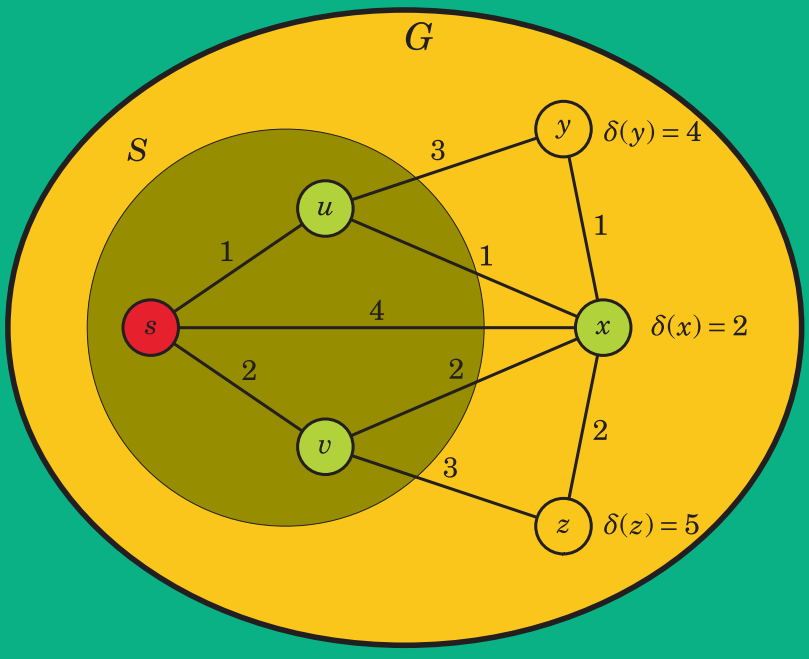
UNIVERZITET SINGIDUNUM

Dejan Živković



UVOD U ALGORITME I STRUKTURE PODATAKA

BEOGRAD, 2010.

UNIVERZITET SINGIDUNUM

Dejan Živković

UVOD U ALGORITME I STRUKTURE PODATAKA

Prvo izdanje

Beograd, 2010.

UVOD U ALGIORITME I STRUKTURE PODATAKA

Autor:
Prof. dr Dejan Živković

Recenzent:
Prof. dr Dragan Cvetković

Izdavač:
UNIVERZITET SINGIDUNUM
Beograd, Danijelova 32
www.singidunum.ac.rs

Za izdavača:
Prof. dr Milovan Stanišić

Tehnička obrada:
Dejan Živković

Dizajn korica:
Dragan Cvetković

Godina izdanja:
2010.

Tiraž:
240 primeraka

Štampa:
Mladost Grup
Loznica

ISBN: 978-86-7912-239-1

Predgovor

Oblast algoritama i struktura podataka potvrdila je svoje mesto u računarstvu zbog praktične važnosti i teorijske elegancije. Sa jedne strane, praktičari mogu da neposredno koriste dobre algoritme i efikasne strukture podataka u složenim programskim projektima. Sa druge strane, za istraživače su algoritmi važan alat kojim se meri stepen složenosti nekog računarskog problema. Algoritmi i strukture podataka se nalaze u centru svih metodologija koje se koriste u radu sa savremenim računarima. Mada na prvi pogled možda tako ne izgleda, neka napredna metodologija sigurno se temelji na sofisticiranim algoritmima i strukturama podataka.

Cilj ove knjige je da čitaocima predstavi osnove ove fascinantne oblasti računarstva. Radi toga se u knjizi prevashodno razmatraju neke od najosnovnijih metoda i paradigmi za dizajniranje i analiziranje struktura podataka i algoritama. Pri tome je pokušano da se naglasak stavi na idejama i lakšem razumevanju koncepata, a ne na implementacionim detaljima.

Ova knjiga je primarno zamišljena kao fakultetski udžbenik za prvo upoznavanje sa algoritmima i strukturama podataka na prvoj ili drugoj godini studija računarstva. Razumevanje materijala iz knjige zahteva bazično predznanje iz računarstva i matematike. Konkretnije, od čitalaca se očekuje da imaju iskustva u pisanju programa na nekom programskom jeziku višeg nivoa kao što su Pascal, C ili Java. Pretpostavlja se i da čitaoci poznaju osnovne matematičke koncepte u koje spadaju skupovi, funkcije i indukcija.

U knjizi se nalazi preko 150 slika i tabela koje služe za ilustraciju materijala. Algoritmi u tekstu su navedeni na jednostavnom pseudo jeziku tako da čitaoci ne moraju da uči novi programski jezik da bi razumeli kako algoritmi rade. Dodatno, na kraju svakog poglavlja se nalazi više zadataka čija težina ide u rasponu od prostih vežbi za utvrđivanje izloženog materijala pa do kreativnih problema. Ovi zadaci se smatraju integralnim delom knjige i čitaoci treba bar da pokušaju da ih reše.

Zahvalnost. Najdublju zahvalnost dugujem svima koji su mi pomogli u pripremi ove knjige. Zahvaljujem se mnogim kolegama i studentima koji su koristili radne verzije knjige i odgovorili na moj zahtev da predlože neka poboljšanja i otkriju neizbežne greške.

Na kraju, bio bih veoma zahvalan čitaocima na njihovom mišljenju o knjizi. Svi komentari i pronađene greške mogu se poslati elektronskom poštom na adresu `dzivkovic@singidunum.ac.rs`.

DEJAN ŽIVKOVIĆ
Beograd, Srbija
januar 2010.

Sadržaj

GLAVA

1

Uvod

Fundamentalni predmet izučavanja u računarstvu su strukture podataka i algoritmi koji ih obrađuju. To je posledica činjenice što, pojednostavljeno rečeno, rad većine računarskih sistema svodi se u suštini na *smeštanje* podataka u memoriji računara i *manipulisanje* podacima iz memorije bitnim za konkretan domen primene. Zbog toga u primenjenim oblastima računarstva, među koje spadaju operativni sistemi, baze podataka, računarska grafika, veštačka inteligencija i druge, izučavaju se zapravo pitanja efikasnog smeštanja podataka i njihovog manipulisanja koja su važna za odgovarajuću oblast primene računara.

U ovoj knjizi međutim, smeštanje podataka i njihovo manipulisanje posmatra se iz vrlo opšteg ugla, uglavnom bez neke konkretne interpretacije podataka na umu. U najopštijem smislu, termin *struktura podataka* koristi se za način organizacije određenih podataka u programu, a termin *algoritam* za postupak obrade podataka. Strukture podataka i algoritmi su međusobno tesno povezani gradivni elementi od kojih se sastoje programski sistemi. Zato se ti delovi obično zajednički predstavljaju u literaturi sa većim ili manjim naglaskom na jedan od njih.

U uvodnom poglavlju se podrobnije definišu koncepti strukture podataka i algoritma, kao i model računara i jezik koji služe za njihovo interpretiranje i izražavanje.

1.1 Podaci

Manipulisanje podacima je glavna svrha rada softverskih sistema i zato je neophodno na samom početku razjasniti sam pojam podataka. U stvari, u ovom kontekstu se često koriste termini „tip podataka" (ili samo kratko „tip"), „apstraktni tip podataka" i „struktura podataka", koji slično zvuče i zato je korisno ukazati na njihove specifičnosti.

Pod *tipom podataka* se podrazumeva skup nekih vrednosti zajedno sa skupom operacija koje su dozvoljene nad tim vrednostima. Tako logički tip podataka sadrži samo dve vrednosti `true` i `false`, a dozvoljene operacije nad njima su uobičajene logičke operacije negacije, konjunkcije, disjunkcije i tako dalje. Slično, tip podataka pod imenom string sadrži vrednosti koje predstavljaju niske znakova, a dozvoljene operacije nad njima su spajanje dve niske, određivanje dužina jedne niske, poređenje dve niske i tako dalje.

Svaki programski jezik obezbeđuje neke primitivne tipove podataka čije vrednosti imaju atomičnu prirodu, odnosno ne sastoje se od manjih delova. Ovi „ugrađeni" tipovi podataka se razlikuju od jezika do jezika, ali većina njih obezbeđuje celobrojni, realni, logički i znakovni tip podataka. Svaki programski jezik sadrži i pravila za konstruisanje složenih tipova podataka polazeći od primitivnih, koji se takođe razlikuju od jezika do jezika.

Apstraktni tipovi podataka su generalizacije primitivnih tipova podataka i rezultat su primene važnog naučnog principa *apstrakcije* u računarstvu. Naime, iskustvo je pokazalo da se uspešna i efikasna realizacija svakog softverskog sistema kojim se rešava složeni problem ne može sprovesti bez podele takvog problema na manje celine koje se mogu nezavisno rešavati. Ali ni ova dekompozicija složenog problema nije dovoljna, već se za rešavanje manjih problema moraju zanemariti mnogobrojni nevažni detalji koji ne utiču na celokupno rešenje. Ovim pristupom *apstrakcije problema* se znatno smanjuje složenost razvoja i održavanja velikih softverskih sistema.

Apstrakcija problema obično ima dva aspekta: proceduralnu apstrakciju i apstrakciju podataka. Proceduralnom apstrakcijom se obezbeđuje razdvajanje onoga što neki modul radi od programskog načina na koji se ta funkcionalnost postiže. Apstrakcijom podataka se obezbeđuje razdvajanje logičke slike informacija koje se obrađuju od njihove fizičke realizacije u računaru. Apstrakcija problema omogućava dakle postupan pristup rešavanju složenog problema, jer se softverski sistem može izgraditi deo po

deo zanemarivanjem mnogih implementacionih detalja o kojma se ne mora voditi računa.

Apstrakcija podataka podrazumeva logički opis kako kolekcije podataka tako i operacija koje se mogu primeniti nad tim podacima. Apstrakcija podataka omogućava programeru da se skocentriše samo na to kako se podaci koriste radi rešenja konkretnog problema. To je značajno olakšanje, jer programer ne mora da se istovremeno bavi načinom predstavljanja i manipulisanja tih podataka u memoriji.

Definisanje logičkog modela podataka i njegova realizacija na računaru obuhvata sledeće elemente:

(1) način na koji su entiteti problema modelirani kao apstraktni matematički objekti;

(2) skup dozvoljenih operacija koji je definisan nad ovim matematičkim objektima;

(3) način na koji su ovi matematički objekti predstavljeni u memoriji računara;

(4) algoritme koji se koriste za programsku realizaciju dozvoljenih operacija nad ovim matematičkim objektima.

Obratite pažnju na to da su (1) i (2) matematički elementi koji određuju prirodu podataka za dati problem, odnosno *šta* su suštinski podaci i kako se obrađuju. S druge strane, (3) i (4) su programski elementi koji određuju *kako* su ti podaci implementirani u računaru. Pojam *apstraktnog tipa podataka* (ili *ATP*) obuhvata elemente (1) i (2), odnosno određuje domen matematički definisanih objekata i skupa operacija koje se mogu primeniti na objekte iz tog domena. S druge strane, pojam *strukture podataka* sadrži elemente (3) i (4) kojima se određuje kako su ovi apstraktni matematički objekti programski implementirani u računaru. Prema tome, za razliku od struktura podataka, apstraktni tipovi podataka su nezavisni od implementacije, pa čak i od programskog jezika.

Na primer, razmotrimo program za simuliranje rada nekog restorana. Ako restoran na meniju ima jelo ražnjiće, onda bi jedan od modula programa verovatno morao da simulira stavljanje komada mesa na žicu ražnjića za posluživanje gosta i skidanje komada mesa sa žice prilikom konzumiranja. Stavljanje komada mesa na žicu ražnjića i njihovo skidanje sa žice može se opisati apstraktnim modelom koji se naziva *stek*.

Stek je apstraktni tip podataka koji ima samo jednu pristupnu tačku — vrh steka. Pri tome, novi podaci se uvek dodaju na vrh steka, a stari podaci

u steku se mogu dobiti i skinuti samo sa vrha steka. Ako komade mesa apstrahujemo kao podatke i žicu ražnjića kao stek, primetimo da postupak rada sa podacima u steku tačno odgovara načinu na koji se postupa sa komadima mesa na žici ražnjića: dodaju se na vrh žice i odatle se skidaju. Naravno, stek kao logički model podataka nije karakterističan samo za banalan primer ražnjića, pa važnost steka proizilazi iz toga što se može prepoznati u mnogim drugim primenama.

S druge strane, stek kao struktura podataka posmatra se sa stanovišta programske realizacije njegove apstraktne organizacije podataka. Pitanja na koja treba odgovoriti u tom slučaju su, recimo, da li je stek bolje realizovati pomoću niza ili povezane liste, kolika je efikasnost operacija dodavanja i uklanjanja podataka iz steka u pojedinim slučajevima i tako dalje.

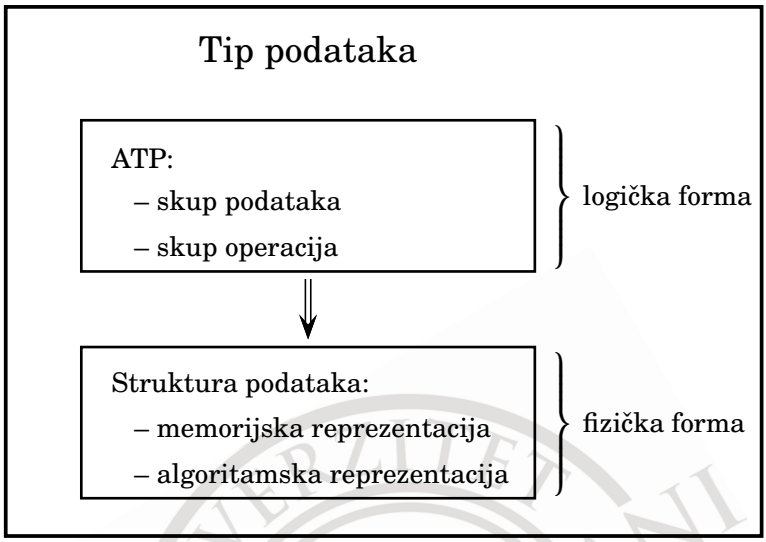
Apstraktni tip podataka definiše interfejs između programera i podataka i zaokružuje neki model podataka u smislu da je implementacija tog modela lokalizovana u jednom delu programa. To je od velike praktične važnosti, jer ako želimo da promenimo realizaciju odgovarajuće strukture podataka, najpre znamo gde to treba učiniti. Takođe znamo da ako promenimo jedan mali deo programa, time nećemo izazvati teško uočljive greške u nekom drugom delu koji nema veze sa odgovarajućim tipom podataka. Na kraju, sa podacima apstraktnog tipa možemo raditi kao da su primitivnog tipa i ne obraćati pažnju na njihovu konkretnu realizaciju.

Apstraktni tipovi podataka imaju dva poželjna svojstva, generalizaciju i enkapsulaciju, koje nam omogućavaju da ispunimo važan cilj softverskog inženjerstva — višekratnu upotrebljivost. Naime, pošto implementiramo apstraktni tip podataka, možemo ga koristiti u različitim programima.

Možemo dakle zaključiti da podaci u računarstvu imaju dva pojavna oblika: logički i fizički. Njihov logički oblik se određuje definicijom apstraktnog tipa podataka, dok se fizički oblik dobija implementacijom odgovarajuće strukture podataka. Ovo zapažanje je ilustrovano na slici 1.1.

1.2 Algoritmi

Spomenuli smo da su algoritmi u tesnoj vezi sa strukturama podataka i da se koriste za programsku realizaciju dozvoljenih operacija nekog apstraktnog tipa podataka. To je ipak samo jedna specifična primena algoritama, a da bismo ih posmatrali opštije, moramo poći od računarskih problema koje algoritmi rešavaju.



SLIKA 1.1: Logička i fizička forma podataka.

Pod *(računarskim) problemom* se podrazumeva zadatak koji treba izvršiti na računaru. Takav zadatak se opisuje preciznom specifikacijom željenog ulazno-izlaznog odnosa kojim se izražava priroda problema. Na primer, *problem sortiranja* niza brojeva može se opisati na sledeći način: ulaz je niz brojeva u proizvoljnom redosledu, a izlaz je niz istih brojeva koji su preuređeni tako da se pojavljuju u rastućem redosledu od najmanjeg do najvećeg. Još jedan primer je *problem vraćanja kusura*: ulaz je neki novčani iznos i apoeni metalnih novčića, a izlaz je najmanji broj tih novčića kojima se dati iznos može potpuno usitniti.

Kod mnogih problema nije dovoljno na ovaj neformalan način definisati njihov ulazno-izlazni odnos koji se traži. Na primer, kod problema sortiranja kako smo ga naveli, nije jasno da li neki brojevi u ulaznom nizu mogu biti jednaki i kako onda jednaki brojevi treba da budu poređani u izlaznom nizu. Zato se u opštem slučaju koristi formalniji, matematički aparat, o kome govorimo u nastavku, kako bi se otklonila svaka dvosmislenost. Iako taj način nećemo posebno naglašavati u ovoj knjizi, jer se koristi opširan prikaz svakog problema, treba znati da je u praksi to dominantan način zbog svoje konciznosti i preciznosti.

Formalno, računarski problem se zadaje *ulaznim parametrima* i odgovarajućim *izlaznim parametrima*, pri čemu obe vrste parametara zadovoljavaju određene uslove kojima se definiše sâm problem. *Instanca*

problema se sastoji od valjanih konkretnih vrednosti za ulazne parametre problema, a *rešenje instance problema* se sastoji od odgovarajućih vrednosti izlaznih parametara za date ulazne vrednosti. Na primer, jedna instanca problema sortiranja je niz $[23,17,8,20,15]$, a rešenje te instance je niz $[8,15,17,20,23]$.

Standardni format za definisanje problema se sastoji od dva dela koji određuju njegove ulazne i izlazne parametre. Oba dela se izražavaju pomoću pogodnih matematičkih objekata, kao što su brojevi, skupovi, funkcije i tako dalje, pri čemu ulazni deo predstavlja generičku instancu problema i izlazni deo predstavlja generičko rešenje ove generičke instance problema. Na primer, problem vraćanja kusura možemo formalno definisati u obliku u kojem se traži da bude zadovoljen ovaj ulazno-izlazni odnos:

Ulaz: Pozitivan ceo broj k („broj metalnih novčića"), niz od k pozitivnih celih brojeva $[c_1,\ldots,c_k]$ („pojedninačni apoeni novčića") i pozitivan ceo broj a („novčani iznos").

Izlaz: Niz od k nenegativnih celih brojeva $[b_1,\ldots,b_k]$ („pojedninačni brojevi novčića sitnine") tako da su zadovoljena dva uslova:

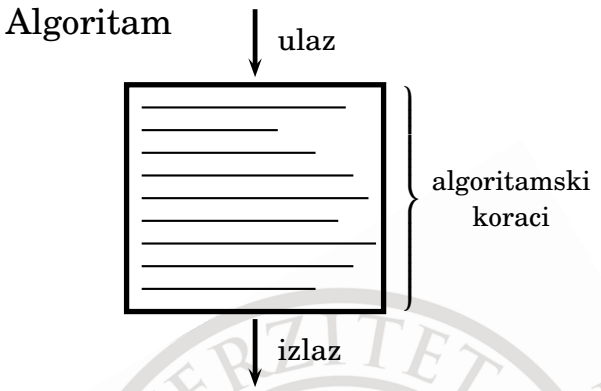
1. $a = \sum_{i=1}^{k} b_i \cdot c_i$ („sitnina je ispravna za dati novčani iznos"), i
2. $\sum_{i=1}^{k} b_i$ je minimalno („zbir novčića sitnine je najmanji").

Jedna instanca problema vraćanja kusura je specifikacija konkretnih vrednosti za ulazne parametre ovog problema, odnosno za broj k, za niz od k elemenata i za broj a. Na primer, $k = 4$, niz $[1,5,10,25]$ od četiri elementa i $a = 17$ predstavljaju jednu instancu problema vraćanja kusura. Rešenje ove instance problema vraćanja kusura je niz $[2,1,1,0]$ od četiri elementa, jer je $17 = 2\cdot 1 + 1\cdot 5 + 1\cdot 10 + 0\cdot 25$ i zbir $2+1+1+0 = 4$ određuje najmanji mogući broj novčića datih apoena za usitnjavanje iznosa od 17 novčanih jedinica.

Sada kada znamo šta su računarski problemi možemo preciznije govoriti o algoritmima, jer oni rešavaju te probleme. Pre svega, *algoritam* je tačno definisana računarska procedura koja pretpostavlja neke podatke kao *ulaz* i proizvodi neke podatke kao *izlaz*. Ovo prvenstveno znači da se algoritam sastoji od jasne i nedvosmislene specifikacije niza koraka koji se mogu mehanički izvršiti na računaru (slika 1.2).

Za dobru intuitivnu sliku algoritma može poslužiti analogija sa kuhinjskim receptom za neko jelo: baš kao što se kuhinjski recept sastoji od niza prostijih koraka koje izvršava kuvar, tako se algoritam sastoji od niza računarskih koraka koje izvršava računar. Ulazni podaci za recept su

potrebni sastojci za kuvanje, a dobijeni izlaz posle „izvršavanja" recepta je spremljeno jelo.



