g \in G$ ima dužinu ograničenu sa c, tada dijametar grupe G najviše može biti $2c$.*

Kao rezultat zadatka A.10 slijedi da svaki stohastički algoritam za ograničenje srednje dužine permutacija u G, može biti transformisan u stohastički algoritam koji daje granicu dijametra grupe G. Ako se eksperimentalno pokaže da je za Rubikovu kocku dužina ove medijane maksimalno 18, onda zaključujemo da je dijametar grupe G najviše 36. Ovo je dosta više od Singmasterove procjene granice od 20, ali je znatno manje nego do sada dokazana granica Thistlethwaite-a od 52.

Manuelno rješavanje problema Rubikove kocke se obično izvodi u fazama od dna ka vrhu. U svakoj fazi se okreću partikularne podkocke bez vodjenja računa o efektu koji će to ostaviti na dijelove iznad njih. U ovom slučaju permutacije su parcijalno specifikovane, oblika $g =< u_1, u_2, \ldots, u_i, ?, ?, \ldots, ? >$. Ipak, i u ovome slučaju se problem može riješiti sa istom složenošću kao i u kompletno specifikovanom slučaju:

Teorema A.8. *Problem predstavljanja za parcijalno specifikovane permutacije može biti riješen u $O(R^{1/2})$ vremenu i u $O(R^{1/4})$ prostoru.*

Algoritmi učenja

Problem učenja, u slučaju permutacija, u suštini je problem rješavanja sistema jednačina oblika $i\omega = j$, gdje su i, j stanja u sistemu S, a ω je riječ nad nepoznatim permutacijama iz $\Sigma \cup \Sigma^{-1}$.

Primjer A.5. *Neka je $S = \{1, 2, 3\}$ i $\Sigma = \{x, y\}$. Za sljedeći skup jednačina:*

$$\begin{array}{ccc} 1xy = 1, & 2y^{-1}x = 3, & 1xy^{-1} = 2, \\ 3yxy = 1, & 2yxx = 3, & 3x^{-1}yx^{-1} = 2 \end{array}$$

problem je izvesti njihovo jedinstveno rješenje $x =< 3,\ 2,\ 1 >$ i $y =< 2,\ 3,\ 1 >$.

Za sistem jednačina nad proizvoljnim domenom možemo definisati rješivost na dva načina:

1. **Semantička rješivost.** Tačno jedan skup vrijednosti zadovoljava jednačine.
2. **Sintaksna rješivost.** Jedinstveno rješenje može biti izvedeno iz datog skupa jednačina, primjenom konačnog broja transformacija koje čuvaju jednakkost.

Semantička rješivost sistema jednačina nad permutacijama je krajnje težak problem. Vrijedi

Lema A.6. *1. Odredjivanje kada sistem ima najmanje jedno rješennje je NP-kompletan problem.*

2. Odredjivanje kada sistem ima najviše jedno rješenje je co-NP-kompletan problem.

Ipak, nalazeći više i više jednačina, sistem postaje semantički rješiv, ali je teško naći tranzicije ili riješiti jednačine. Dodavanjem više jednačina sistem može postati sintaksno rješiv. Rješenje se može naći u vremenu koje je gotovo linearno.

Ovom prilikom nećemo dalje razmatrati teoriju učenja nad permutacijama.

Bibliografija

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, 1974.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms,* Addison-Wesley, 1983.

[3] M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook,* CRC Press, 1999.

[4] E. R. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways,* Academic Press, 1981

[5] M. Blum et all. *The bounds of selection*, J. Comp. and Sys. Sci., 1972, 7, p. 448 - 461

[6] R. S. Boyer, J. S. Moore, *A fast string searching algorithm*, Comm. ACM, 1977, 20, p. 762 - 772

[7] R. Cole et al. *Optimally fast parallel algorithms for preprocessing and pattern matching in one and two dimensions*, in FOCS'93, 1993, 248–258.

[8] D. Coppersmith and Sh. Winograd, *Matrix multiplication via arithmetic progressions*, J. Symbolic Comput., 9(3):251–280, 1990

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein *Introduction to Algorithms,* 3rd ed., MIT, 2009

[10] J. Edmonds: *Matroids and the greedy algorithms,* Math. Programming, 1971,1, s. 127-136.

[11] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. H. Freeman, 1979

[12] R. L. Graham, *Bounds for certain multiprocessing anomalies,* Bell Systems Technical Journal, 45, 1966, s. 1563 - 1581

[13] D. Gusfield, *Algorithms on strings, trees, and sequences : computer science and computational biology*, Cambridge, 1997

[14] E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer Algorithms,* Computer Science Press, 1998.

[15] D. E. Knuth, *Estimating the efficiency of backtrack programs.* Mathematics of Computation, vol. 29, 1975, 121-136.

[16] D. E. Knuth, *Sorting and Searching, volume 3 of The Art of Computer Programming,* Addison-Wesley, 1973. Second edition, 1998.

[17] D. E. Knuth, J. Morris and V. Pratt, *Fast pattern matching in strings*, SIAM J. Computing, 1977, 6, 323 - 350

[18] U. Manber, *Introduction to algorithms,* Addison-Wesley, 1989

[19] N. Mitić, *Osnovi računarskih sistema*, CET, Beograd 2003.

[20] J. Nivergelt, J. C. Farar, E. M. Reingold, *Computer approaches to mathematical problems*, Prentice-Hall, 1974

[21] Dj. Paunić, *Strukture podataka i algoritmi,* Univerzitet u Novom Sadu, Novi Sad, 1997

[22] R. Rado, *Note on independence function,* Proc. London Math. Soc, 1957, 7, s. 337-343.

[23] W. Rytter, *A correct preprocessing algorithm for Boyer-Moore preprocessing*, SIAM J. Computing, 1980, **9**

[24] R. Sedgewick and Ph. Flajolet, *An Introduction to the Analysis of Algorithms,* Addison-Wesley, 1996.

[25] A. Schonhage, M. Paterson, N. Pipinger, *Finding the median,* J. Comp. Sys. Sci., 1976, 13, pp 184 - 199

[26] C. Sims, *Computational method in the study of permutation groups*, u knjizi John Leech, Computational Problems in Abstract Algebra, Pergamon press, 1970, pp. 171 -189

[27] M. Sipser, *Introduction to the theory of computation*, Second edition, Thomson, 2006.

[28] A. Stothers, *On the complexity of matrix multiplication*, Ph.D. Thesis, U. Edinburgh, 2010.

[29] V. Strassen, *Gaussian elimination is not optimal,* Numerische Mathematik, 14(3):354–356, 1969.

[30] V. Vassilevska Williams, *Multiplying matrices faster than Coppersmith-Winograd*, UC Berkeley and Stanford University, 2011.

[31] M. Živković, *Algoritmi,* Matematički fakultet, Beograd, 2000.