SLIKA 1.2: Pojednostavljena slika algoritma.

Drugo, algoritam *rešava* neki problem ukoliko se na ulazu algoritma pretpostavlja ulaz tog problema i na izlazu algoritma se dobija rešenje tog problema za svaku instancu problema. Na primer, algoritam koji rešava problem sortiranja na ulazu dobija neuređen niz brojeva i kao rezultat na izlazu proizvodi niz istih brojeva u rastućem redosledu.

Algoritam za neki problem je dakle precizno naveden niz instrukcija koje računar treba da izvrši radi rešavanja tog problema, odnosno radi transformisanja svake instance problema u njeno rešenje. Naravno, jedan problem može imati više algoritama koji ga rešavaju.

Na osnovu svega do sada rečenog možemo sumirati glavne odlike svakog algoritma:

- Algoritam mora biti postupak koji se sastoji od konačno mnogo koraka koji se mogu izvršiti na računaru.
- Algoritam mora biti postupak po kojem je nedvosmisleno određen svaki sledeći korak za izvršavanje.
- Algoritam mora biti *ispravan (korektan)* postupak. To prvenstveno znači da se za svaku instancu problema na ulazu algoritma mora dobiti njeno odgovarajuće rešenje na izlazu algoritma. Ali to obuhvata i uslov, koji ćemo podrazumevati da je ispunjen, da za ulaz algoritma koji nije instanca problema, izlaz algoritma nije bitan, ali se izvršavanje algoritma mora završiti u svakom slučaju.

Kreativni proces pisanja algoritama kojim se dolazi do nedvosmislenih instrukcija od kojih se sastoji jedan algoritam naziva se *dizajn algoritma*. Za ovaj proces ne postoji čarobna formula koja se može primeniti u opštem slučaju, već se od dizajnera složenog algoritma zahteva i doza lucidnosti. Ipak, postoje standardni metodi koje možemo koristiti da bismo algoritamski rešili mnoge tipove problema. Tu spadaju algoritamske paradigme kao što su rekurzivni algoritmi, „pohlepni" algoritmi, algoritmi dinamičkog programiranja, randomizirani algoritmi i tako dalje.

U postupku *analize algoritma* se kvantitativno određuje upotreba računarskih resursa nekog algoritma. Najskuplji resursi koje obično želimo da minimizujemo su vreme izvršavanja i memorijski zahtevi nekog algoritma, i prema tome govorimo o njegovoj vremenskoj i memorijskoj složenosti. Ovde treba obratiti pažnju na to da mera veličine resursa koje neki algoritam koristi umnogome zavisi od modela računara na kojem se algoritam izvršava. U ovoj knjizi se krećemo isključivo u granicama modela sekvencijalnih mašina — modeli paralelnih računara i paralelni algoritmi zaslužuju posebnu knjigu.

1.3 Računarski model

Ono što je ostalo nedorečeno iz prethodne diskusije o algoritmima jesu računarske instrukcije (ili koraci) koje čine algoritam. Iako je to, nadamo se, intuitivno jasno čitaocima koji imaju programersko iskustvo, radi veće jasnoće potrebno je ipak nešto više reći o tome koje nam instrukcije tačno stoje na raspolaganju kada pišemo neki algoritam. Drugim rečima, potrebno je malo detaljnije opisati model računara na kojem se izvršavaju algoritmi o kojima smo govorili u prethodnom odeljku.

Uopšteno govoreći, računarski modeli su pojednostavljeni opisi stvarnih računara kojima se zanemaruje njihova puna složenost, osim nekoliko svojstava za koje se smatra da najbolje odslikavaju fenomen koji se izučava. Pošto je merenje računarskih resursa (vreme i prostor) koje algoritmi koriste jedan od predmeta našeg izučavanja, to znači da najpre treba precizno formulisati model računarske mašine na kojoj se algoritmi izvršavaju i definisati primitivne operacije koje se koriste prilikom izvršavanja. Međutim, nećemo ići u toliko detalja za opis formalnog računarskog modela, već ćemo u nastavku samo neformalno opisati glavne odlike računarskog modela sa kojim radimo. To je dovoljno za našu svrhu, jer će nas interesovati asimptotske performanse algoritama, odnosno njihova efikasnost za veliki broj

ulaznih podataka. Pored toga, tačni računarski modeli su zaista potrebni za dokazivanje donjih granica složenosti algoritama. Pošto se time nećemo baviti u ovoj knjizi, nema potrebe da komplikujemo izlaganje nepotrebnim detaljima.

Osnovna pretpostavka je da koristimo računarski model koji predstavlja idealizovan „realan" računar sa jednim procesorom. Ne postavljamo nikakvu gornju granicu na veličinu memorije takvog računara, ali pretpostavljamo da programi koji se na njemu izvršavaju ne mogu da modifikuju sami sebe. Sa druge strane, u ovom modelu ne postoji memorijska hijerarhija, na primer keš ili virtualna memorija, kao kod pravih računara.

Tačan repertoar instrukcija ovog modela računara nije posebno bitan, pod uslovom da je skup tih instrukcija sličan onom koji se može naći kod pravih računara. Zato pretpostavljamo da postoje aritmetičke operacije (za sabiranje, oduzimanje, množenje, deljenje, deljenje sa ostatkom, celobrojni deo broja), logičke operacije (i, ili, negacija), ulazno-izlazne operacije (čitanje, pisanje), operacije za premeštanje podataka (kopiranje iz memorije u procesor i obratno), operacije za pomeranje bitova podataka levo ili desno za mali broj pozicija, kao i upravljačke operacije (za uslovno i bezuslovno grananje, poziv i povratak iz potprocedure). Ove operacije rade nad svojim operandima koji su smešteni u memorijskim rečima čije se adrese određuju korišćenjem direktnih ili indirektnih šema adresiranja.

Model računara koji se koristi je generički jednoprocesorski računar sa proizvoljnim pristupom memoriji. Ova mašina se popularno naziva RAM (skraćenica od engl. *random-access machine*) i rad sa njom je vrlo sličan radu sa pravim računarima na mašinskom jeziku. Drugim rečima, podrazumeva se da će algoritmi biti realizovani u obliku mašinskih programa koji se izvršavaju na RAM-u.

Osnovni podaci kojima se manipuliše u RAM-u su brojevi celobrojnog tipa i oni sa pokretnim zarezom. Dodatno pretpostavljamo da postoji granica veličine memorijske reči tako da brojevi koji se čuvaju u njima ne mogu rasti neograničeno. Na primer, ako je veličina memorijske reči jednaka w bita, tada u našem modelu možemo predstaviti cele brojeve koji su manji ili jednaki 2^{w-1} po apsolutnoj vrednosti. Brojevi sa pokretnim zarezom su predstavljeni u standardnom IEEE formatu.

U sekvencijalnom modelu RAM-a se instrukcije izvršavaju jedna za drugom bez paralelizma. Zato pre nego što možemo analizirati složenost nekog algoritma, moramo znati i vreme koje je potrebno za izvršavanje svake instrukcije, kao i to koliki memorijski prostor zauzima svaki operand. U tu svrhu ćemo koristiti *jedinični model složenosti*, u kojem se svaka

instrukcija izvršava za jednu jedinicu vremena i svaki operand zauzima jednu jedinicu memorijskog prostora.

Ovaj model je odgovarajući ako se može pretpostaviti da se svaki broj koji se pojavljuje u programu može smestiti u jednu memorijsku reč. Ako program izračunava vrlo velike brojeve, koji se ne mogu realistično uzeti kao operandi mašinskih instrukcija, a jedinični model složenosti se koristi sa nepažnjom ili zlom namerom, poznato je da možemo dobiti lažnu sliku o efikasnosti programa. Ali, ako se jedinični model složenosti primenjuje u svom duhu (na primer, rad sa velikim brojevima se deli u više jednostavnijih koraka i ne koriste se nerealistično moćne instrukcije), tada je analiza složenosti u ovom modelu umnogome jednostavnija i precizno odražava eksperimentalne rezultate.

Ukratko, u dizajnu i analizi algoritama i struktura podataka vodimo se time kako rade pravi računari. Naime, našem idealizovanom računaru ne želimo da damo nerealistično velike mogućnosti. S druge strane, voleli bismo da nam on omogućuje komforan rad bar sa nizovima podataka sa proizvoljnim pristupom, kao i sa osnovnim operacijama čiji su operandi srednje veliki brojevi.

1.4 Zapis algoritama

Sledeće pitanje koje se postavlja u vezi sa algoritmima je notacija koja se koristi za opis postupka koji čini neki algoritam. Da bismo precizno izrazili korake od kojih se sastoji neki algoritam, možemo koristiti, po redosledu rastuće preciznosti, prirodan jezik, pseudo jezik i pravi programski jezik. Nažalost, lakoća izražavanja ide u obrnutom redosledu.

Algoritamski postupak često obuhvata komplikovane ideje, pa bismo za opis algoritama želeli da koristimo zapis koji je prirodniji i lakši za razumevanje nego što je to odgovarajući mašinski program za RAM, ili čak i računarski program na programskom jeziku visokog nivoa. Zbog toga, jedini uslov za zapis algoritama je da specifikacija odgovarajućeg postupka obezbeđuje precizan niz instrukcija koje računar treba da izvrši. To znači da, u principu, možemo koristiti običan jezik za algoritamske korake. Ali u složenijim slučajevima, da bismo nedvosmisleno izrazili algoritamske korake, potrebno je ipak koristiti i programske konstrukcije.

Obratite pažnju na to da pojam algoritma nije isto što i konkretniji pojam računarskog programa. Računarski program se izvršava na pravom

računaru, pa se zato piše na nekom programskom jeziku poštujući njegova striktna pravila. Znamo da su tu važni svako slovo, svaka zapeta i svaki simbol, tako da nemamo pravo na grešku prilikom pisanja ako hoćemo da napisani program izvršimo na računaru. Za algoritme nam ne treba takav apsolutni formalizam — na kraju krajeva, algoritmi se ne izvršavaju na pravim i „glupim" računarima. Algoritmi se mogu zamisliti da se izvršavaju na jednoj vrsti idealizovanog računara sa neograničenom memorijom. Oni su u stvari matematički objekti koji se mogu analizirati da bi se prepoznali suštinski delovi složenih problema. Stoga, vrlo je bitno da možemo apstrahovati nevažne detalje programskog jezika. Još jedna razlika između algoritama i pravih programa je da nam kod algoritama nisu bitna pitanja softverskog inženjerstva. Na primer, da bi se suština algoritama prenela što jasnije i jezgrovitije, obrada grešaka kod algoritama se obično zanemaruje.

Algoritmi su obično mali po veličini ili se sastoje od kratkih delova naredbi prožetih tekstom koji objašnjava glavne ideje ili naglašava ključne tačke. Ovo je uglavnom prvi korak za rešavanje problema na računaru pošto sitni, nevažni detalji programskih jezika mogu samo da smetaju za dobijanje prave slike problema. Ovo ne znači da algoritam sme da sadrži bilo kakve dvosmislenosti u vezi sa vrstom naredbi koje treba izvršiti ili sa njihovim tačnim redosledom izvršavanja. To samo znači da možemo biti nemarni da li neki znak dolazi ovde ili onde, ako to neće zbuniti onoga koji pokušava da razume algoritam. Pored toga, neki elementi koji su formalno neophodni kod računarskih programa, na primer, tipovi podataka, obično se podrazumevaju iz konteksta bez navođenja.

Zbog ovih razloga se algoritmi najčešće neformalno izražavaju na *pseudo jeziku* koji sadrži elemente kako prirodnog tako i programskog jezika. U stvari, u ovoj knjizi se često običnim jezikom najpre opisuje glavna ideja algoritma, a zatim se prelazi na pseudo jezik da bi se pojasnili neki ključni detalji algoritma.

Pseudo jezik koji koristimo u ovoj knjizi za zapis algoritama sadrži elemente programskih jezika C, Pascal i Java, tako da čitalac koji je programirao na ovim jezicima neće imati nikakvih problema da razume napisane algoritme. U tom pseudo jeziku se koriste uobičajeni koncepti programskih jezika kao što su promenljive, izrazi, uslovi, naredbe i procedure. Ovde nećemo pokušavati da damo precizne definicije sintakse i semantike pseudo jezika koji se koristi u knjizi, pošto to nije potrebno za našu svrhu i svakako prevazilazi namenu ove knjige. Umesto toga, u nastavku navodimo samo kratak spisak konvencija u korišćenom pseudo jeziku:

1. Osnovni tipovi podataka su celi brojevi, realni brojevi sa pokretnim zarezom, logički tip sa vrednostima `true` (tačno) i `false` (netačno), znakovi i pokazivači. Dodatni tipovi podataka kao što su stringovi, liste, redovi za čekanje i grafovi se uvode po potrebi.
2. Promenljive i (složeni) tipovi podataka se ne deklarišu formalno, već se njihovo značenje uzima na osnovu njihovog imena ili iz konteksta.
3. Komentari se navode iza simbola dve kose crte (`//`).
4. Naredbe petlje (ciklusa) `for-do`, `while-do` i `repeat-until`, kao i naredbe grananja `if-then-else` i `case`, imaju uobičajenu interpretaciju. Promenljiva brojača u `for` petlji zadržava svoju vrednost nakon završetka petlje. Klauzula `else` u `if` naredbi nije naravno obavezna.
5. Svaka naredba se završava tačkom-zapetom (`;`). Jedan red algoritma može sadržati više kratkih naredbi.
6. Sve promenljive su lokalne za algoritam u kome se koriste. Isto ime za dve različite promenljive se nikad ne koristi. Globalne promenljive su uvek eksplicitno naglašene i koriste se samo kada su potrebne. Podrazumeva se da one postoje u odgovarajućem spoljašnjem okruženju.
7. Operator dodeljivanja se označava simbolom za jednakost (=). Na primer, izrazom `x = e` se promenljivoj x dodeljuje vrednost izraza e.
8. Nizovi elemenata su indeksirani od 1 do broja tih elemenata, a rad sa elementima niza se zapisuje navodeći ime niza i indeks željenog elementa u (uglastim) srednjim zagradama. Na primer, `a[i]` označava i-ti element a_i niza a.
9. Logički operatori konjunkcije, disjunkcije i negacije se označavaju simbolima &&, `||` i `!`. Logički izrazi u kojima učestvuju ovi operatori se uvek navode sa svim zagradama. Ti izrazi se skraćeno izračunavaju sleva na desno. To jest, da bi se izračunala vrednost izraza `p && q`, najpre se izračunava p. Ako je vrednost za p jednaka netačno, vrednost za q se ne izračunava pošto vrednost izraza `p && q` mora biti netačno. Slično, za izraz `p || q` se izračunava q samo ako je izračunata vrednost za p jednaka netačno, jer u svim ostalim slučajevima vrednost izraza `p || q` mora biti tačno. Logički izrazi koji se izračunavaju na ovaj način omogućavaju nam da ispravno koristimo uslove kao što je `(i <= n) && (a[i] == 0)` čak i u slučaju kada indeks i premašuje veličinu n niza a.
10. Blokovska struktura složenih naredbi naznačava se jedino uvlačenjem. Ovaj način isticanja logičke strukture kratkih algoritama umesto uobičajenih simbola za ograničavanje bloka naredbi (na primer, { i } ili `begin` i `end`), zapravo doprinosi boljoj čitljivosti.

11. Neformalne naredbe sa elementima običnog jezika koriste se svuda gde takve naredbe čine algoritam razumljivijim od ekvivalentnog niza programskih naredbi. Na primer, naredba

    for (svaki čvor v grafa G) **do**

    mnogo je izražajnija od njene detaljnije, ali irelevantne programske realizacije.

12. Zaglavlje algoritma počinje službenom rečju `algorithm`, a iza nje se navodi ime algoritma. Za imena algoritama se koriste engleski nazivi, jer je to uobičajeno u stručnoj literaturi.

13. Algoritmi se koriste na dva načina. Jedan način je kao funkcija koja daje jedan rezultat. U tom slučaju poslednja izvršena naredba u algoritmu mora biti naredba `return` iza koje sledi neki izraz. Naredba `return` dovodi do izračunavanja tog izraza i do kraja izvršavanja algoritma. Rezultat algoritma je vrednost izračunatog izraza. Drugi način korišćenja algoritma je kao potprogram. U ovom slučaju naredba `return` nije obavezna, a izvršavanje poslednje naredbe ujedno predstavlja kraj izvršavanja algoritma.

14. Algoritam može komunicirati sa drugim algoritmima pomoću globalnih promenljivih i navođenjem parametara algoritma. Ulazni (ili ulazno/izlazni) parametri se pišu u zaglavlju algoritma iza imena algoritma u zagradama, a izlazni parametri se pišu iza naredbe `return`. U pozivu algoritma se navode stvarni argumenti umesto njegovih parametara koji se prenose na transparentan način kojim se ispravno tretiraju formalni ulazni, ulazno/izlazni i izlazni parametri algoritma. Argumenti složenih tipova podataka se prenose samo po referenci, odnosno pozvani algoritam dobija pokazivač na stvarni argument.

15. Podrazumeva se da je rekurzija omogućena.

Da bismo praktično ilustrovali neki od ovih elemenata pseudo jezika koji koristimo, u nastavku navodimo algoritme za dva jednostavna problema.

Primer: zamena vrednosti dve promenljive

Za date dve promenljive, problem je zameniti njihove vrednosti tako da nova vrednost svake promenljive bude stara vrednost druge promenljive. Ako ne znamo ništa o tipu vrednosti datih promenljivih, potrebna nam je pomoćna promenljiva koja služi za privremeno čuvanje stare vrednosti

jedne od promenljivih, pre nego što u tu promenljivu upišemo vrednost druge promenljive. Na pseudo jeziku se ovaj postupak izražava preciznije na sledeći način:

```
// Ulaz:  promenljive x i y sa svojim vrednostima
// Izlaz: promenljive x i y sa zamenjenim vrednostima
algorithm swap(x, y)

   z = x;
   x = y;
   y = z;

   return x, y;
```

Algoritam swap služi prvenstveno za ilustraciju zapisa na pseudo jeziku, jer je njegov postupak za rešenje problema zamene vrednosti dve promenljive očigledan.

Primer: najveći element niza

Ako je dat niz a od n neuređenih brojeva $a_1, a_2, \ldots, a_n$, problem je naći indeks (prvog) najvećeg elementa tog niza. Prirodno rešenje je da početno pretpostavimo da je prvi element niza najveći. Zatim proveravamo sve ostale elemente tako što redom ispitujemo aktuelni element da li je veći od najvećeg elementa nađenog do tada. Ako je to slučaj, taj aktuelni element postaje najveći od onih ispitanih do sada; u suprotnom slučaju, nije potrebno ništa dodatno uraditi.

```
// Ulaz:  niz a, broj elemenata n niza a
// Izlaz: indeks najvećeg elementa niza a
algorithm max(a, n)

   m = a[1];  // najveći element nađen do sada
   j = 1;     // indeks najvećeg elementa

   i = 2;     // proveriti ostale elemente niza
   while (i <= n) do
      if (m < a[i]) then  // nađen veći element od privremeno najvećeg
         m = a[i];        // zapamtiti veći element
         j = i;           // ... i njegov indeks
      i = i + 1;  // preći na sledeći element niza

   return j;  // vratiti indeks najvećeg elementa
```

Primetimo da se postupak algoritma max zasniva na induktivnom rezonovanju: ako znamo da je element a_j najveći u podnizu $a_1, \ldots, a_i$, da bismo

našli najveći element u podnizu $a_1,\ldots,a_i,a_{i+1}$ dužem za jedan element, neophodno je i dovoljno samo uporediti elemente a_j i a_{i+1}. U algoritmu se upravo ovo radi za $i = 1,\ldots,n$, tako da se na kraju ispravno određuje najveći element niza a.

Zadaci

1. Problem *pretrage niza* je:

Ulaz: Broj n, niz a od n neuređenih brojeva $a_1,a_2,\ldots,a_n$, broj x.

Izlaz: Indeks niza k takav da je $x = a_k$ ili 0 ukoliko se broj x ne nalazi u nizu a.

a) Navedite nekoliko instanci problema pretrage i odgovarajuća rešenja.

b) Napišite iterativni algoritam kojim se broj x traži u nizu a ispitivanjem tog niza od njegovog početka.

2. Problem *preostalog dela kredita* je:

Ulaz: Ukupan iznos kredita (principal) p, godišnja kamatna stopa k (na primer, $k = 0.08$ označava kamatnu stopu od 8%), iznos mesečne rate za kredit r, broj meseci m.

Izlaz: Iznos neotplaćenog kredita p_m posle m meseci.

a) Navedite nekoliko instanci problema preostalog dela kredita i odgovarajuća rešenja.

b) Napišite iterativni algoritam za izračunavanje p_m ukoliko su date vrednosti za p, k, r i m. (*Savet:* Svakog meseca, iznos kredita se povećava usled zaračunavanja mesečne kamate i smanjuje usled otplate mesečne rate. Ako uvedemo oznaku $a = 1 + k/12$ i stavimo $p_0 = p$, pokažite da je posle $m = 1,2,\ldots$ meseci preostali deo kredita $p_m = ap_{m-1} - r$.)

UNIVERZITET
PRIVATNI
AKREDITOVANI
UNIVERZITET
SINGIDUNUM

GLAVA

2

Dizajn i analiza algoritama

Prilikom pisanja algoritama se mora obratiti posebna pažnja na njihova dva aspekta: ispravnost i efikasnost. Kod dizajna algoritama se obraća posebna pažnja na prvo pitanje ispravnosti (korektnosti) algoritama. To je naročito važno za složene algoritme kod kojih njihova ispravnost nije očigledna, već zahteva pažljiv matematički dokaz. Cilj analize algoritama su kvantitativne ocene efikasnosti algoritama. To je takođe važno pitanje, jer je čak i ispravan algoritam beskorisan ukoliko njegovo izvršavanje traje nepraktično dugo ili zauzima veliku količinu memorije.

Efikasnost algoritma je mera veličine računarskih resursa, obično su to vreme i memorija, koje algoritam koristi tokom svog izvršavanja. Intuitivno, ta veličina računarskih resursa umnogome zavisi od količine i strukture ulaznih podataka za algoritam. Na primer, ako uzmemo analogiju algoritama i kuhinjskih recepata, jasno je da izvršavanje recepta za pravljenje pica zahteva više vremena i posuđa kada se to radi za 10 osoba nego za dve osobe. Prema tome, da bismo odredili efikasnost algoritma, treba da odredimo računarske resurse koje algoritam koristi tokom izvršavanja kao funkciju veličine ulaza. Sa druge strane, ova funkcija je obično vrlo komplikovana, pa da bismo pojednostavili njenu analizu možemo uzeti maksimalno moguće vreme izvršavanja (ili memorijsku zauzetost) za sve ulaze algoritma iste veličine. U knjizi se obično bavimo ovim ocenama u najgorem slučaju, mada ćemo na odgovarajućim mestima pominjati i druge vidove mere performansi algoritama.

Drugi aspekti dobrih algoritama su takođe važni. Tu naročito spadaju jednostavnost i jasnoća. Jednostavni algoritmi su poželjni zato što ih je

lakše pretvoriti u računarske programe, jer dobijeni programi tada najverovatnije nemaju skrivenih grešaka koje se manifestuju tek u proizvodnom okruženju. Naravno, algoritam je bolji ukoliko je lako razumljiv onome ko čita njegov opis na pseudo jeziku. Nažalost, u praksi ovaj zahtev za efikasnim i jednostavnim algoritmima je često kontradiktoran, pa moramo pronaći balans između njih kada pišemo algoritme.

U vrlo opštim crtama, postupak dobijanja dobrih algoritama za većinu problema obuhvata ova tri koraka:

1. Najpre se eleminišu svi nevažni i remetilački detalji praktičnog problema koji se rešava, kako bi se dobio računarski problem u što čistijem obliku. To obično znači da se problem predstavlja u apstraktnim terminima prostih matematičkih pojmova kao što su skupovi, funkcije, grafovi i slično.
2. Pošto je problem jasno matematički definisan, piše se algoritam koji ga rešava i dokazuje se njegova ispravnost. Ovo se većim delom izvodi uz pomoć poznatih algoritamskih paradigmi i matematičkih alata.
3. Na kraju se sprovodi analiza algoritma da bi se ustanovila njegova efikasnost. Pri tome se, pošto ocena efikasnosti algoritma može biti vrlo komplikovana, primenjuju određene tehnike kojima se znatno pojednostavljuje analiza. O ovome će biti više reči u ostalom delu knjige.

Naravno, ovi koraci nisu potpuno nezavisni jedan od drugog i često se međusobno prepliću. Isto tako, za prostije probleme nisu potrebni svi koraci ovog postupka u svojoj punoj opštosti.

2.1 Dizajn algoritama

Dizajn dobrih algoritama za složene probleme zahteva od programera visok stepen obrazovanja, intuicije i iskustva. Vrhunski programeri moraju dobro poznavati, između ostalog, domen problema koji se rešava, računarstvo i matematiku. Čak ni ova svestranost programera ne garantuje uspeh, jer je dizajn algoritama kreativan proces za koji ne postoji unapred propisan šablon. Ali sve ovo ipak nije toliko crno, jer se za rešavanje različitih klasa problema mogu primeniti neki standardni metodi koji obezbeđuju dobre algoritme. U ove opšte algoritamske paradigme spadaju rekurzivni metod, „pohlepni” metod, dinamičko programiranje i randomizacija.

Za dobre algoritme ne postoji čarobna formula, ali bez obzira na to kako se dolazi do nekog algoritma, prvi i najvažniji uslov koji dobijeni algoritam

mora da zadovolji jeste da ispravno rešava dati problem. Na putu dokazivanja ispravnosti algoritma postoje mnoge prepreke, kako praktične tako i teorijske. Što se tiče praktičnih teškoća, one su u vezi sa činjenicom da se većina programa piše tako da zadovoljavaju neke neformalne specifikacije, koje su i same nepotpune ili protivrečne. Prepreke na teorijskoj strani su to što, uprkos brojnim pokušajima da se formalno definiše značenje programa, nije se iskristalisao opšte prihvaćeni formalizam kojim se pokazuje ispravnost netrivijalnih algoritama.

U svakom slučaju, za skoro sve algoritme je potrebno istaći i pokazati njihove bitne odlike. Za jednostavne algoritme je obično dovoljno kratko objašnjenje njihovih glavnih odlika. Odlike o kojima je reč su najčešće formalna tvrđenja koje odslikavaju način na koji konkretan algoritam radi. Onaj ko piše algoritam bi bar trebao da ima u vidu te odlike, iako je možda nepraktično da se one dokazuju u svim detaljima. Te odlike treba da služe i kao određen vodič u kreativnom procesu pisanja nekog algoritma.

Primer: najveći zajednički delilac

Da bismo prethodnu diskusiju ilustrovali jednim prostim primerom, razmotrimo problem izračunavanja najvećeg zajedničkog delioca dva pozitivna cela broja x i y. *Najveći zajednički delilac* za par pozitivnih celih brojeva x i y označava se sa $\text{nzd}(x,y)$ i predstavlja najveći ceo broj koji bez ostatka deli oba broja x i y. To jest, najveći zajednički delilac je broj $d = \text{nzd}(x,y)$ koji deli i x i y bez ostatka, pa $x = md$ i $y = nd$ za neke brojeve $m,n \in \mathbb{N}$,[1] i svaki drugi takav zajednički delilac brojeva x i y je manji od d.

Da li je ovo dobra definicija? Drugim rečima, da li svaki par pozitivnih celih brojeva ima najveći zajednički delilac? Jasno je da svaki par pozitivnih celih brojeva ima broj 1 za zajedničkog delioca, a najveći broj koji istovremeno može da deli x i y bez ostataka je onaj broj koji je minimum od x i y. Dakle, $\text{nzd}(x,y)$ uvek postoji i leži negde u intervalu između 1 i $\min\{x,y\}$.

Ovo zapažanje možemo iskoristiti za postupak kojim se traži $\text{nzd}(x,y)$ tako što se redom proveravaju svi potencijalni kandidati za najveći zajednički delilac brojeva x i y. Ovi kandidati su uzastopni celi brojevi od broja jedan do minimuma brojeva x i y. Pri tome se ti brojevi proveravaju obrnutim redom da li je neki od njih zajednički delilac za x i y, sve dok se ne otkrije prvi zajednički delilac. Pošto se provera obavlja obrnutim redom od

[1] U ovoj knjizi sa $\mathbb{N}$ označavamo skup $\{0,1,2,\ldots\}$ nenegativnih celih brojeva.

najveće do najmanje mogućnosti, prvi nađeni zajednički delilac biće ujedno i najveći.

```
// Ulaz:  pozitivni celi brojevi x i y
// Izlaz: nzd(x,y)
algorithm gcd(x, y)

   d = min{x,y};
   while ((x % d != 0) || (y % d != 0)) do
      d = d - 1;

   return d;
```

Iako je ispravnost algoritma gcd nadamo se očigledna, pokušajmo da na formalniji način pokažemo da taj algoritam ispravno izračunava nzd(x, y). U ovom konkretnom slučaju je formalni dokaz koji je izložen u nastavku verovatno suviše detaljan, ali pokazuje sve principe koje treba slediti u opštem slučaju.

Pre svega, ključni korak u dokazivanju ispravnosti iterativnih algoritama (kao što je gcd) jeste pokazati da petlje zadovoljavaju određene uslove. Neophodan uslov za svaku petlju je da njen izlazni uslov bude zadovoljen u jednom trenutku kako bi se izvršavanje petlje sigurno završilo.

Druga vrsta uslova su oni čija se tačnost ne menja izvršavanjem petlje. Takav jedan uslov se naziva *invarijanta petlje* i predstavlja formalni iskaz koji je tačan pre svake iteracije petlje. To znači da će invarijanta petlje biti tačna nakon izvršavanja poslednje iteracije petlje (ili pre prve neizvršene iteracije), odnosno ona će biti tačna nakon završetka rada cele petlje. Prepoznati pravu invarijantu petlje je važna veština, jer znajući invarijantu petlje i izlazni uslov petlje obično možemo zaključiti nešto korisno o ispravnosti samog algoritma.

Da bismo pokazali da se invarijanta petlje ne menja izvršavanjem petlje, moramo pokazati dve činjenice o invarijanti petlje:

1. Invarijanta petlje je tačna pre prve iteracije petlje.
2. Ako je invarijanta petlje tačna pre neke iteracije petlje, ona ostaje tačna i nakon izvršavanja te iteracije petlje (to jest, pre izvršavanja sledeće iteracije).

Prema tome, u konkretnom slučaju algoritma gcd pokažimo najpre da se while petlja sigurno završava, odnosno da nije beskonačna. Primetimo da se izlaznim uslovom te petlje proverava da li d nije delilac broja x ili broja y bez ostatka. Ako je to slučaj, telo petlje se ponavlja; u suprotnom

slučaju, petlja se završava. Da bismo videli zašto će izlazni uslov naposletku postati netačan, primetimo da je početno $d = \min\{x, y\}$, a da se d u svakoj iteraciji smanjuje za 1. Ovim uzastopnim smanjivanjem će d u najgorem slučaju postati jednako 1, izlazni uslov će onda biti netačan pošto je istovremeno $x \,\%\, 1 = 0$ i $y \,\%\, 1 = 0$, pa će se petlja tada završiti.

Sada još moramo pokazati tačnost izračunavanja $\text{nzd}(x, y)$ u algoritmu `gcd`. Odgovarajuća invarijanta `while` petlje je uslov $d \geq \text{nzd}(x, y)$. Jasno je da je ovaj uslov zadovoljen pre prve iteracije te petlje, jer d početno dobija vrednost $\min\{x, y\}$ i uvek je $\min\{x, y\} \geq \text{nzd}(x, y)$. Dalje, treba pokazati da ako ovaj uslov važi pre neke iteracije, on će važiti i posle izvršavanja te iteracije. Zato pretpostavimo da je vrednost promenljive d veća ili jednaka vrednosti $\text{nzd}(x, y)$ pre neke iteracije, i pretpostavimo da je ta iteracija `while` petlje izvršena. To znači da je izlazni uslov petlje bio tačan i da se vrednost promenljive d smanjila za 1. Pošto je izlazni uslov bio tačan, stara vrednost promeljive d *nije* bila delilac oba broja x i y, pa zato stara vrednost promeljive d nije jednaka $\text{nzd}(x, y)$ nego je striktno veća od $\text{nzd}(x, y)$. Pošto je nova vrednost promenljive d za jedan manja od njene stare vrednosti, možemo zaključiti da je nova vrednost promenljive d opet veća ili jednaka vrednosti $\text{nzd}(x, y)$ pre sledeće iteracije. Ovim je dokazano da $d \geq \text{nzd}(x, y)$ predstavlja invarijantni uslov `while` petlje u algoritmu `gcd`.

Na kraju, ispitajmo koji uslov važi kada se `while` petlja završi. Iz uslova prekida te petlje vidimo da se ona završava kada je vrednost promenljive d delilac oba broja x i y. Pošto je $\text{nzd}(x, y)$ najveći delilac oba broja x i y, sledi da nakon završetka `while` petlje važi $d \leq \text{nzd}(x, y)$. Ali pošto nakon završetka `while` petlje važi i invarijanta petlje $d \geq \text{nzd}(x, y)$, sledi da mora biti $d = \text{nzd}(x, y)$. Pošto se ova vrednost promeljive d vraća kao rezultat algoritma `gcd`, to pokazuje da ovaj algoritam ispravno izračunava $\text{nzd}(x, y)$.

2.2 Analiza algoritama

Pošto smo napisali algoritam za dati problem i pokazali da je algoritam ispravan, sledeći važan korak je određivanje veličine računarskih resursa koje taj algoritam koristi tokom izvršavanja. Ovaj zadatak ocene efikasnosti nekog algoritma se naziva analiza algoritma. To ne treba shvatiti kao nezavisan i odvojen proces od dizajna algoritma, već se oni međusobno prožimaju i podjednako doprinose krajnjem cilju dobijanja ispravnih, efikasnih i elegantnih algoritama.

Dva najvrednija računarska resursa koje svaki algoritam treba da štedi su vreme i memorija. Neki algoritam kome trebaju meseci da bi završio svoj rad ili koji koristi nekoliko gigabajta radne memorije nije baš mnogo koristan, iako je možda potpuno ispravan. Mogući su i drugi kriterijumi za efikasnost algoritama kao što su, na primer, količina mrežnog saobraćaja u toku rada ili količina podataka koji se dvosmerno prenose na diskove. Međutim, u ovoj knjizi skoro ekskluzivno proučavamo vremensku složenost algoritama — memorijski zahtevi biće pomenuti samo ako nisu u „normalnim" granicama. Drugim rečima, efikasnost algoritma identifikujemo sa dužinom njegovog vremena izvršavanja.

Kako možemo odrediti vreme izvršavanja nekog algoritma? Dva glavna načina da se to uradi su empirijski i analitički. Empirijski način podrazumeva da se prosto izmeri stvarno vreme rada odgovarajućeg programa na računaru. Analitičko određivanje vremena izvršavanja zasniva se na teorijskoj analizi rada algoritma i prebrojavanju jediničnih instrukcija koje se izvršavaju tokom rada algoritma.

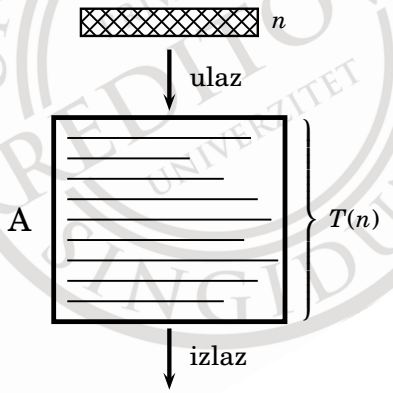
Dobre strane empirijskog pristupa su da se to može relativno lako uraditi i da se dobijaju tačni vremenski rezultati. Ali nedostaci ovog pristupa su ozbiljniji i odnose prevagu, pa se zato ovaj pristup retko primenjuje u praksi. Pre svega, empirijsko merenje se može primeniti samo na program napisan u celosti, jer je uticaj pojedinih delova na ukupno vreme teško proceniti. Možda još gore, empirijski izmereno vreme rada programa važi samo za poseban slučaj ulaznih podataka i za jedan konkretan računar pod određenim programskim režimom rada. Na osnovu jednog merenja ne može se nešto opštije zaključiti o vremenu izvršavanja istog algoritma za različite ulazne podatke, niti o tome kakvo će vreme izvršavanja biti na nekom drugom računaru koji radi pod drugačijim okruženjem.

Osnovna ideja analitičkog pristupa je jednostavna: analizirati kako radi algoritam korak po korak i na osnovu toga zaključiti koliko će biti vreme njegovog izvršavanja. Ovo je lako reći, ali postoje mnoge prepreke na putu teorijske analize algoritama. O tome više govorimo odmah u nastavku, a za sada samo skrećemo pažnju na činjenicu koju treba stalno imati na umu: složenost analize se savlađuje pojednostavljivanjem glavnih ciljeva. Zbog toga se, naravno, ne dobija apsolutno precizna ocena vremena izvršavanja, ali je ona dovoljno dobra da steknemo opštu sliku o efikasnosti algoritama.

Vreme izvršavanja nekog algoritma skoro uvek zavisi od broja ulaznih podataka koje treba obraditi. Tako, prirodno je očekivati da sortiranje 10 000 brojeva zahteva više vremena nego sortiranje samo 10 brojeva.

Vreme izvršavanja algoritma je dakle funkcija veličine ulaznih podataka. Algoritmi za različite probleme mogu imati različite numeričke forme veličine ulaza. Na primer, za problem sortiranja niza brojeva, veličina ulaza je broj članova niza koje treba sortirati; za množenje dva velika cela broja x i y, veličina ulaza je broj cifara broja x plus broj cifara broja y; za grafovske probleme, veličina ulaza je broj čvorova ili broj grana grafa. Srećom, forma veličine ulaza za konkretan problem je obično očigledna iz same prirode problema.

Kako bi se veličina ulaza obuhvatila analizom algoritma, najpre se ulazni podaci algoritma grupišu prema njihovom broju koji se predstavlja pozitivnim celim brojem n. Zatim se vreme izvršavanja algoritma za ulaz veličine n izražava pomoću neke funkcije $T(n)$. Drugim rečima, za svaki pozitivan ceo broj n, vrednost $T(n)$ daje broj vremenskih jedinica koliko traje izvršavanje algoritma za svaki njegov ulaz veličine n. Na slici 2.1 je ilustrovan ovaj pristup.



SLIKA 2.1: Vreme izvršavanja $T(n)$ algoritma A je funkcija veličine ulaznih podataka n.

Međutim, tačno vreme izvršavanja algoritma za neki konkretni ulaz, pored njegove veličine n, zavisi i od drugih faktora kao što su brzina procesora računara ili struktura samih ulaznih podataka. Zato da bismo pojednostavili analizu i da bi definicija funkcije $T(n)$ imala smisla, pretpostavljamo da se sve osnovne instrukcije algoritma izvršavaju za jednu vremensku jedinicu i uzimamo najgori slučaj ulaznih podataka. Onda da bismo dobili funkciju vremena izvršavanja $T(n)$ nekog algoritma treba da prosto prebrojimo osnovne instrukcije koje će se u algoritmu izvršiti *u najgorem*

slučaju. Preciznije, funkciju $T(n)$ za *vreme izvršavanja u najgorem slučaju* nekog algoritma definišemo da bude maksimalno vreme izvršavanja tog algoritma za sve ulaze veličine n. Ova funkcija se naziva i *vremenska složenost u najgorem slučaju*. Poštujući tradiciju da se najgori slučaj podrazumeva, ako nije drugačije naglašeno skoro uvek ćemo koristiti kraće termine *vreme izvršavanja* i *vremenska složenost* algoritma.

Jedinična instrukcija algoritma je osnovna računarska operacija čije je vreme izvršavanja konstantno. Ovo konstantno vreme može biti različito za različite vrste osnovnih instrukcija, ali je bitno da ono zavisi samo od modela računara na kome se podrazumeva izvršavanje osnovnih instrukcija, a ne i od veličine podataka u algoritmu.

U stvari, analiza algoritama se dalje pojednostavljuje time što se uzima da se svaka jedinična instrukcija izvršava za jednu vremensku jedinicu. Apsolutna vrednost vremenske jedinice nije bitna (da li mikrosekunda, nanosekunda, ...), jer se kasnije pokazuje da nas zapravo interesuje samo asimptotsko ponašanje vremena izvršavanja, odnosno njegova brzina rasta u obliku neke konstante pomnožene prostom funkcijom veličine ulaza. Prema tome, za analizu algoritma je važan samo ukupan broj jediničnih instrukcija koje se izvršavaju, a ne njihovo apsolutno ukupno vreme izvršavanja.

Tipične jedinične instrukcije algoritama su:

- dodela vrednosti promenljivoj;
- poređenje vrednosti dve promenljive;
- aritmetičke operacije;
- logičke operacije;
- ulazno/izlazne operacije.

Jedno upozorenje: ove instrukcije se ponekad ne mogu smatrati jediničnim instrukcijama u algoritmu. Na primer, neka aritmetička operacija koja izgleda naivno kao što je to na primer sabiranje, nije jedinična instrukcija ukoliko je veličina operanada aritmetičke operacije velika. U tom slučaju, vreme potrebno za njeno izvršavanje se povećava sa veličinom njenih operanada, pa se ne može ograničiti nekom konstantom nezavisnom od veličine podataka. Imajući ovo na umu, ipak ćemo prethodne instrukcije smatrati jediničnim sem ako to nije realistično za dati problem.

Prethodnu diskusiju o analizi algoritama možemo pretočiti u praktični postupak za određivanje vremenske složenosti algoritama ukoliko primetimo da je *najgori slučaj izvršavanja* algoritma onaj u kojem se izvršava *najveći broj* jediničnih instrukcija. Kao jednostavan primer, razmotrimo

A1:

```
1  n = 5;
2  repeat
3     read(m);
4     n = n - 1;
5  until ((m == 0) || (n == 0));
```

A2:

```
1  read(n);
2  repeat
3     read(m);
4     n = n - 1;
5  until ((m == 0) || (n == 0));
```

Mada se ovi fragmenti razlikuju samo u prvom redu, u fragmentu A1 se izvršava pet iteracija `repeat` petlje u najgorem slučaju, dok se u fragmentu A2 izvršava n iteracija iste petlje u najgorem slučaju. To sledi otuda što se izvršavanje `repeat` petlje završava kada se ispuni jedan od uslova (`m == 0`) ili (`n == 0`). Prvi uslov nikad neće biti ispunjen ukoliko promenljiva `m` ne dobije vrednost 0 u trećem redu, pre nego što se vrednost promenljive n umanji do 0 ponavljanjem naredbe u četvrtom redu. To je dakle najgori slučaj za broj iteracija `repeat` petlje, jer ispunjenost prvog uslova može samo da smanji taj broj iteracija.

Analiza najgoreg slučaja vremena izvršavanja algoritma je važna iz dva razloga. Prvo, time se dobijaju garantovane performanse algoritma, jer najgori slučaj određuje gornju granicu vremena izvršavanja za svaki skup ulaznih podataka date veličine. Drugo, najgori slučaj je mnogo lakši za analizu (često i jedino mogući) nego drugi kriterijumi efikasnosti algoritama.

Prebrojavanje broja jediničnih instrukcija u najgorem slučaju izvršavanja algoritma kako bi se dobilo vreme izvršavanja tog algoritma može se umnogome pojednostaviti ako znamo vremenske relacije tipičnih algoritamskih konstrukcija. Naime, vreme izvršavanja složenijih konstrukcija možemo izvesti na osnovu vremena izvršavanja njihovih sastavnih delova. U tabeli 2.1 su sumirana vremena izvršavanja osnovnih algoritamskih konstrukcija, pri čemu je sa T_B označeno vreme izvršavanja (u najgorem slučaju) bloka naredbi B. Tom tabelom nije obuhvaćena rekurzija, pošto će o njoj biti više reči u poglavlju 5.

U tabele 2.1 je očigledno da je vreme izvršavanja serije dva bloka naredbi jednako zbiru vremena izvršavanja pojedinih blokova. Isto tako, jasno je da je vreme izvršavanja alternative dva bloka naredbi jednako dužem vremenu izvršavanja jednog od ta dva bloka, jer se posmatra najgori slučaj izvršavanja.

Najvažnije pravilo u ovoj tabeli jeste da je vreme izvršavanja petlje jednako vremenu izvršavanja naredbi u telu petlje pomnoženim sa najve-

Konstrukcija	**Vreme izvršavanja**
Naredba serije S: $P; Q;$	$T_S = T_P + T_Q$
Naredba grananja S: **if** C **then** P; **else** Q;	$T_S = T_C + \max\{T_P, T_Q\}$
Naredba petlje S: (1) **while** C **do** P; (2) **repeat** P; **until** C; (3) **for** $i = j$ **to** k **do** P;	$T_S = n \cdot T_P$ n – najveći broj iteracija petlje

TABELA 2.1: Vremenska složenost osnovnih algoritamskih konstrukcija.

ćim mogućim brojem iteracija petlje. (Obratite pažnju na to da ovde radi jednostavnosti nismo uzeli u obzir vreme potrebno za proveru uslova prekida rada petlji.) Ovo se odnosi i na ugnježđene petlje: vreme izvršavanja naredbi unutar grupe ugnježđenih petlji jednako je vremenu izvršavanja naredbi u telu najunutrašnjije petlje pomnoženo sa proizvodom broja iteracija svih petlji. Na primer, posmatrajmo ovaj algoritamski fragment:

```
1  for i = 1 to n do
2     for j = 1 to n do
3        if (i < j) then
4           swap(a[i,j], a[j,i]);  // jedinična instrukcija
```

Pod pretpostavkom da je naredba `swap()` u četvrtom redu jedna od jediničnih instrukcija i uzimajući najgori slučaj da je uslov `if` naredbe uvek ispunjen, vreme izvršavanja ovog fragmenta je:

$$T(n) = n \cdot n \cdot (1+1) = 2n^2.$$

U ovom izrazu za funkciju $T(n)$ na desnoj strani imamo proizvod $n \cdot n$ pošto je broj iteracija spoljašnje i unutrašnje petlje jednak n, dok faktor $(1+1)$ obuhvata vreme za jedno izvršavanje tela unutrašnje petlje u redovima 3 i 4. Naime, to izvršavanje se sastoji od provere ispunjenosti uslova `if` naredbe i izvršavanja jedinične instrukcije `swap`.

Iako u ovoj knjizi koristimo isključivo vreme izvršavanja u najgorem slučaju, mogući su i drugi pristupi za određivanje efikasnosti algoritama.

Jedan od njih je probabilistička analiza srednjeg vremena izvršavanja nekog algoritma. *Prosečno vreme izvršavanja* je funkcija $T_p(n)$ koja daje srednju vrednost vremena izvršavanja algoritma za slučajne ulazne podatke veličine n. Prosečno vreme izvršavanja je realističnija mera performansi algoritma u praksi, ali je često to mnogo teže odrediti nego vreme izvršavanja u najgorem slučaju. Razlog za to je da moramo znati raspodelu verovatnoće slučajnih ulaznih podataka, što je teško zamislivo u praksi. Ali da bismo ipak mogli lakše izvesti matematička izračunavanja, obično pretpostavljamo da su svi ulazi veličine n jednako verovatni, što u nekom konkretnom slučaju možda i nije pravo stanje stvari.

Probabilističku analizu determinističkih algoritama treba razlikovati od analize takozvanih *randomiziranih algoritama*. Randomizirani algoritmi su po svojoj prirodi „slučajni" jer se u toku njihovog izvršavanja donose slučajne odluke. Njihovo vreme izvršavanja zavisi od slučajnih ishoda tih odluka i zato predstavlja slučajnu vrednost. Za randomizirane algoritme se određuje *očekivano vreme izvršavanja*, pri čemu se matematičko očekivanje izračunava u odnosu na slučajne odluke tokom izvršavanja.

Sasvim drugačiji pristup je ocena efikasnosti algoritma u odnosu na efikasnost drugih algoritama koji rešavaju isti problem. Ovaj metod se sastoji u tome da se formira mala kolekcija tipičnih ulaznih podataka koji se nazivaju *benčmark podaci*. Benčmark podaci se dakle smatraju reprezentativnim za sve ulazne podatke, odnosno podrazumeva se da će algoritam koji ima dobre performanse za benčmark podatke imati isto tako dobre performanse za *sve* ulazne podatke. Kako je naš cilj analitička ocena efikasnosti algoritama, a benčmark analiza je u suštini empirijski metod, o tome više nećemo govoriti u ovoj knjizi.

Na osnovu nađenog vremena izvršavanja nekog algoritma možemo pokušati da odgovorimo i na tri dodatna pitanja:

1. Kako možemo reći da li je neki algoritam dobar?
2. Kako znamo da li je neki algoritam optimalan?
3. Zašto nam je uopšte potreban efikasan algoritam — zar ne možemo prosto kupiti brži računar?

Da bismo odgovorili na prvo pitanje, lako možemo uporediti vreme izvršavanja posmatranog algoritma sa vremenima izvršavanja postojećih algoritama koji rešavaju isti problem. Ako to poređenje ide u prilog posmatranom algoritmu, možemo kvantitativno zaključiti da je on dobar algoritam za neki problem.

Na drugo pitanje je teško odgovoriti u opštem slučaju. To je zato što iako za bilo koja dva algoritma možemo reći koji je bolji ako uporedimo njihova vremena izvršavanja, ipak ne možemo reći da li je onaj bolji algoritam zaista i najbolji mogući. Drugim rečima, moramo nekako znati donju granicu vremena izvršavanja svih algoritama za konkretan problem. To znači da za dati problem ne samo poznati algoritmi, nego i oni koji još nisu otkriveni, moraju se izvršavati za vreme koje je veće od neke granice. Određivanje ovakvih dobrih granica nije lak zadatak i time se nećemo baviti u ovoj knjizi.

Treće pitanje obično postavljaju neuki ljudi koji misle se novcem može sve kupiti. Pored toga što su intelektualno izazovniji, efikasni algoritmi će se skoro uvek brže izvršavati na sporijem računaru nego loši algoritmi na super računaru za dovoljno velike ulazne podatke, u šta ćemo se i sami uveriti. U stvari, instance problema ne moraju da postanu toliko velike pre nego što brži algoritmi postanu superiorniji, čak i kada se izvršavaju na običnom personalnom računaru.

2.3 Primeri algoritama

Da bismo bolje razumeli suštinu dizajna i analize algoritama, u ovom odeljku ćemo prethodnu diskusiju o tome potkrepiti konkretnim, reprezentativnim primerima. Ti primeri za čitaoce imaju nesumnjivu praktičnu vrednost u sticanju neophodne rutine, ali oni imaju i pedagošku vrednost jer uspostavljaju osnove dizajna i analize algoritama na koje se oslanjaju ostali delovi ove knjige.

Primer: najmanji element niza

Problem najmanjeg elementa niza je sličan problemu najvećeg elementa niza iz odeljka 1.4: za dati niz a od n neuređenih brojeva $a_1, a_2, \ldots, a_n$ treba naći indeks (prvog) *najmanjeg* elementa tog niza. Algoritam `min` za ovaj problem je zato jednostavna modifikacija algoritma `max` kojim se nalazi indeks najvećeg elementa niza. Naime, za nalaženje globalno najmanjeg elementa niza, svi elementi niza se redom proveravaju da li su manji od najmanjeg elementa koji je nađen do odgovarajućeg trenutka provere.

```
// Ulaz: niz a, broj elemenata n niza a
// Izlaz: indeks najmanjeg elementa niza a
algorithm min(a, n)
```

```
m = a[1];  // najmanji element nađen do sada
j = 1;     // indeks najmanjeg elementa

i = 2;     // proveriti ostale elemente niza
while (i <= n) do
   if (m > a[i]) then  // nađen manji element od privremeno najmanjeg
      m = a[i];        // zapamtiti manji element
      j = i;           // ... i njegov indeks
   i = i + 1;  // preći na sledeći element niza

return j;  // vratiti indeks najmanjeg elementa
```

Analizu jednostavnog algoritma `min` nije teško izvršiti. Pošto broj iteracija tela `while` petlje iznosi $n-1$, a telo te pelje se sastoji od `if` naredbe i naredbe dodele, vreme izvršavanja algoritma `min` je:

$$\begin{aligned} T(n) &= 1+1+1+(n-1)(3+1)+1 \\ &= 4+4(n-1) \\ &= 4n. \end{aligned}$$

U ovom izrazu za funkciju $T(n)$ u prvom redu, prve tri jedinice u zbiru su vremena izvršavanja prve tri naredbe dodele u algoritmu `min`, faktor $(3+1)$ predstavlja vreme jednog izvršavanja tela `while` petlje (provera uslova i dve naredbe dodele u `if` naredbi, kao i naredba dodela na kraju tela petlje), a poslednjom jedinicom je obuhvaćeno vreme izvršavanja `return` naredbe.

U izrazu $4n$ za funkciju $T(n)$ vremena izvršavanja algoritma `min` se veličina ulaza n (tj. broj elemenata ulaznog niza) pojavljuje na prvi stepen. Zato za taj algoritam kažemo da je *linearni* algoritam. Primetimo još da je vreme izvršavanja algoritma `max` iz odeljka 1.4 takođe $T(n) = 4n$, jer su algoritmi `min` i `max` praktično isti.

Primer: maksimalna suma podniza

Problem maksimalne sume podniza se često pojavljuje u računarskoj biologiji prilikom analize DNA ili proteinskih sekvenci. Taj problem je računarski interesantan pošto postoje mnogi algoritmi različite efikasnosti za njegovo rešenje. U ovom odeljku ćemo opisati samo najsporiji, kvadratni algoritam za njegovo očigledno rešenje na osnovu definicije problema. Brži, linearni algoritmi zahtevaju dublji uvid u prirodu problema i složenije algoritamske tehnike o kojima se govori u nastavku knjige.

Ali pre definicije samog problema, potrebno je precizirati šta smatramo podnizom nekog niza. Za niz a od n brojeva $a_1, a_2, \ldots, a_n$, *neprekidni podniz* niza a je drugi niz koji se sastoji od nula ili više uzastopnih elemenata niza a. Na primer, ako je

$$a = [2, -1, 1, 3, -4, -6, 7, -2, 3, -5],$$

onda su $[1, 3, -4, -6, 7]$ i $[-1]$ neprekidni podnizovi, dok niz $[2, 1, 3]$ to nije. Pošto nas u ovom odeljku interesuju samo podnizovi koji se sastoje od uzastopnih elemenata, radi jednostavnosti izražavanja umesto termina „neprekidni podniz" često koristimo kao sinonim kraći izraz „podniz".

Ako je dat (neprekidni) podniz $a_i, \ldots, a_j$ niza a, suma tog podniza koju označavamo $S_{i,j}$ jeste običan zbir svih elemenata podniza:

$$S_{i,j} = a_i + a_{i+1} + \cdots + a_j = \sum_{k=i}^{j} a_k .$$

Obratite pažnju na to da se prazan niz dužine nula smatra neprekidnim podnizom svakog niza, pa po definiciji uzimamo da je suma praznog podniza jednaka nuli.

Problem maksimalne sume (neprekidnog) podniza sastoji se u tome da se odredi najveća suma svih neprekidnih podnizova datog niza. (Opštije, potrebno je odrediti i sâm neprekidni podniz koji ima najveću sumu.) Na primer, čitaoci mogu proveriti da rešenje problema maksimalne sume podniza za prethodni niz

$$a = [2, -1, 1, 3, -4, -6, 7, -2, 3, -5],$$

iznosi 8, što je zbir elemenata podniza $[7, -2, 3]$.

Elementi datog niza mogu biti pozitivni, negativni i nule. Ako dati niz sadrži samo pozitivne brojeve, rešenje problema maksimalne sume podniza takvog niza je očigledno zbir svih elemenata niza. Ako niz sadrži samo negativne brojeve, neprekidni podniz koji ima najveću sumu je prazan podniz i rešenje u ovom slučaju je zato uvek nula. Interesantan slučaj je dakle kada imamo zajedno pozitivne i negativne brojeve u datom nizu.

Za dati niz a od n brojeva, neka je M suma njegovog neprekidnog podniza koji ima najveću sumu. Formalno,

$$M = \max\{0, \max_{1 \le i \le j \le n} S_{i,j}\}.$$

Nula na desnoj strani ove jednakosti je potrebna zato što bez ove nule jednakost ne bi bila tačna kada se dati niz sastoji samo od negativnih elemenata. Ali radi jednostavnijeg pisanja, u daljem tekstu pretpostavljamo da implicitno uvek uzimamo 0 kao maksimalnu sumu u tom specijalnom slučaju. Tako, prethodnu jednakost pišemo sažetije:

$$M = \max_{1 \le i \le j \le n} S_{i,j}.$$

Da bismo izračunali M po definiciji, treba da izračunamo sume svih podnizova koji počinju u poziciji 1 datog niza, zatim treba da izračunamo sume svih podnizova koji počinju u poziciji 2 datog niza i tako dalje, sve dok ne izračunamo sume svih podnizova koji počinju u poziciji n datog niza. Na kraju, potrebno je da dodatno izračunamo maksimum svih ovih suma.

Koliko imamo ovih mogućih suma čiji maksimum tražimo? Lako je videti da broj svih podnizova koji počinju u poziciji 1 datog niza (i dakle njihovih suma) iznosi n, da broj svih podnizova koji počinju u poziciji 2 datog niza iznosi $n-1$ i tako dalje, da broj svih podnizova koji počinju u poziciji n datog niza iznosi 1. Prema tome, ukupan broj suma čiji maksimum tražimo je $n+(n-1)+\cdots+1 = n(n+1)/2$.

Primetimo da postupak izračunavanja M po definiciji možemo reorganizovati bez uticaja na konačan rezultat. Naime, umesto da računamo sve moguće sume i tek nakon toga tražimo njihov maksimum, možemo računati parcijalne maksimume suma elemenata od i-te pozicije niza i na kraju tražiti maksimum ovih parcijalnih maksimalnih suma.[2] Preciznije, ako za svako $i = 1,2,\ldots,n$ u i-tom koraku dodatno izračunamo parcijalni maksimum suma svih podnizova koji počinju u i-poziciji datog niza, onda na samom kraju preostaje samo da izračunamo maksimum ovih n parcijalnih maksimalnih suma. Ovo možemo predstaviti tako da najpre računamo drugi niz b od n elemenata $b_1, b_2, \ldots, b_n$, pri čemu je

$$b_i = \max\{S_{i,i}, S_{i,i+1}, \ldots, S_{i,n}\} = \max_{j=i,\ldots,n} S_{i,j},$$

a zatim računamo M kao

$$M = \max\{b_1, b_2, \ldots, b_n\} = \max_{i=1,\ldots,n} b_i.$$

Za algoritamsko izračunavanje M primenjujemo upravo ovaj pristup. Realizacija tog postupka na pseudo jeziku sastoji se od dva algoritma. Prvi,

[2] Ovom reorganizacijom izračunavanja se doduše ne ubrzava odgovarajući algoritam, ali se štedi na memoriji.

pomoćni algoritam računa element b_i niza b. Drugi, glavni algoritam koristi prvi algoritam kao potprogram za nalaženje rešenja problema maksimalne sume podniza datog niza a. Pomoćni algoritam je po strukturi sličan algoritmu `min` iz prethodnog odeljka:

```
// Ulaz:  indeks i, niz a, broj elemenata n niza a
// Izlaz: element b_i niza b
algorithm B(i, a, n)

   m = 0;     // maksimalna suma podniza od pozicije i
   s = 0;     // sume S_{i,j} za j=i,...,n

   for j = i to n do   // izračunati S_{i,j} za j=i,...,n
      s = s + a[j];    // S_{i,j} = S_{i,j-1} + a_j
      if (m < s) then  // nađena veća suma
         m = s;        // ažurirati maksimalnu sumu

   return m;  // vratiti b_i
```

Algoritam `B` izračunava element b_i na sledeći način. Promenljiva m sadrži privremeno izračunatu maksimalnu sumu podnizova koji počinju u poziciji i, a promenljiva s sadrži sume tih podnizova. Ove promenljive se najpre pravilno inicijalizuju vrednošću 0. Potom se u svakom koraku j od i do n izračunava suma $S_{i,j}$ u promenljivoj s dodavanjem elementa a_j prethodno izračunatoj sumi $S_{i,j-1}$. Odmah zatim se proverava da li je $m < s$, odnosno da li je pronađen podniz od pozicije i čija je suma veća od trenutno maksimalne. Naravno, ako je to slučaj, privremenu maksimalnu sumu treba ažurirati, što se i čini.

Da bismo analizirali vreme izvršavanja algoritma `B`, primetimo da ono zavisi od dva parametra: broja elemenata n niza a i indeksa i elementa b_i koji se izračunava. U vremenu izvršavanja ovog algoritma dominira `for` petlja čije se telo ponavlja $n-i+1$ puta, jer se brojač j ove petlje menja od i do n. Prema tome, vreme izvršavanja algoritma `B` je:

$$\begin{aligned} T(n,i) &= 1+1+(n-i+1)(1+2)+1 \\ &= 3+3(n-i+1) \\ &= 3(n-i+2). \end{aligned}$$

Sada je lako napisati algoritam kojim se nalazi maksimalna suma podniza datog niza: algoritmom `B` treba najpre redom izračunati sve elemente

b_i, a zatim prosto treba odrediti njihov maksimum. Ovaj postupak na pseudo jeziku je:[3]

```
// Ulaz:  niz a, broj elemenata n niza a
// Izlaz: maksimalna suma podniza M
algorithm mcss(a, n)

   for i = 1 to n do
      b[i] = B(i, a, n);
   M = max(b, n);

   return M;
```

S obzirom na to da znamo vremena izvršavanja algoritama `B` i `max`, lako je onda odrediti vreme izvršavanja algoritma `mcss`:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} \big(3(n-i+2)+1\big) + (4n+1) + 1 \\ &= 3\sum_{i=1}^{n}(n-i) + 7n + 4n + 2 \\ &= 3\frac{(n-1)n}{2} + 11n + 2 \\ &= 1.5n^2 + 9.5n + 2. \end{aligned}$$

Podsetimo se da funkcija $T(n)$ predstavlja vreme izvršavanja (u najgorem slučaju) nekog algoritma za sve ulazne podatke tog algoritma veličine n. Pošto se promenljiva n u prethodnom izrazu za funkciju $T(n)$ vremena izvršavanja algoritma `mcss` pojavljuje na drugi stepen, za taj algoritam kažemo da je *kvadratni* algoritam.

Primetimo da je moguće poboljšati ovaj algoritam ukoliko željeni maksimum M elemenata b_i izračunavamo „u hodu". Naime, umesto da maksimum elemenata b_i tražimo nakon što sve te elemente izračunamo, možemo da određujemo njihov privremeni maksimum kako ih redom izračunavamo. Pored toga, na ovaj način nam ne treba poseban niz za čuvanje elemenata b_i, pa zato u ovom slučaju štedimo na memorijskom prostoru.

U stvari, ovaj pristup možemo dalje poboljšati ukoliko određujemo privremeni maksimum ne od elemenata b_i već od suma $S_{i,j}$ kako ih redom izračunavamo. Postupak na pseudo jeziku za ovu „štedljivu" verziju algoritma `mcss` je:

[3] Ime algoritma `mcss` je skraćenica od engleskog termina za problem najveće sume neprekidnog podniza: *the maximum contiguous subsequence sum problem*.

```
// Ulaz:  niz a, broj elemenata n niza a
// Izlaz: maksimalna suma podniza M
algorithm mcss(a, n)

   M = 0;   // maksimalna suma podniza
   for i = 1 to n do   // izračunati b_i za i=i,...,n
      s = 0;
      for j = i to n do   // izračunati S_{i,j} za j=i,...,n
         s = s + a[j];    // S_{i,j} = S_{i,j-1} + a_j
         if (M < s) then
            M = s;

   return M;
```

Čitaocima se za vežbu ostavlja da se uvere da ova varijanta nije brža od prvobitne, odnosno da je i dalje reč o kvadratnom algoritmu.

Primer: sortiranje niza

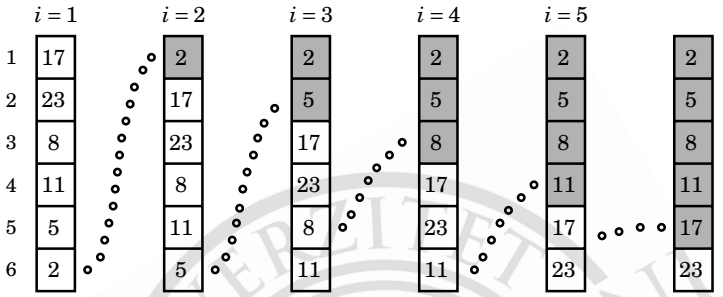
Ako je dat niz neuređenih brojeva, problem njegovog sortiranja se sastoji od preuređivanja brojeva tog niza tako da oni obrazuju rastući niz. Preciznije, za dati niz a od n elemenata $a_1, a_2, \ldots, a_n$ treba naći permutaciju svih indeksa elemenata niza $i_1, i_2, \ldots, i_n$ tako da novi prvi element a_{i_1}, novi drugi element a_{i_2} i tako dalje, novi n-ti element a_{i_n} u nizu zadovoljavaju uslov $a_{i_1} \leq a_{i_2} \leq \cdots \leq a_{i_n}$.

Za rešavanje problema sortiranja niza ćemo pokazati tri jednostavna algoritma. To su klasični algoritmi koji približno imaju isto vreme izvršavanja koje kvadratno zavisi od veličine ulaza (tj. broja elemenata niza). U nastavku knjige ćemo upoznati efikasnije metode sortiranja.

Sortiranje zamenjivanjem (*bubble-sort*). Najbolji način za razumevanje osnovne ideje prvog metoda sortiranja, koji se popularno naziva *sortiranje zamenjivanjem* (engl. *bubble-sort*), jeste da zamislimo da je dati niz potopljen u vodu i okrenut vertikalno. Ako dalje zamislimo da su elementi datog niza sa malim vrednostima „laki" po težini, onda će oni ovim metodom „isplivavati" na vrh proizvodeći „mehuriće". Ova analogija je upravo razlog za (interesantan?) engleski naziv ovog metoda.

Postupak sortiranja zamenjivanjem se sastoji od višestrukih prolaza duž datog niza od dna ka vrhu. Pri tome, ako dva susedna elementa nisu u dobrom redosledu („lakši" element je ispod „težeg"), zamenjujemo im mesta. Efekat ovog zamenjivanja je da se u prvom prolazu „najlakši"

element sa najmanjom vrednošću penje sve do vrha. U drugom prolazu, drugi najmanji element se penje do druge pozicije. Slično, treći najmanji element u trećem prolazu dolazi na svoje mesto i tako dalje. Na slici 2.2 je prikazan ovaj metod sortiranja za niz $[17, 23, 8, 11, 5, 2]$ od 6 elemenata. Na toj slici vrednost promenljive i ukazuje na redni broj prolaza.



SLIKA 2.2: Sortiranje zamenjivanjem.

Primetimo sa slike 2.2 da u drugom prolazu možemo da stanemo do druge pozicije, jer se najmanji element već nalazi na svom prvom mestu. Isto tako, u trećem prolazu možemo da stanemo do treće pozicije, pošto znamo da se prva dva najmanja elementa već nalaze u svojim završnim pozicijama. Sledstveno tome, u i-tom prolazu možemo da stanemo do pozicije i pošto se $i-1$ najmanjih elemenata već nalaze u svojim završnim pozicijama.

Precizniji opis ovog postupka na pseudo jeziku za sortiranje niza zamenjivanjem njegovih elemenata je:

```
// Ulaz:   niz a, broj elemenata n niza a
// Izlaz:  niz a sortiran u rastućem redosledu
algorithm bubble-sort(a, n)

   for i = 1 to n-1 do
      for j = n downto i+1 do
         if (a[j] < a[j-1]) then
            swap(a[j], a[j-1]);

   return a;
```

Da bismo analizirali vreme izvršavanja algoritma `bubble-sort`, najpre primetimo da algoritam `swap` na strani 13 možemo smatrati jediničnom instrukcijom. To ima smisla za algoritam `swap` pošto se njegove četiri

naredbe izvršavaju za ukupno konstantno vreme.[4]

Vreme jednog izvršavanja tela unutrašnje `for` petlje je dve vremenske jedinice: jedna za proveru uslova `if` naredbe i jedna za instrukciju `swap`). Stoga preostaje još da odredimo koliko puta će se to telo izvršiti u algoritmu `bubble-sort`. Čitaoci mogu lako proveriti da za fiksnu vrednost brojača $i = 1, 2, \ldots, n-1$ spoljašnje `for` petlje, telo unutrašnje `for` petlje će se izvršiti tačno $n-i$ puta. Zbog toga ukupan broj izvršavanja tela unutrašnje petlje iznosi:

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

Prema tome, ako uzmemo u obzir i naredbu `return` na kraju, ukupno vreme izvršavanja algoritma `bubble-sort` je:

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

Algoritam `bubble-sort` je dakle kvadratni algoritam pošto vreme njegovog izvršavanja $T(n)=n^2-n+1$ zavisi od veličine ulaza n na drugi stepen.

Vreme izvršavanja nekog algoritma posmatramo u najgorem slučaju, pa je to najduže vreme koje će taj algoritam raditi za datu veličinu ulaznih podataka. Primetimo da najgori slučaj algoritma `bubble-sort` (kada je ulazni niz a početno sortiran u opadajućem redosledu) nije mnogo gori od najboljeg slučaja (kada je ulazni niz a početno već sortiran u rastućem redosledu). U najboljem slučaju se nikad ne izvršava operacija `swap` u unutrašnjoj `for` petlji, ali je vreme izvršavanja u tom slučaju ipak jednako

$$\frac{(n-1)n}{2}\cdot 1+1=0.5n^2-0.5n+1,$$

što je i dalje kvadratna funkcija od n.

Ovo svojstvo algoritma `bubble-sort` ne važi u opštem slučaju, jer se vremena izvršavanja nekih algoritama u najgorem i najboljem slučaju mogu drastično razlikovati. To je na neki način dobra osobina algoritma, jer je prirodno očekivati da u najboljem slučaju ulaznih podataka algoritam radi mnogo brže.

Prethodno opisani algoritam `bubble-sort` možemo lako modifikovati kako bi za „dobre" ulazne podatke taj algoritam radio mnogo brže. Primetimo da dodatni prolazi duž datog niza nisu više potrebni ukoliko se

[4] Ovo je jedan od načina za pojednostavljivanje analize algoritama: svaki algoritam koji se izvršava za konstantno vreme smatramo jediničnom instrukcijom, jer tačna vrednost tog konstantnog vremena nije bitna.

u poslednjem prolazu nijedanput nije izvršila operacija swap. To sledi otuda što onda ne postoji element niza koji je van svog mesta u rastućem redosledu, odnosno onda se svi elementi niza nalaze na svojim mestima u rastućem redosledu.

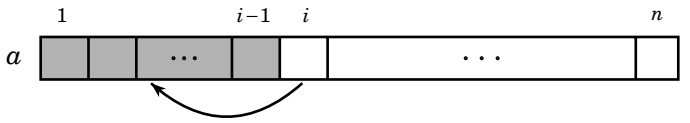
Ovo zapažanje se može lako iskoristiti u algoritmu bubble-sort: kako se u tom algoritmu prolazi duž datog niza realizuju spoljašnjom for petljom, potrebno ju je prosto zameniti drugom petljom čije se izvršavanje kontroliše indikatorom da li se u poslednjem prolazu nijedanput nije izvršila operacija swap. Ova poboljšana varijanta algoritma je:

```
// Ulaz:   niz a, broj elemenata n niza a
// Izlaz:  niz a sortiran u rastućem redosledu
algorithm bubble-sort(a, n)

   repeat
      s = false;  // indikator izvršavanja operacije swap
      i = 1;
      for j = n downto i+1 do
         if (a[j] < a[j-1]) then
            swap(a[j], a[j-1]);
            s = true;
      i = i + 1;
   until (s == false);

   return a;
```

Čitaocima se za vežbu ostavlja da se uvere da je u najgorem slučaju poboljšana verzija algoritma bubble-sort i dalje kvadratni algoritam, ali u najboljem slučaju je to linearni algoritam. Ovde je interesantno napomenuti da je ovaj način jedan od najbržih metoda sortiranja u slučaju kada se unapred zna da je samo mali broj elemenata van svog mesta u rastućem redosledu (recimo, kada se već sortiranom nizu dodaju dva-tri elementa).

Sortiranje umetanjem (*insert-sort*). Drugi metod za sortiranje niza a od n elemenata $a_1, a_2, \ldots, a_n$ popularno se naziva *sortiranje umetanjem* (engl. *insert-sort*). Ovaj metod se sastoji od n iteracija, pri čemu se u i-toj iteraciji umeće element a_i na njegovo pravo mesto između prvih $i-1$ najmanjih elemenata prethodno već uređenih u rastućem redosledu. Ovaj način sortiranja u i-toj iteraciji je ilustrovan na slici 2.3.

Da bi postupak sortiranja umetanjem bio jasniji na pseudo jeziku, pogodno je uvesti nulti element niza a_0 čija se vrednost pretpostavlja da je manja od svih vrednosti pravih elemenata $a_1, a_2, \ldots, a_n$. (Takva vrednost se obično naziva *sentinela*.)



SLIKA 2.3: Sortiranje umetanjem.

```
// Ulaz:   niz a, broj elemenata n niza a
// Izlaz:  niz a sortiran u rastućem redosledu
algorithm insert-sort(a, n)

   a[0] = -∞;
   for i = 1 to n do
      j = i
      while (a[j] < a[j-1]) do
         swap(a[j], a[j-1]);
         j = j - 1;

   return a;
```

Vreme izvršavanja algoritma `insert-sort` se određuje na sličan način kao za algoritam `bubble-sort`. Doduše, broj iteracija ugnježđene `while` petlje nije odmah očigledan, ali možemo primetiti da taj broj nije veći od $i-1$. To je zato što promenljiva j dobija početnu vrednost jednaku vrednosti promenljive i pre početka `while` petlje, a j se smanjuje za jedan u svakoj iteraciji te petlje do završetka rada petlje. A `while` petlja se sigurno završava kada je $j = 1$, jer je onda $a_{j-1} = a_0 = -\infty$, čime uslov nastavka te petlje postaje netačan. U najgorem slučaju dakle, iteracije `while` petlja se izvršavaju za sve uzastopne vrednosti j od i do 2, pa je broj tih iteracija najviše $i-1$.

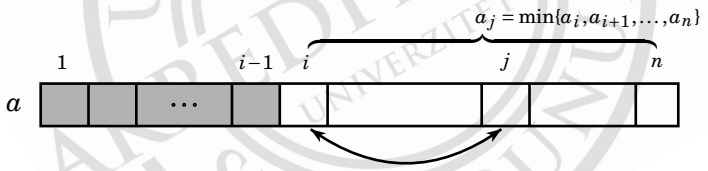
Prema tome, u algoritmu `insert-sort` se izvršava n iteracija spoljašnje `for` petlje, a u svakoj od tih iteracija za $i = 1,2,\ldots,n$ izvršava se najviše $i-1$ iteracija unutrašnje `while` petlje. Pošto se telo unutrašnje `while` petlje izvršava za 2 vremenske jedinice, ukupno vreme izvršavanja algoritma `insert-sort` je:

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

$$= n^2 + 2.$$

Algoritam `insert-sort` je dakle kvadratni algoritam. Primetimo da kada je ulazni niz početno sortiran u opadajućem redosledu, onda je broj iteracija unutrašnje `while` petlje jednak tačno $i-1$. Zato se u tom najgorem slučaju algoritam `insert-sort` zaista izvršava za kvadratno vreme dato funkcijom $T(n)$. Sa druge strane, u najboljem slučaju kada je ulazni niz početno sortiran u rastućem redosledu, broj iteracija unutrašnje `while` petlje je 0. Čitaoci mogu lako proveriti da je onda vreme izvršavanja algoritma `insert-sort` jednako $2n+2$, što je mnogo bolje nego u najgorem slučaju.

Sortiranje izborom (*select-sort*). Metod sortiranja niza a od n elemenata $a_1, a_2, \ldots, a_n$, koji se popularno naziva *sortiranje izborom* (engl. *select-sort*), zasniva se kao i prethodni metodi na ponavljanju određenog postupka nad datim nizom. Taj postupak u ovom metodu je da se u i-toj iteraciji uzima najmanji element među elementima $a_i, a_{i+1}, \ldots, a_n$ i zamenjuje sa elementom a_i. Ovaj način sortiranja u i-toj iteraciji je ilustrovan na slici 2.4.



SLIKA 2.4: Sortiranje izborom.

Ispravnost postupka za sortiranje izborom sledi na osnovu toga što posle prve iteracije će najmanji element niza biti na prvom mestu, posle druge iteracije će drugi najmanji element niza biti na drugom mestu i tako dalje, posle i-te iteracije će i najmanjih elemenata niza biti poređano u rastućem redosledu. Na kraju, posle $(n-1)$-ve iteracije će ceo niz biti poređan u rastućem redosledu.

Precizniji opis celokupnog postupka za sortiranje izborom na pseudo jeziku je:

```
// Ulaz:   niz a, broj elemenata n niza a
// Izlaz:  niz a sortiran u rastućem redosledu
algorithm select-sort(a, n)

   for i = 1 to n-1 do
```

```
    j = minr(a, i, n);
    swap(a[i], a[j]);

return a;
```

Algoritam `minr`, koji se u algoritmu `select-sort` koristi kao potprogram, kao rezultat daje indeks najmanjeg elementa „desnog" podniza a_i, $a_{i+1}, \ldots, a_n$ datog niza a. Taj algoritam je mala modifikacija algoritma `min` na strani 28 i zato se prepušta čitaocima za vežbu. Čitaoci mogu usput lako proveriti da vreme izvršavanja algoritma `minr` iznosi $4(n-i+2)$.

Da bismo sada analizirali algoritam `select-sort`, primetimo da se telo `for` petlje u tom algoritmu izvršava za vreme $4(n-i+2)+1+1$, gde su dvema jedinicama obuhvaćena vremena za naredbu dodele i operaciju `swap` koju smatramo jediničnom instrukcijom. Prema tome, ukupno vreme izvršavanja algoritma `select-sort` je:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n-1} \big(4(n-i+2)+2\big)+1 \\ &= \sum_{i=1}^{n-1} \big(4(n-i)+10\big)+1 \\ &= 4\frac{(n-1)n}{2} + 10(n-1)+1 \\ &= 2n^2+8n-9\,. \end{aligned}$$

I algoritam `select-sort` je dakle kvadratni algoritam, a čitaocima se ostavlja da analiziraju njegov najgori i najbolji slučaj.

Zadaci

1. Koristeći odgovarajuću invarijantu petlje, pokažite ispravnost algoritma za problem pretrage niza u 1. zadatku prethodnog poglavlja. Pored toga, odredite vreme izvršavanja tog algoritma.

2. Koristeći odgovarajuću invarijantu petlje, pokažite ispravnost algoritma za problem preostalog dela kredita u 2. zadatku prethodnog poglavlja. (*Savet:* Iznos kredita koji preostaje nakon i-te iteracija petlje je p_i.) Pored toga, odredite vreme izvršavanja tog algoritma.

3. Odredite vreme izvršavanja sledećih algoritamskih fragmenata:

a)
```
if (x == 0) then
   for i = 1 to n do
      a[i] = i;
```

b)
```
i = 1;
repeat
   a[i] = b[i] + c[i];
   i = i + 1;
until (i == n);
```

c)
```
if (x == 0) then
   for i = 1 to n do
      for j = 1 to n do
         a[i,j] = 0;
else
   for i = 1 to n do
      a[i,i] = 1;
```

d)
```
for i = 1 to n do
   for j = 1 to n do
      for k = 1 to j do
         if (i == j) then
            a[k,k] = 0;
```

4. Ako niz brojeva predstavimo u „prirodnom" horizontalnom položaju, postupak sortiranja zamenjivanjem (*bubble-sort*) na strani 34 sastoji se od ponavljanja prolaza duž datog niza od najvećeg potrebnog indeksa niza do najmanjeg, odnosno ti prolazi uvek idu u smeru zdesna na levo.

a) Napišite algoritam za sortiranja zamenjivanjem u kojem se duž datog niza uvek prolazi u smeru sleva na desno.

b) Napišite algoritam za sortiranja zamenjivanjem u kojem se duž datog niza naizmenično prolazi u oba smera.

UNIVERZITET
PRIVATNI
AKREDITOVANI
UNIVERZITET
SINGIDUNUM

GLAVA

3

Vreme izvršavanja algoritma

Analiza jednostavnih algoritama u prethodnom poglavlju pokazuje da za određivanje vremena izvršavanja algoritma moramo voditi računa o mnogim „sitnim" koracima u algoritmu. Kod iole složenijih algoritama bi nas brojanje svake jedinične instrukcije svakako omelo da steknemo pravu sliku o brzini izvršavanja algoritama, ako ne i sprečilo da do kraja ispravno odredimo vreme izvršavanja. Dobar uvid u složenost algoritama zasniva se na pravoj meri pojednostavljenja njihove analize.

Zato da ne bismo otišli u drugu krajnost i bili *suviše* precizni, analiza algoritama se pojednostavljuje do krajnih granica kako bi se pravilno ocenilo *asimptotsko* vreme izvršavanja algoritma, odnosno njegovo vreme izvršavanja samo za veliki broj ulaznih podataka. To drugim rečima možemo kazati da nas zapravo ne interesuje apsolutno tačno vreme izvršavanja nekog algoritma, nego samo to koliko brzo raste vreme izvršavanja algoritma sa povećanjem veličine ulaza algoritma.

U ovom poglavlju ćemo bliže razraditi ovu ideju dajući njenu motivaciju i formalizaciju.

3.1 Asimptotsko vreme izvršavanja

Prilikom analize algoritama možemo dobiti vrlo komplikovane funkcije za njihova vremena izvršavanja. Na primer, u prethodnom poglavlju smo za tri algoritma sortiranja dobili tri različite, relativno proste kvadratne

funkcije od veličine ulaza n. Ali sasvim je moguće i da za neki algoritam dobijemo vreme izvršavanja izraženo funkcijom, recimo,

$$T_1(n) = 10n^3 + n^2 + 40n + 80\,.$$

Ili slično, za neki drugi algoritam možemo dobiti da je njegovo vreme izvršavanja dato funkcijom

$$T_2(n) = 17n\log n - 23n - 10\,.$$

Ako zamislimo još komplikovaniji izraz za funkciju vremena izvršavanja, postavlja se pitanje da li nam složenost tog izraza zamagljuje sliku o odgovarajućem algoritmu? Na primer, da li na osnovu funkcija $T_1(n)$ i $T_2(n)$ možemo lako reći koji je od dva algoritma brži? Za odgovor na ovo pitanje možemo naravno uvek tačno nacrtati uporedo funkcije $T_1(n)$ i $T_2(n)$ i videti za razne vrednosti argumenta n koja od funkcija ima manje vrednosti, odnosno koji je algoritam brži.

Crtanje i izračunavanje složenih matematičkih funkcija nije jednostavno (kao što svi znamo, zar ne?), ali srećom to nije ni potrebno u analizi algoritama. To je zato što analizu algoritama težimo da pojednostavimo što više možemo i tražimo zapravo samo red veličine funkcije vremena izvršavanja algoritma za velike vrednosti njenog argumenta. Na primer, ako je n zaista veliko (zamislimo da je n jedan milion ili jedna milijarda), onda su vrednosti funkcija $T_1(n)$ i n^3 približno jednake. Isto tako, za veliko n, funkcije $T_2(n)$ i $n\log n$ imaju slične vrednosti. Zato kažemo za funkciju $T_1(n)$ da je reda veličine n^3 i za funkciju $T_2(n)$ da je reda veličine $n\log n$,[1] podrazumevajući da je n veliko.

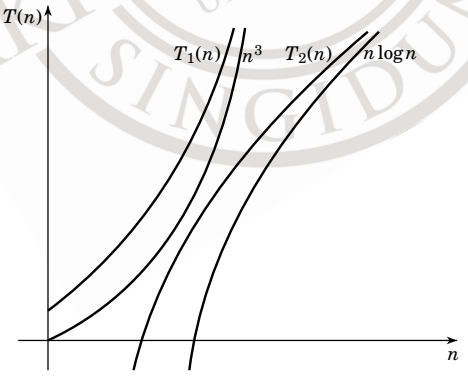
Prema tome, za vreme izvršavanja nekog algoritma uzimamo *jednostavnu* funkciju koja za velike vrednosti njenog argumenta najbolje aproksimira tačnu funkciju vremena izvršavanja tog algoritma. To približno vreme izvršavanja se naziva *asimptotsko vreme izvršavanja*. Primetimo da je ono, u stvari, mera brzine rasta tačnog vremena izvršavanja algoritma sa povećanjem broja njegovih ulaznih podataka. U daljem tekstu ćemo podrazumevati asimptotsko vreme izvršavanja algoritama, ali radi kratkoće izražavanja često izostavljamo pridev „asimptotsko".

Pojednostavljivanje cilja analize algoritama, odnosno određivanje što jednostavnijeg asimptotskog vremena izvršavanja, opravdano je iz tri ra-

[1] U računarstvu se logaritam skoro uvek uzima za osnovu 2, a ne za osnovu 10 ili e. U skladu sa ovom tradicijom, u knjizi nećemo pisati osnovu za logaritam podrazumevajući da je ona 2 ukoliko to nije drugačije naglašeno.

zloga. Prvo, brzina algoritama je naročito bitna kada je broj ulaznih podataka veliki, jer se vreme izvršavanja (skoro) svakog algoritma povećava sa brojem ulaznih podataka. Ako za isti problem imamo više algoritama i treba da odlučimo koji od njih da primenimo za njegovo rešavanje, onda je važno za koje vreme oni rešavaju taj problem kada je broj ulaznih podataka vrlo veliki. Naime, obično je svaki algoritam dovoljno efikasan za mali broj ulaznih podataka. To jest, uglavnom je irelevantno da li neki algoritam radi, recimo, jednu milisekundu ili pet mikrosekundi za desetak ulaznih podataka. Ta dva vremena su ionako neprimetna za ljude, ali to da li algoritam radi pet minuta ili tri meseca za milion ulaznih podataka jeste svakako značajno za odluku da li ćemo ga primeniti za rešavanje datog problema.

Drugo, za dovoljno velike vrednosti argumenta, vrednost funkcije vremena izvršavanja je preovlađujuće određena njenim dominantnim termom. Na slici 3.1 je ovo ilustrovano graficima parova funkcija $T_1(n)$ i n^3, kao i $T_2(n)$ i $n\log n$. Sa slike se možemo uveriti da su krive za $T_1(n)$ i n^3, kao i za $T_2(n)$ i $n\log n$, vrlo blizu jedna drugoj za velike vrednosti n. To znači da ukoliko govorimo o ponašanju tih funkcija za veliko n, možemo smatrati da su njihove vrednosti skoro jednake, odnosno da asimptotsko vreme izvršavanja dva algoritma iz prethodnog primera možemo jednostavnije izraziti pomoću $T_1(n) \approx n^3$ i $T_2(n) \approx n\log n$.



SLIKA 3.1: Grafici dva para približnih funkcija.

Treće, jednostavna funkcija za vreme izvršavanja se uzima bez konstanti, jer tačna vrednost konstante uz dominantni term nije značajna za računare različite snage (mada relativna vrednost tih konstanti može biti važna za funkcije koje rastu istom brzinom). Te konstante odražavaju uzetu jedinicu mere za vreme izvršavanja jediničnih operacija, ali to vreme

je svakako različito za, na primer, personalne i super računare.

Da bismo ovo bolje razumeli, pretpostavimo da su $A1$, $A2$ i $A3$ tri algoritma čija su vremena izvršavanja data funkcijama 2^n, $5n^2$ i $100n$, tim redom, gde je n veličina ulaza. Vrednosti ovih funkcija za različite vrednosti argumenta n su date u levoj tabeli na slici 3.2. Sa druge strane, ako se svaki algoritam izvršava na računaru koji obavlja jedan milion (10^6) instrukcija u sekundi, tada će se svaki algoritam izvršavati u stvarnom vremenu čije su vrednosti date u desnoj tabeli na istoj slici.

n	2^n	$5n^2$	$100n$
1	2	5	100
10	1024	500	1000
100	2^{100}	50000	10000
1000	2^{1000}	$5 \cdot 10^6$	100000

n	$A1$	$A2$	$A3$
1	$1\mu s$	$5\mu s$	$100\mu s$
10	$1ms$	$0{,}5ms$	$1ms$
100	$2^{70}g$	$0{,}05s$	$0{,}01s$
1000	$2^{970}g$	$5s$	$0{,}1s$

SLIKA 3.2: Vremena izvršavanja tri algoritma.

Vrednosti stvarnih vremena izvršavanja algoritama u desnoj tabeli na slici 3.2 jasno pokazuju da je za veliko n (recimo, $n > 100$) algoritam $A1$ najsporiji, zatim algoritam $A2$ srednji, a algoritam $A3$ najbrži. Dakle, oblici jednostavnih funkcija od n (tj. 2^n, n^2 i n) su značajniji od tačnih konstantnih faktora (tj. 1, 5 i 100) tih funkcija. Zato da bismo što jednostavnije i tačnije izrazili asimptotska vremena izvšavanja algoritama $A1$, $A2$ i $A3$, dovoljno je uzeti da su njihova vremena izvršavanja, redom, $T_1(n) = 2^n$, $T_2(n) = n^2$ i $T_3(n) = n$.

Pojednostavljena analiza algoritama

Asimptotsko vreme izvršavanja algoritma *ne* određuje se, naravno, tako što se odredi njegovo tačno vreme izvršavanja, pa se onda dobijena funkcija uprošćava uzimanjem dominantnog terma koji najbolje opisuje njen red veličine. Podsetimo se da smo upravo hteli da izbegnemo određivanje tačnog vremena izvršavanja i uzimanje u obzir „nevažnih" jediničnih instrukcija koje mnogo ne doprinose vremenu izvršavanja algoritma za veliki broj ulaznih podataka.

Analiza algoritama se zapravo sprovodi na sličan način kao i ranije, ali se pri tome pojednostavljuje sve ono što ne utiče bitno na određivanje reda veličine vremena izvršavanja algoritma. To znači da se i dalje vo-

dimo tabelom vremenske složenosti osnovnih algoritamskih konstrukcija na strani 26, ali uz zanemarivanje nedominantnih termova dobijenih u izrazu za funkciju vremena izvršavanja koji ne utiču na njen red veličine za velike vrednosti njenog argumenta. Mada se ovaj pristup može tačnije i formalnije opisati uvođenjem matematičkih pojmova (o čemu govorimo u odeljku 3.2 u nastavku), njegovu suštinu je najbolje razumeti kroz primere.

Kao prvi primer, razmotrimo ovaj algoritamski fragment (koji matricu a dimenzije $n \times n$ inicijalizuje da bude jedinična matrica):

```
1  for i = 1 to n do
2     for j = 1 to n do
3        a[i,j] = 0;
4  for i = 1 to n do
5     a[i,i] = 1;
```

Vreme izvršavanja $T(n)$ ovog fragmenta jednako je zbiru vremena izvršavanja dve petlje: prve `for` petlje u prvom redu i druge `for` petlje u četvrtom redu. Kako prva `for` petlja sadrži ugnježđenu `for` petlju u drugom redu, isto kao ranije moramo najpre odrediti broj iteracija ovih petlji i vreme jednog izvršavanja tela unutrašnje petlje, a zatim pomnožiti ove tri vrednosti.

Pošto se telo unutrašnje petlje u trećem redu izvršava za konstantno vreme, a broj iteracija spoljašnje i unutrašnje petlje jednak je n, ukupno vreme izvršavanja `for` petlje u prvom redu je neka konstanta pomnožena sa n^2. Ovu konstantu možemo međutim zanemariti kada je n veliko, pa zaključujemo da je vreme izvršavanja prve `for` petlje reda n^2.

Slično, vreme izvršavanja druge `for` petlje u četvrtom redu je reda n, jer je broj njenih iteracija jednak n i njeno telo u petom redu se izvršava za konstantno vreme. Prema tome, ukupno vreme izvršavanja $T(n)$ datog fragmenta je reda n^2+n, ali to možemo dalje uprostiti i zaključiti da je $T(n)$ reda veličine n^2, pošto funkcija n^2 dominira funkcijom n za veliko n.

Kao drugi primer razmotrimo algoritamski fragment koji ima istu funkcionalnost kao prethodni primer (tj. matrica a dimenzije $n \times n$ se inicijalizuje da bude jedinična matrica):

```
1  for i = 1 to n do
2     for j = 1 to n do
3        if (i == j) then
4           a[i,j] = 1;
5        else
6           a[i,j] = 0;
```

Vreme izvršavanja $T(n)$ ovog fragmenta jednako je vremenu izvršavanja dve ugnježđene `for` petlje u prvom i drugom redu. Kako se telo unutrašnje petlje sastoji samo od `if` naredbe u redovima 3–6, a nije se teško uveriti da se ta naredba izvršava za konstantno vreme, to znači da se telo unutrašnje petlje izvršava za konstantno vreme. Kako je dalje broj iteracija spoljašnje i unutrašnje petlje jednak n, vreme izvršavanja dve ugnježđene `for` petlje je neka konstanta pomnožena sa n^2. Prema tome, pošto ovu konstantu možemo zanemariti kada je n veliko, ukupno vreme izvršavanja $T(n)$ datog fragmenta je reda n^2.

Obratite pažnju na to da smo u prethodnim primerima za određivanje asimptotskog vremena izvršavanja često koristili fraze „vreme izvršavanja je reda veličine", „konstanta se može zanemariti" i „jedna funkcija dominira drugom funkcijom". To nisu jedine fraze u opštem slučaju koje se stalno ponavljaju, zbog čega se uvodi specijalna *asimptotska notacija* kojom se izražavanje olakšava i dodatno precizira.

Jedan oblik ove notacije je takozvani O-zapis, pa bismo recimo za prethodna dva algoritamska fragmenta kratko rekli da je njihovo vreme izvršavanja $T(n) = O(n^2)$ (ovo se izgovara: veliko o od n^2). Ovaj sažet zapis obuhvata ne samo to da je njihovo vreme izvršavanja reda veličine n^2 za veliko n, nego i tačno određuje šta podrazumevamo pod time kada kažemo da je jedna funkcija reda veličine druge funkcije. O ovome ćemo mnogo detaljnije govoriti u odeljku 3.2 u nastavku.

Primer: binarna pretraga sortiranog niza

Opšti problem pretrage niza (videti 1. zadatak na strani 15) je da, za dati niz a od n neuređenih brojeva $a_1, a_2, \ldots, a_n$ i dati broj x, treba odrediti da li se broj x nalazi u nizu a. Ukoliko je to slučaj, rezultat treba da bude indeks niza i takav da je $x = a_i$; u suprotnom slučaju, rezultat treba da bude 0.

U ovom primeru posmatramo specijalni slučaj opšteg problema pretrage kada se dati niz a ne sastoji od neuređenih brojeva, već je taj niz brojeva sortiran u rastućem redosledu: $a_1 \leq a_2 \leq \cdots \leq a_n$. Ovu pretpostavku da je niz sortiran možemo iskoristiti da bismo pretragu broja x u nizu znatno ubrzali.

Metod koji se primenjuje u specijalnom slučaju sortiranog niza naziva se *binarna pretraga* i sličan je načinu na koji se traži neka osoba u debelom telefonskom imeniku. Naime, da bismo dato ime i prezime neke osobe pronašli u imeniku, najpre bismo imenik otvorili negde na sredini imenika

i tu pokušali da pronađemo željenu osobu. Ako je tu nema, onda bismo prezime tražene osobe uporedili sa prezimenima osoba na toj srednjoj stranici. Ukoliko je prvo slovo prezimena tražene osobe po azbučnom redu *iza* prvog slova prezimena osoba na srednjoj stranici imenika, pretragu bismo usmerili na drugu polovinu imenika. Naime, tražena osoba se sigurno ne može nalaziti u prvoj polovini imenika pošto su osobe u imeniku poređane po azbučnom redu njihovih prezimena (tj. sortirane su u rastućem redosledu). Naravno, u drugom slučaju kada je prvo slovo prezimena tražene osobe po azbučnom redu *ispred* prvog slova prezimena osoba na srednjoj stranici imenika, pretragu bismo usmerili na prvu polovinu imenika.

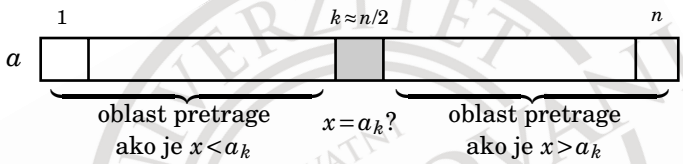
U drugom koraku bismo pretragu nastavili u odgovarajućoj polovini imenika ponavljanjem sličnog postupka. Negde na polovini odgovarajuće polovine bismo pokušali da pronađemo željenu osobu, pa ako je tu nema, pretragu bismo usmerili na jednu od polovina te polovine (tj. četvrtinu celog imenika) u kojoj se mora nalaziti tražena osoba zavisno od prvog slova njenog prezimena. U sledećem koraku bismo pretragu usmerili na odgovarajuću polovinu prethodne četvrtine imenika (tj. osminu celog imenika) i tako dalje. Na ovaj način, u svakom koraku bismo za faktor dva smanjivali dužinu prethodnog dela imenika u kojem se može nalaziti tražena osoba.

Prema tome, ponavljanjem ovih koraka moguća su dva ishoda: ili se u nekom koraku pronalazi željena osoba na njenoj stranici imenika, ili se na kraju dužina dela imenika u kome je moguće naći traženu osobu svodi na nulu. Naravno, u prvom slučaju je jasno da se tražena osoba nalazi u imeniku, dok u drugom slučaju možemo zaključiti da se ona ne nalazi u imeniku.

Ovaj duži opis ideje binarne pretrage je, nadamo se, dovoljan za lako razumevanje sličnog postupka za traženje broja x u sortiranom nizu a:

```
// Ulaz:   sortiran niz a, broj elemenata n niza a, traženi broj x
// Izlaz:  k takvo da x=a_k ili 0 ako se x ne nalazi u nizu a
algorithm bin-search(a, n, x)

   i = 1; j = n;       // oblast pretrage je ograničena indeksima i i j
   while (i <= j) do
      k = (i + j)/2;   // indeks srednjeg elementa
      if (x < a[k]) then
         j = k - 1;    // x se nalazi u prvoj polovini oblasti pretrage
      else if (x > a[k]) then
         i = k + 1;    // x se nalazi u drugoj polovini oblasti pretrage
      else
         return k;     // x je nađen
   return 0;           // x nije nađen
```

U algoritmu `bin-search` se najpre pravilno inicijalizuje oblast pretrage niza koja je ograničena donjim indeksom i i gornjim indeksom j. Ta oblast se na početku naravno proteže na ceo niz od prvog do n-tog elementa. Zatim se, sve dok donji indeks oblasti pretrage ne premašuje njen gornji indeks (tj. $i \leq j$), izračunava indeks k srednjeg elementa oblasti pretrage (pri čemu se podrazumeva celobrojno deljenje brojem dva) i ispituje taj srednji element a_k. Ako ovaj element nije jednak traženom broju x, pretraga se nastavlja u prvoj ili drugoj polovini oblasti pretrage zavisno od toga da li je x manje ili veće od a_k. Ako je srednji element jednak traženom broju x, pretraga se prekida i vraća se rezultat k. Na slici 3.3 je ilustrovana prva iteracija binarne pretrage broja x u sortiranom nizu a.



SLIKA 3.3: Binarna pretraga.

Ako se izvršavanje `while` petlje završilo zbog toga što je donji indeks oblasti pretrage premašio njen gornji indeks (tj. $i > j$), to znači da se dužina oblasti pretrage niza smanjila do nule. U tom slučaju dakle broj x nije pronađen u nizu a, pa se kao rezultat vraća 0.

Da bismo odredili (asimptotsko) vreme izvršavanja $T(n)$ algoritma binarne pretrage `bin-search`, primetimo da je dovoljno samo odrediti vreme izvršavanja njegove `while` petlje. Ostali delovi tog algoritma (inicijalizacija oblasti pretrage na početku i `return` naredba na kraju) izvršavaju se za konstantno vreme, koje je zanemarljivo u odnosu na vreme izvršavanja `while` petlje kada je dužina niza n dovoljno velika.

Sa druge strane, vreme izvršavanja `while` petlje je proporcionalno broju iteracija te petlje, jer se telo te petlje izvršava za konstantno vreme. U to se lako možemo uveriti ukoliko primetimo da se telo `while` petlje sastoji samo od jediničnih instrukcija, pa je očigledno njihovo ukupno vreme izvršavanja u najgorem slučaju konstantno. Pojednostavljena analiza algoritma `bin-search` se svodi dakle na određivanje najvećeg broja iteracija `while` petlje u tom algoritmu.

Međutim, najveći mogući broj iteracija `while` petlje nije očigledan, jer se petlja završava kada uslov $i \leq j$ nije ispunjen ili je traženi broj x nađen, a to zavisi od samih brojeva niza a i traženog broja x. Moguće je da broj x

nađemo odmah u prvom koraku ili u nekom koraku posle toga, ali se nije teško uveriti da se najveći broj iteracija `while` petlje izvršava kada se broj x ne nalazi u nizu a. Onda se naime prolazi kroz ceo ciklus sužavanja oblasti pretrage — od celog niza, zatim jedne polovine niza, pa jedne četvrtine niza, pa jedne osmine niza i tako dalje, na kraju do oblasti pretrage od samo jednog elementa niza, posle čega više ne važi uslov $i \leq j$ i petlja se završava.

Koliki je onda najveći broj iteracija `while` petlje u tom slučaju kada se broj x ne nalazi u nizu a? Obratite pažnju na to da na ovo pitanje možemo ne razmišljajući mnogo odgovoriti da je taj broj najviše n, jer se nikad neki element niza ne ispituje više od jednom. To je tačno, ali onda bismo mogli reći i da je taj broj najviše n^2 ili n^3 ili nešto slično. Ovde se srećemo sa još jednim svojstvom asimptotskog vremena izvršavanja algoritma: iako se njegova analiza umnogome pojednostavljuje, ipak se trudimo da dobijemo što tačniju ocenu pravog vremena izvršavanja algoritma.

Da bismo tačnije odredili najveći broj iteracija `while` petlje, moramo na neki način naći dobru meru onoga što se zaista dešava prilikom izvršavanja te petlje u algoritmu `bin-search`. U postupku binarne pretrage suština je da dužina oblasti pretrage početno iznosi n, a na početku svake iteracije osim prve, dužina oblasti pretrage jednaka je otprilike polovini dužine oblasti pretrage na početku prethodne iteracije. Drugim rečima, ako m označava ukupan broj iteracija `while` petlje kada se broj x ne nalazi u nizu a, onda:

- na početku prve iteracije, dužina oblasti pretrage iznosi n;
- na početku druge iteracije, dužina oblasti pretrage iznosi otprilike $n/2 = n/2^1$;
- na početku treće iteracije, dužina oblasti pretrage iznosi otprilike $(n/2)/2 = n/4 = n/2^2$;
- na početku četvrte iteracije, dužina oblasti pretrage iznosi otprilike $(n/4)/2 = n/8 = n/2^3$;
- i tako dalje ...
- na početku poslednje m-te iteracije, dužina oblasti pretrage iznosi otprilike $n/2^{m-1}$.

Ali ono što još znamo je da se oblast pretrage na početku poslednje m-te iteracije sastoji od samo jednog elementa niza. Zato mora biti

$$\frac{n}{2^{m-1}} = 1,$$

odnosno

$$n = 2^{m-1}.$$

Sada lako možemo dobiti ukupan broj iteracija m u zavisnosti od broja elemenata niza n ukoliko poslednju jednakost logaritmujemo za osnovu dva i dobijenu jednakost rešimo po m:

$$m = \log n + 1.$$

Najveći broj iteracija `while` petlje u algoritmu `bin-search` je dakle reda $\log n$, pa je to i red veličine vremena izvršavanja tog algoritma. Još bolje, ukoliko primenimo O-zapis, ovu činjenicu možemo sažeto izraziti izrazom $T(n) = O(\log n)$.

Tipične funkcije vremena izvršavanja

Pojednostavljenom analizom algoritma, čiji je cilj određivanje njegovog asimptotskog vremena izvršavanja za veliki broj ulaznih podataka, dobija se nekoliko standardnih funkcija koje najčešće opisuju vremena izvršavanja algoritama. Poznavajući zato samo ovaj mali broj funkcija možemo mnogo brže i lakše steći uvid u performanse algoritma.

U tabeli 3.1 se nalaze najčešće funkcije koje srećemo u analizi algoritama. Te funkcije su navedene redom odozgo na dole po rastućim brzinama rasta za velike vrednosti svog argumenta.

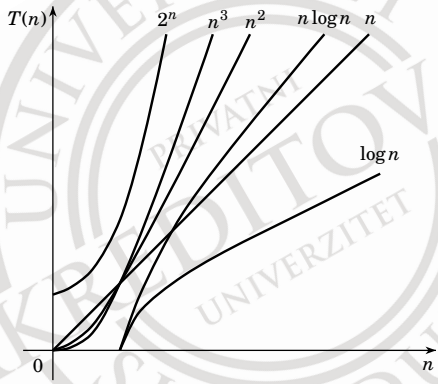
Funkcija	**Neformalno ime**
1	konstantna funkcija
$\log n$	logaritamska funkcija
n	linearna funkcija
$n \log n$	linearno-logaritamska funkcija
n^2	kvadratna funkcija
n^3	kubna funkcija
2^n	eksponencijalna funkcija

TABELA 3.1: Tipične funkcije vremena izvršavanja algoritama.

Funkcije u tabeli 3.1 nisu interesantne u računarstvu zbog svoje apstraktne matematičke prirode, već one predstavljaju asimptotska vremena

izvršavanja algoritama. Uočimo zato da se u toj tabeli svako neformalno ime funkcije odnosi, u stvari, na klasu funkcija iste brzine rasta. Tako, kvadratnom funkcijom se nazivaju sve funkcije koje su reda veličine n^2 za veliko n. Slično, za konstantnu funkciju je uobičajeno da se uzima najprostiji broj 1 za vrednost konstante, mada vrednost konstantne funkcije može biti bilo koji broj nezavisan od n.

Brzina algoritma je dakle bolja ukoliko se njegova funkcija vremena izvršavanja nalazi bliže vrhu tabele 3.1. Drugim rečima, jedan algoritam je brži od drugog ukoliko je njegova funkcija vremena izvršavanja ispod, ili neformalno „manja od", funkcije vremena izvršavanja drugog algoritma. Radi boljeg razumevanja relativne brzine rasta tipičnih funkcija vremena izvršavanja iz tabele 3.1, na slici 3.4 su otprilike prikazani njihovi grafici.



SLIKA 3.4: Tipične funkcije vremena izvršavanja.

Sa slike 3.4 možemo videti da za male vrednosti argumenta n postoje tačke u kojima je jedna kriva početno ispod druge, mada kasnije to nije slučaj. Na primer, početno je kvadratna kriva ispod linearne krive, ali kada n postane dovoljno veliko, ta osobina se menja u korist linearne krive. Za malo n je teško upoređivati funkcije pošto su tada konstante vrlo bitne. Na primer, funkcija $7n + 120$ je veća od n^2 kada je n manje od 15. Ali, za mali broj ulaznih podataka se vreme izvršavanja obično meri delićem sekunde, pa oko toga ne moramo ni da brinemo.

Na slici 3.4 se vidi da je logaritamska funkcija od neke tačke uvek ispod ostalih funkcija. To znači da su logaritamski algoritmi u praksi uvek brži od algoritama ostalih vremenskih složenosti (osim, naravno, konstantne funkcije). Obratite ipak pažnju na to da stvarna vremenska razlika zavisi od konstanti koje učestvuju u izrazima za vremena izvršavanja, pa je

moguće da, recimo, linearno-logaritamski algoritam bude brži od linearnog algoritma za relativno mali broj ulaznih podataka.

Sa druge strane, ono što se potvrđuje na slici 3.4 i što upada u oči u tabelama na slici 3.2 jeste da su eksponencijalni algoritmi beskorisni za veliki broj ulaznih podataka. Eksponencijalni algoritam možemo pažljivo optimizovati i realizovati na vrlo efikasnom mašinskom jeziku, ali opet će takav algoritam raditi neprihvatljivo dugo za veliki broj ulaznih podataka. Prema tome, najpametnija programerska poboljšanja ili sjajni računari ne mogu učiniti brzim neki neefikasan algoritam. Zbog toga, pre nego što uzalud utrošimo vreme na popravku lošeg algoritma ili potrošimo novac na brže računare, moramo se najpre potruditi da napišemo efikasan algoritam.

3.2 Asimptotska notacija

Iz prethodne diskusije možemo zaključiti da algoritme upoređujemo prema njihovim funkcijama asimptotskog vremena izvršavanja: što je takva funkcija „manja", to je algoritam bolji (brži). Obično je jasno da li je neka funkcija „manja" od druge, ali za složenije slučajeve je potrebno imati precizniju definiciju toga što se podrazumeva kada se tvrdi da je neka funkcija „manja" od druge. Asimptotska notacija omogućava upravo da se na precizan način uvede relativni poredak za funkcije.

Asimptotska notacija ima četiri oblika: O-zapis (veliko o), Ω-zapis (veliko omega), Θ-zapis (veliko teta) i o-zapis (malo o). S obzirom na to da se u praksi najčeće koristi O-zapis, u ovom odeljku ćemo najveću pažnju posvetiti tom zapisu. Ali iako u ostalom delu knjige skoro isključivo koristimo O-zapis, u nastavku ćemo se kratko osvrnuti i na ostale varijante asimptotske notacije radi kompletnosti.

Jedna napomena: nekim čitaocima će ovo poglavlje izgledati kao „suva matematika" — nažalost ili na sreću, to je najbolji način da se precizno izrazi ono o čemu se govori.

O-zapis

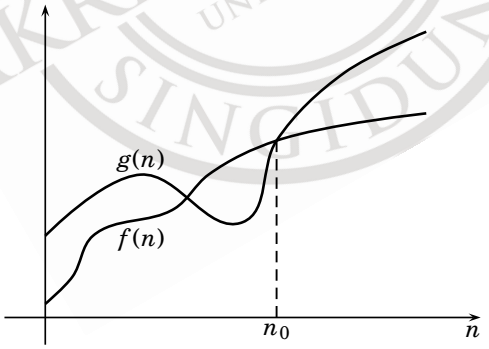
O-zapis se koristi za precizno definisanje svojstva da je neka funkcija „manja" od druge. Na početku napomenimo da za sve funkcije o kojima govorimo, podrazumevamo da su to pozitivne (ili tačnije, nenegativne) funkcije, a nenegativnost se podrazumeva i za njihove argumente. Preciznije

rečeno, posmatramo funkcije koje preslikavaju skup prirodnih brojeva $\mathbb{N}$ u skup nenegativnih realnih brojeva $\mathbb{R}^+$. Ova restriktivnost ne utiče na opštost razmatranja, ali se vodimo time da posmatrane funkcije predstavljaju vremena izvršavanja algoritama, koja su prirodno neke pozitivne vrednosti za neki pozitivni broj ulaznih podataka.

Ako za dve funkcije f i g kažemo da je f manja od g, onda intuitivno mislimo da je $f(n) \leq g(n)$ za sve vrednosti argumenta n. U stvari, pošto nas interesuje asimptotsko ponašanje funkcija (vremena izvršavanja), možemo reći da je $f(n) \leq g(n)$ za sve osim za konačno mnogo vrednosti argumenta n. Ovo je upravo motivacija za O-zapis $f(n) = O(g(n))$ kojim predstavljamo svojstvo da je funkcija f manja od funkcija g, odnosno da funkcija g dominira nad funkcijom f.

DEFINICIJA (O-ZAPIS) *Za dve nenegativne funkcije $f, g: \mathbb{N} \to \mathbb{R}^+$ kažemo da je $f(n) = O(g(n))$ ako postoje pozitivne konstante c i n_0 takve da je $f(n) \leq c \cdot g(n)$ za svako $n \geq n_0$.*

Zapis $f(n) = O(g(n))$ formalno dakle znači da postoji tačka n_0 takva da za sve vrednosti n od te tačke, funkcija $f(n)$ je ograničena vrednošću proizvoda neke pozitivne konstante c i funkcije $g(n)$. Drugim rečima, O-zapis označava da funkcija $g(n)$ predstavlja *asimptotsku gornju granicu* za funkciju $f(n)$. Ova činjenica je ilustrovana na slici 3.5.



SLIKA 3.5: $f(n) = O(g(n))$.

Da bismo ovu sliku interpretirali u kontekstu vremena izvršavanja algoritama, pretpostavimo da funkcija $f(n)$ predstavlja vreme izvršavanja $T(n)$ nekog algoritma i da je recimo $g(n) = n^2$. Sa slike dakle vidimo da ukoliko uspemo da pokažemo da je $T(n) = O(n^2)$, onda smo zapravo pokazali, zanemarujući konstante, da je vreme izvršavanja algoritma sigurno ograničeno kvadratnom funkcijom. Primetimo da O-zapisom dobijamo samo

gornju granicu vremena izvršavanja. To ipak nije veliki problem, pošto ionako vreme rada algoritama određujemo za najgori slučaj izvršavanja, odnosno pravo vreme rada algoritma ograničavamo najgorim vremenom njegovog izvršavanja.

Da bismo za konkretne funkcije pokazali da je $T(n) = O(g(n))$, možemo u krajnjem slučaju koristiti formalnu definiciju O-zapisa. Za to je potrebno da nađemo konkretne vrednosti za n_0 i $c > 0$ takve da je $T(n) \leq cg(n)$ za svako $n \geq n_0$. Na primer, pokažimo da je vreme izvršavanja algoritma `bubble-sort` na strani 34 jednako $O(n^2)$. Pošto smo u ovom slučaju izračunali da je $T(n) = n^2 - n + 1$, za $n \geq 1$ imamo

$$n^2 - n + 1 \leq n^2 + 1 \leq n^2 + n^2 = 2n^2 .$$

Dakle, ako izaberemo $n_0 = 1$ i $c = 2$, po definiciji možemo zaključiti da je $T(n) = O(n^2)$.

Praktični postupak dobijanja dominirajuće funkcije može se znatno ubrzati ukoliko se iskoriste opšte osobine O-zapisa. Neke od ovih osobina, koje zainteresovani čitaoci mogu lako dokazati na osnovu definicije, navodimo u nastavku.

1. Konstantni faktori nisu važni:[2] $T(n) = O(cT(n))$ za svaku pozitivnu konstantu $c > 0$. Na primer, u slučaju algoritma `bubble-sort` možemo tvrditi da je $T(n) = O(3n^2)$ ukoliko u prethodnom dokazu izaberemo konstantu $c = 2/3$. Ili čak možemo tvrditi da je $T(n) = O(0.01n^2)$ ukoliko izaberemo konstantu $c = 200$.

2. Dominantni termi su najvažniji: ako je $T(n) = O\big(f(n) + g(n)\big)$ i $g(n) = O(f(n))$, onda je $T(n) = O(f(n))$. Na primer, opet u slučaju algoritma `bubble-sort`, odmah možemo zaključiti da je $T(n) = O(n^2)$, jer je očigledno $n^2 - n = O(n^2)$ i $1 = O(n^2)$.

3. Pravilo zbira: $O\big(f(n) + g(n)\big) = O(f(n)) + O(g(n))$. Na primer, ako je $T(n) = O(n) + O(17)$, onda je $T(n) = O(n + 17)$, odnosno $T(n) = O(n)$ prema prethodnom pravili jer je $17 = O(n)$.

4. Pravilo proizvoda: $O\big(f(n) \cdot g(n)\big) = O(f(n)) \cdot O(g(n))$. Na primer, ako je $T(n) = n \cdot O(1)$, onda je $T(n) = n \cdot O(1) = O(n) \cdot O(1) = O(n)$.

[2] Pravilo da konstante nisu važne ne treba pogrešno razumeti. Praktični programeri ponekad danima pokušavaju da jedan algoritam ubrzaju za neki vrlo mali faktor. Ovde je reč samo o tome da želimo dobru opštu sliku o algoritmima, koja bez jednostavnosti asimptotske notacije ne bi bila moguća.

5. Pravilo tranzitivnosti: ako je $T(n) = O(f(n))$ i $f(n) = O(g(n))$, onda je $T(n) = O(g(n))$. Na primer, u slučaju algoritma `bubble-sort` je $T(n) = O(n^2)$. Ali kako je očigledno $n^2 = O(n^3)$, možemo reći da je i $T(n) = O(n^3)$.

Poznavajući ove osobine O-zapisa možemo obično relativno lako odrediti dominirajuću funkciju vremena izvršavanja jednostavnih algoritama. Za malo složenije algoritme je potrebno poznavati i relativne odnose nekih osnovnih funkcija, na primer:

- izraz n^a dominira nad izrazom n^b za $a > b$: na primer, $n = O(n^3)$;
- svaka eksponencijalna funkcija dominira nad svakom polinomskom funkcijom: na primer, $n^5 = O(2^n)$;
- svaka polinomska funkcija dominira nad svakom logaritamskom funkcijom: na primer, $(\log n)^3 = O(n)$. Primetimo da odavde sledi da je $n \log n = nO(n) = O(n^2)$.

Pažljivi čitaoci su možda primetili da je O-zapis unekoliko dvosmislen i da se postavlja pitanje kako tim zapisom, u stvari, izražavamo vreme izvršavanja nekog algoritma. Recimo, videli smo da za vreme izvršavanja algoritma `bubble-sort` $T(n) = n^2 - n + 1 = O(n^2)$ možemo reći na osnovu pravila tranzitivnosti da je i $T(n) = O(n^3)$, ili dalje čak i $T(n) = O(n^{10})$.

Sa druge strane, prirodno je tražiti najbolju (u smislu „najbližu") dominirajuću funkciju za vreme izvršavanja. To jest, ako je $T(n) = 2n^2 + n - 1$, to želimo da ocenimo sa $T(n) = O(n^2)$, a ne tehnički tačnom ali lošijom ocenom $T(n) = O(n^3)$. Međutim, prema pravilu da konstantni faktori nisu važni, možemo bez kraja navoditi bliže granice, recimo $T(n) = O(0.01n^2)$ ili $T(n) = O(0.0001n^2)$ i tako dalje.

Da bismo razrešili ovaj naizgled kontradiktoran problem, uvek kada je to moguće koristimo O-zapis koji je najprostiji. To znači da je on izražen jednim termom i da je konstantni faktor uz taj term jednak broju 1. Treba ipak imati na umu da su jednostavnost i najbolja ocena u nekim slučajevima suprostavljeni ciljevi, ali ti slučajevi su srećom retki u praksi.

Na kraju, primetimo da u radu zloupotrebljavamo simbol $=$, jer znak jednakosti nije zapravo simetričan (refleksivan) kada se koristi O-zapis. To znači da ako je $f_1(n) = O(g(n))$ i $f_2(n) = O(g(n))$, onda to ne implicira da je $f_1(n) = f_2(n)$. Striktno govoreći, za nenegativnu funkciju $g : \mathbb{N} \to \mathbb{R}^+$, $O(g(n))$ treba definisati kao *klasu* nenegativnih funkcija:

$$O(g(n)) = \left\{ f : \mathbb{N} \to \mathbb{R}^+ \mid (\exists c > 0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)\, f(n) \leq cg(n) \right\}.$$

Međutim, radi jednostavnosti se nauštrb preciznosti tradicionalno više koristi prostija definicija koju smo naveli. Nadamo se da to neće zbuniti čitaoce koji su više naklonjeni „matematičkoj korektnosti".

Ω-zapis, Θ-zapis i *o*-zapis

Baš kao što je O-zapis analogan relaciji $\leq$ za funkcije, tako isto možemo definisati analogne zapise za relacije $\geq$ i $=$ na sledeći način: $f(n) = \Omega(g(n))$ ako je $g(n) = O(f(n))$, a $f(n) = \Theta(g(n))$ ako je istovremeno $f(n) = O(g(n))$ i $f(n) = \Omega(g(n))$. Poslednji oblik asimptotske notacije, o-zapis, nema svoju analogiju sa uobičajenim relacijama poređenja, ali se ponekad može pokazati korisnim za opis asimptotskih relacija među funkcijama.

Definicija (Ω-zapis) *Za dve nenegativne funkcije* $f, g: \mathbb{N} \to \mathbb{R}^+$ *kažemo da je* $f(n) = \Omega(g(n))$ *ako postoje pozitivne konstante* c *i* n_0 *takve da je* $f(n) \geq c \cdot g(n)$ *za svako* $n \geq n_0$.

Definicija (Θ-zapis) *Za dve nenegativne funkcije* $f, g: \mathbb{N} \to \mathbb{R}^+$ *kažemo da je* $f(n) = \Theta(g(n))$ *ako je* $f(n) = O(g(n))$ *i* $f(n) = \Omega(g(n))$.

Definicija (*o*-zapis) *Za dve nenegativne funkcije* $f, g: \mathbb{N} \to \mathbb{R}^+$ *kažemo da je* $f(n) = o(g(n))$ *ako za svaku pozitivnu konstantu* c *postoji konstanta* n_0 *takva da je* $f(n) \leq c \cdot g(n)$ *za svako* $n \geq n_0$.

Definicija Ω-zapisa ukazuje da funkcija $g(n)$ predstavlja *asimptotsku donju granicu* za funkciju $f(n)$. Ovaj oblik se obično koristi u dokazima donjih granica vremena izvršavanja kada želimo da pokažemo da se neki algoritam mora izvršiti bar za neko određeno vreme.

Definicija Θ-zapisa ukazuje da funkcija $g(n)$ predstavlja *asimptotski najbolju ocenu* za funkciju $f(n)$. Ukoliko koristimo ovaj oblik u analizi algoritama, onda ne samo da dajemo gornju granicu za vreme izvršavanja nekog algoritma, nego i njegovu donju granicu. Drugim rečima, vreme izvršavanja algoritma je najbolje moguće ocenjeno. Uprkos ovoj dodatnoj preciznosti Θ-zapisa češće se ipak koristi O-zapis, jer je obično mnogo lakše odrediti samo gornju granicu vremena izvršavanja.

Na kraju, definicija o-zapisa označava da je brzina rasta funkcije $f(n)$ reda veličine manja od brzine rasta funkcije $g(n)$. To znači da vrednost funkcije $f(n)$ postaje mnogo manja u poređenju sa vrednošću funkcije $g(n)$ kako n teži beskonačnosti, odnosno

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

Intuitivno značenje četiri oblika asimptotske notacije ilustrovano je u tabeli 3.2.

Asimptotska notacija	**Intuitivno značenje**
$f(n) = O(g(n))$	$f \leq g$
$f(n) = \Omega(g(n))$	$f \geq g$
$f(n) = \Theta(g(n))$	$f \approx g$
$f(n) = o(g(n))$	$f \ll g$

TABELA 3.2: Intuitivno značenje asimptotske notacije.

Radi kratke ilustracije prethodnih definicija, tačnije Θ-zapisa, pokažimo da za vreme izvršavanja algoritma `bubble-sort` važi $T(n) = \Theta(n^2)$. Kako je $T(n) = n^2 - n + 1$, a već smo pokazali da je $T(n) = O(n^2)$, dovoljno je samo pokazati da je i $T(n) = \Omega(n^2)$. Po definiciji Ω-zapisa treba dakle da odredimo pozitivne konstante c i n_0 takve da je

$$n^2 - n + 1 \geq cn^2,$$

za svako $n \geq n_0$. Deljenjem ove nejednakosti sa n^2 dobijamo

$$1 - \frac{1}{n} + \frac{1}{n^2} \geq c.$$

Odavde je jasno da možemo uzeti konstante $c = 1/2$ i $n_0 = 1$ tako da prva nejednakost bude zadovoljena.

Postoji drugi način da pokažemo da data funkcija zadovoljava neku asimptotsku granicu. Naime, često je lakše to pokazati pomoću graničnih vrednosti nego direktno po definiciji. Naredna teorema, čiji lagan dokaz prepuštamo čitaocima, daje dovoljne uslove za razne tipove asimptotskih granica.

TEOREMA (PRAVILO LIMESA) *Za nenegativne funkcije $f(n)$ i $g(n)$,*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0 \Rightarrow f(n) = \Theta(g(n))$$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \geq 0 \Rightarrow f(n) = O(g(n))$$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}\neq 0\Rightarrow f(n)=\Omega(g(n))$$
$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=0\Rightarrow f(n)=o(g(n)),$$

gde je c neka konstanta, a u slučaju Ω-zapisa limes može biti beskonačan.

Pravilo limesa je skoro uvek lakše primeniti nego formalne definicije. Pored toga, mada naizgled to pravilo zahteva da se pokažu da važe strožiji uslovi, ono se može primeniti u skoro svim praktičnim primerima funkcija vremena izvršavanja. Jedini izuzeci su slučajevi kada granična vrednost ne postoji (recimo, $f(n)=n^{\sin n}$), ali pošto su vremena izvršavanja obično „dobre" funkcije, to retko predstavlja problem.

Na primer, ako treba uporediti funkcije $\log^2 n$ i $n/\log n$, možemo izračunati limes njihovog količnika (pomoću, recimo, Lopitalovog pravila):

$$\lim_{n\to\infty}\frac{\log^2 n}{n/\log n}=0.$$

Po pravilu limesa je zato $\log^2 n = O(n/\log n)$.

Treba još naglasiti da asimptotska notacija ne uvodi totalni poredak među funkcijama. To znači da postoje parovi funkcija koje nisu uporedive, recimo, O-zapisom. To jest, postoje funkcije $f(n)$ i $g(n)$ takve da niti je $f(n)$ jednaka $O(g(n))$ niti je $g(n)$ jednaka $O(f(n))$. Na primer,

$$f(n)=\begin{cases}n^3, & \text{ako je } n \text{ parno}\\ n, & \text{ako je } n \text{ neparno}\end{cases}\quad\text{i}\quad g(n)=\begin{cases}n, & \text{ako je } n \text{ parno}\\ n^2, & \text{ako je } n \text{ neparno}\end{cases}$$

takve su *neuporedive* funkcije, jer nijedna nije veliko-o od druge.

Primer: Euklidov algoritam

U odeljku 2.1 na strani 19 razmotrili smo problem najvećeg zajedničkog delioca: ako su data dva pozitivna cela broja x i y, odrediti njihov najveći zajednički delilac nzd(x,y). Tamo smo pokazali (neefikasan) algoritam `gcd` koji rešava taj problem, a ovde ćemo predstaviti drugi, Euklidov algoritam za isti problem koji je mnogo brži.

Ali pošto algoritam `gcd` nismo analizirali u odeljku 2.1, ovde ćemo ga najpre ponoviti da bismo mogli lakše da ocenimo njegovo vreme izvršavanja i uporedimo to sa Euklidovim algoritmom.

```
// Ulaz:  pozitivni celi brojevi x i y
// Izlaz: nzd(x,y)
algorithm gcd(x, y)

   d = min{x,y};
   while ((x % d != 0) || (y % d != 0)) do
      d = d - 1;

   return d;
```

Očigledno je vreme izvršavanja ovog algoritma proporcionalno broju iteracija `while` petlje, jer se ostale naredbe izvršavaju za konstantno vreme. Taj broj iteracija sa svoje strane zavisi od ulaznih brojeva x i y. Najbolji slučaj je kada su ulazni brojevi jednaki, $x = y$, jer je onda broj iteracija jednak 0. U slučaju kada su ulazni brojevi različiti, broj iteracija je $\min\{x, y\} - 1$, pa je najgori slučaj kada se ulazni brojevi razlikuju za jedan.

Pošto je vreme izvršavanja algoritma `gcd` linearna funkcija od manjeg od dva ulazna broja, da li to znači da je on brz algoritam? Odgovor je negativan, što je možda neočekivano s obzirom na to da smo govorili da su linerani algoritmi jedni od najbržih mogućih algoritama. Objašnjenje ove kontroverze sadržano je u odgovoru na pravo pitanje u vezi sa aritmetičkim algoritmom `gcd`.

Ključno pitanje za analizu svakog aritmetičkog algoritma je: šta je zapravo veličina ulaza za taj algoritam? Na primer, ako algoritam `gcd` primenimo na dva broja x i y od, recimo, 100 cifara, onda ne možemo realistično očekivati da se aritmetičke operacije koje učestvuju u izračunavanju $\text{nzd}(x, y)$ mogu obaviti za jednu jedinicu vremena. Stoga u aritmetičkim algoritmima moramo sami realizovati celobrojnu aritmetiku „veće tačnosti", odnosno koristiti posebne algoritme za rad sa velikim brojevima. U konkretnom slučaju algoritma `gcd` radi se o algoritmima za operaciju dodele, računanje po modulu i smanjivanje vrednosti promenljive za jedan. Ti algoritmi na ulazu dobijaju (velike) brojeve u obliku niza cifara i njihova vremenska složenost nije jedinična, nego zavisi od broja ovih cifara. Zbog toga se i vreme ovih algoritama, koje može biti poprilično, mora uzeti u obzir prilikom analize aritmetičkh algoritama.

Iz ove diskusije se može naslutiti da je prava mera veličine ulaza aritmetičkh algoritama jednaka zbiru broja cifara (tačnije, broja bitova u binarnom zapisu) njihovih ulaznih brojeva. Naime, sâm broj ulaznih brojeva nije dobra mera, jer je konstantan (na primer, algoritam `gcd` uvek ima dva ulazna broja). Ali broj bitova u binarnom zapisu ulaznih brojeva raste sa magnitudom ulaznih brojeva i time se na pravi način obuhvata vremen-

ska složenost aritmetičkih operacija u zavisnosti od veličine brojeva koji u njima učestvuju.

Primenimo ovaj pristup u analizi algoritma gcd. Neka je n_1 broj bitova binarnog zapisa ulaznog broja x i n_2 broj bitova binarnog zapisa ulaznog broja y. Mera veličine ulaza tog algoritma n je dakle zbir brojeva bitova binarnog zapisa ulaznih brojeva: $n = n_1 + n_2$. Sada dakle treba, zavisno od ove prave mere veličine ulaza za algoritam gcd, oceniti njegovo vreme izvršavanja u najgorem slučaju.

U takvoj analizi algoritma gcd možemo čak zanemariti činjenicu da se osnovne aritmetičke operacije ne izvršavaju za jedinično vreme, jer to vreme samo doprinosi ukupnom vremenu izvršavanja, koje je u ovom slučaju ionako veliko i iznosi bar $2^{n/2}$. Da bismo ovo pokazali, primetimo najpre da je broj bitova binarnog zapisa svakog pozitivnog celog broja a jednak $\lfloor \log a \rfloor + 1$.[3] To znači da je $n_1 \approx \log x$ i $n_2 \approx \log y$, odnosno $x \approx 2^{n_1}$ i $y \approx 2^{n_2}$.

U najgorem slučaju algoritma gcd kada su ulazni brojevi susedni, recimo $x = y - 1$, broj njihovih bitova je otprilike podjednak, odnosno $n_1 \approx n_2 \approx n/2$. Kako je vreme izvršavanja tog algoritma $T(n)$ proporcionalno vrednosti manjeg od dva ulazna broja, to je:

$$T(n) = \Omega(y) = \Omega\left(2^{n_2}\right) = \Omega\left(2^{n/2}\right).$$

To pokazuje da se algoritam gcd izvršava za ekponencijalno vreme, što je beskorisno za velike brojeve od 100–200 cifara.

Jedna oblast primene računara u kojoj se problem izračunavanja najvećeg zajedničkog delioca velikih brojeva često pojavljuje jeste kriptografija. Bez preterivanja možemo reći da mnoge važne oblasti ljudske delatnosti (vojska, finansije, Internet, ...) ne bi mogle da funkcionišu bez efikasnog algoritma za njegovo rešavanje. Srećom, takav algoritam je bio poznat još u antičkim vremenima i naziva se Euklidov algoritam. Ovaj algoritam se zasniva na sledećoj prostoj činjenici:

TEOREMA (EUKLID) *Ako su x i y pozitivni celi brojevi takvi da $x \geq y$, onda je* $\text{nzd}(x, y) = \text{nzd}(y, x\%y)$.

Dokaz: Dovoljno je pokazati da su skupovi zajedničkih delilaca celobrojnih parova brojeva (x, y) i $(y, x\%y)$ jednaki, jer su onda i njihovi *najveći*

[3] Ako je r neki realan broj, tada $\lfloor r \rfloor$ (engl. *floor*) označava celobrojni deo realnog broja r, odnosno najveći ceo broj manji ili jednak r. Prema tome, $\lfloor 7.51 \rfloor = 7$ i $\lfloor 7 \rfloor = 7$. Slično, $\lceil r \rceil$ (engl. *ceiling*) označava najmanji ceo broj veći ili jednak r. Prema tome, $\lceil 7.51 \rceil = 8$ i $\lceil 7 \rceil = 7$.

zajednički delioci jednaki. U tom cilju, svaki ceo broj koji deli oba broja x i y bez ostatka, mora deliti bez ostatka i brojeve y i $x\%y = x - ky$, ($k \in \mathbb{N}$). Obrnuto, svaki ceo broj koji deli bez ostatka i y i $x\%y = x - ky$, ($k \in \mathbb{N}$), mora deliti bez ostatka kako $x = x\%y + ky$, ($k \in \mathbb{N}$), tako i y. ■

Uz primedbu da svaki pozitivan ceo broj x deli broj 0 bez ostatka i da zato $\text{nzd}(x,0) = x$, najveći zajednički delilac dva broja možemo lako izračunati uzastopnim ponavljanjem jednakosti iz Euklidove teoreme. Neki primeri tog izračunavanja za razne parove brojeva su:

- $\text{nzd}(1055,15) = \text{nzd}(15,5) = \text{nzd}(5,0) = 5$
- $\text{nzd}(400,24) = \text{nzd}(24,16) = \text{nzd}(16,8) = \text{nzd}(8,0) = 8$
- $\text{nzd}(101,100) = \text{nzd}(100,1) = \text{nzd}(1,0) = 1$
- $\text{nzd}(34,17) = \text{nzd}(17,0) = 17$
- $\text{nzd}(89,55) = \text{nzd}(55,34) = \text{nzd}(34,21) = \text{nzd}(21,13) =$
  $= \text{nzd}(13,8) = \text{nzd}(8,5) = \text{nzd}(5,3) =$
  $= \text{nzd}(3,2) = \text{nzd}(2,1) = \text{nzd}(1,0) = 1$

Na osnovu ovih primera se može otkriti Euklidov postupak kojim se dobija najveći zajednički delilac dva broja: dati par brojeva se transformiše u niz parova, pri čemu je prvi broj narednog para jednak drugom broju prethodnog para, a drugi broj narednog para jednak je ostatku celobrojnog deljenja prvog broja sa drugim brojem prethodnog para. Poslednji par u nizu je onaj kod koga je drugi broj jednak nuli, a onda je najveći zajednički delilac početnog para jednak prvom broju tog poslednjeg para.

Ispravnost ovog postupka je očigledna na osnovu Euklidove teoreme, ali se postavlja pitanje da li će se niz parova jedne transformacije sigurno završiti za svaki početni par pozitivnih celih brojeva (x,y) takvih da $x \geq y$. Da je to slučaj nije se teško uveriti ukoliko primetimo da je drugi broj narednog para uvek striktno manji od drugog broja prethodnog para. To sledi prosto zato što je uvek $x\%y < y$. Prema tome, ovo smanjivanje drugih brojeva u parovima niza jedne transformacije ne može se beskonačno produžiti, jer najgori slučaj tih brojeva u nizu je $y, y-1, y-2, \ldots, 1, 0$.

Precizniji opis Euklidovog algoritma na pseudo jeziku je:

```
// Ulaz:  celi brojevi x i y takvi da x ≥ y > 0
// Izlaz: nzd(x,y)
algorithm euclid(x, y)

   while (y > 0) do
      z = x % y;
```

```
    x = y;
    y = z;

return x;
```

Preostaje još da odredimo vreme izvršavanja Euklidovog algoritma kako bismo se uverili da je vrlo efikasan. U stvari, pokazaćemo da je vreme izvršavanja Euklidovog algoritma $T(n) = O(n^3)$, pri čemu je $n = n_1 + n_2$ i n_1 i n_2 su brojevi bitova binarnog zapisa ulaznih brojeva x i y. To je ogromno, eksponencijalno ubrzanje u odnosu na prvi algoritam `gcd`.

Pre svega, primetimo da u vremenu izvršavanja Euklidovog algoritma dominira broj iteracija `while` petlje u algoritmu `euclid`. Primetimo i da je taj broj ekvivalentan dužini niza parova jedne transformacije od početnog para (x, y). Ali pitanje dužine niza parova jedne transformacije je delikatno. Sa jedne strane, kao što možemo videti iz datih primera izračunavanja za razne parove, neki parovi brojeva zahtevaju dve-tri transformacije, dok je kod drugih parova brojeva odogovarajući niz transformacija znatno duži. Sa druge strane, kada smo utvrdili da je niz parova jedne transformacije uvek konačan, praktično smo pokazali da je njegova dužina najviše y. Ova gruba ocena nas međutim ne zadovoljava, jer time ne bismo dobili ništa bolje vreme od onog za prvi algoritam `gcd`.

Tačnija ocena vremena izvršavanja Euklidovog algoritma zasniva se na sledećoj činjenici:

LEMA *Ako je $x \geq y > 0$, onda je $x\%y < x/2$.*

Dokaz: Moguća su dva slučaja: $y \leq x/2$ ili $y > x/2$. U prvom slučaju je $x\%y < y \leq x/2$. U drugom slučaju je $x\%y = x - y < x/2$. ■

Pretpostavimo da u algoritmu `euclid` za ulazne brojeve x i y, $x \geq y$, imamo m iteracija `while` petlje pre njene poslednje iteracije. Označimo ulazne brojeve algoritma sa x_0 i y_0 da ih ne bismo mešali sa promenljivim x i y u telu `while` petlje. Pretpostavimo dalje da se izvršavanjem `while` petlje dobija niz parova brojeva:

$$(x_0, y_0) \to (x_1, y_1) \to (x_2, y_2) \to \cdots \to (x_m, y_m) \to (x_{m+1} = \text{nzd}(x, y), y_{m+1} = 0),$$

gde (x_i, y_i) označava par brojeva koji su vrednosti promenljivih x i y u telu `while` petlje nakon izvršavanja i-te iteracije.

Kako bismo sada ocenili broj iteracija `while` petlje m, najpre ćemo oceniti proizvod parova brojeva $x_i \cdot y_i$ primenjujući prethodnu lemu više puta:

– nakon prve iteracije je $x_1 = y_0$ i $y_1 = x_0 \% y_0$, a kako je $x_0 \geq y_0$ to je na osnovu leme:

$$x_1 \cdot y_1 = y_0 \cdot (x_0 \% y_0) < x_0 \cdot y_0 / 2$$

– nakon druge iteracije je $x_2 = y_1$ i $y_2 = x_1 \% y_1$, a kako je $x_1 \geq y_1$ to je na osnovu leme i prethodne nejednakosti:

$$x_2 \cdot y_2 = y_1 \cdot (x_1 \% y_1) < x_1 \cdot y_1 / 2 < x_0 \cdot y_0 / 2^2$$

– nakon treće iteracije je $x_3 = y_2$ i $y_3 = x_2 \% y_2$, a kako je $x_2 \geq y_2$ to je na osnovu leme i prethodne nejednakosti:

$$x_3 \cdot y_3 = y_2 \cdot (x_2 \% y_2) < x_2 \cdot y_2 / 2 < x_0 \cdot y_0 / 2^3$$

– i tako dalje ...

– nakon m-te iteracije je $x_m = y_{m-1}$ i $y_m = x_{m-1} \% y_{m-1}$, a kako je $x_{m-1} \geq y_{m-1}$ to je na osnovu leme i prethodne nejednakosti:

$$x_m \cdot y_m = y_{m-1} \cdot (x_{m-1} \% y_{m-1}) < x_{m-1} \cdot y_{m-1} / 2 < x_0 \cdot y_0 / 2^m$$

Kako je svakako $x_m \cdot y_m \geq 1$, a x_0 i y_0 su ulazni brojevi x i y, na osnovu poslednje nejednakosti sledi da je $xy/2^m > 1$, odnosno $xy > 2^m$. Logaritmovanjem ove nejednakosti za osnovu dva napokon dobijamo ocenu za m:

$$m < \log xy = \log x + \log y < n_1 + n_2 = n\,.$$

Obratite pažnju na to da je ovo ključna ocena za veliku efikasnost Euklidovog algoritma: broj iteracija `while` petlje u algoritmu `euclid` je uvek manji ili jednak zbiru brojeva bitova binarnog zapisa ulaznih brojeva. Uporedite to sa nefikasnim algoritmom `gcd` u kome je taj broj iteracija eksponencijalno rastao sa zbirom brojeva bitova binarnog zapisa ulaznih brojeva!

Vreme izvršavanja $T(n)$ Euklidovog algoritma ipak nije optimalna linearna funkcija od n, jer $T(n)$ obuhvata i vreme za osnovne aritmetičke operacije koje nad velikim brojevima nisu jedinične instrukcije. Ovde o tome nećemo mnogo govoriti, nego ćemo samo kratko napomenuti da je izračunavanje ostatka očigledno „najskuplja" operacija, ali da se lako može realizovati tako da njeno vreme izvršavanja bude $O(n^2)$. Zainteresovani čitaoci mogu to i sami pokazati ukoliko primete da se izračunavanje ostatka svodi na deljenje, a za deljenje se može primeniti običan postupak za „ručno" deljenje celih brojeva sa ostatkom.

Uz ostala uprošćavanja možemo dakle uzeti da je vreme izvršavanja jedne iteracije `while` petlje u algoritmu `euclid` jednako $O(n^2)$. Stoga, ukupno vreme izvršavanja ovog algoritma je:

$$T(n) = O(m)O(n^2) = O(n)O(n^2) = O(n^3).$$

Zadaci

1. Odredite (asimptotsko) vreme izvršavanja sledećih algoritamskih fragmenata:

a)
```
if (x == 0) then
    for i = 1 to n do
        a[i] = i;
```

b)
```
i = 1;
repeat
    a[i] = b[i] + c[i];
    i = i + 1;
until (i == n);
```

c)
```
if (x == 0) then
    for i = 1 to n do
        for j = 1 to n do
            a[i,j] = 0;
else
    for i = 1 to n do
        a[i,i] = 1;
```

d)
```
for i = 1 to n do
    for j = 1 to n do
        for k = 1 to j do
            if (i == j) then
                a[k,k] = 0;
```

e)
```
i = 0; k = 1;
while (k <= n) do
    k = 2 * k;
    i = i + 1;
```

f)
```
for i = 1 to n/2 do
    for j = i downto 1 do
        if (j % 2 == 0) then
            a[i,j] = b[i] + c[j];
```

2. Napišite efikasan algoritam za nalaženje najvećeg i drugog najvećeg elementa niza a od n brojeva. Odredite vreme izvršavanja tog algoritma.

3. *Ternarna pretraga.* Binarnu pretragu na nalaženje broja x u sortiranom nizu a od n brojeva možemo modifikovati na sledeći način. Ako je oblast pretrage u sortiranom nizu a indeksirana od i do j, najpre izračunavamo tačku k koja se nalazi na otprilike 1/3 te oblasti, tj. $k = \lfloor (2 \cdot i + j)/3 \rfloor$, a zatim upoređujemo traženu vrednost x sa elementom a_k. U slučaju kada nisu jednaki, postupamo tako što ukoliko je $x < a_k$, pretragu nastavljamo u oblasti od i do $k-1$. A ukoliko je $x > a_k$,

izračunavamo tačku m koja se nalazi na otprilike 2/3 početne oblasti, tj. $m = \lceil (i + 2 \cdot j)/3 \rceil$, i upoređujemo x sa a_m. Drugim rečima, želimo da izolujemo x u tačno jednoj od tri podoblasti dužine jedna trećina polazne oblasti od i do j. Napišite algoritam kojim se realizuje ova tzv. ternarna pretraga. Možete li da odredite njegovo vreme izvršavanja $T(n)$ i uporedite ga sa vremenom izvršavanja binarne pretrage?

4. *Sportski navijač.* Posmatrajmo niz a veličine n koji sadrži samo crvene i bele elemente u proizvoljnom redosledu. (Crveno i belo su hipotetički zvanične boje nekog sportskog tima.) Pretpostavljajući da je proveravanje boje nekog elementa jedinična instrukcija, cilj je da se niz preuredi tako da se svi crveni elementi nalaze na početku, a svi beli elementi da se nalaze iza njih na kraju niza a. Na slici 3.6 je ilustrovan završni rezultat.

SLIKA 3.6.

Navedite bar dva algoritma za rešenje ovog problema i odredite vreme izvršavanja svakog od njih.

5. *Srpska zastava.* Posmatrajmo niz a veličine n koji sadrži crvene, plave i bele elemente u proizvoljnom redosledu. Pretpostavljajući da je proveravanje boje nekog elementa jedinična instrukcija, cilj je da se niz preuredi tako da se svi crveni elementi nalaze na početku, svi plavi elementi da se nalaze u sredini, a svi beli elementi da se nalaze na kraju niza a. Na slici 3.7 je ilustrovan završni rezultat.

SLIKA 3.7.

Navedite bar dva algoritma za rešenje ovog problema i odredite vreme izvršavanja svakog od njih.

6. Dokažite pravilo limesa na strani 59.

7. Formalno dokažite sledeće asimptotske relacije:

a) $10n^3 + n^2 + 40n + 80 = \Theta(n^3)$
b) $17n\log n - 23n - 10 = \Theta(n\log n)$
c) $n\log n = O(n^2)$
d) $\log^2 n = O(n)$
e) $\log n = O(\sqrt{n})$
f) $n^{\log n} = O(2^n)$
g) $\sin n = O(1)$

8. U svakom od ovih slučajeva odredite da li je $f(n) = O(g(n))$ ili $f(n) = \Omega(g(n))$ ili $f(n) = \Theta(g(n))$.

a) $f(n) = n + 100, \quad g(n) = n + 200$
b) $f(n) = n^{1/2}, \quad g(n) = n^{2/3}$
c) $f(n) = 10n + \log n, \quad g(n) = n + \log^2 n$
d) $f(n) = n\log n, \quad g(n) = 10n\log 10n$
e) $f(n) = \log 2n, \quad g(n) = \log 3n$
f) $f(n) = n^2/\log n, \quad g(n) = n\log^2 n$
g) $f(n) = (\log n)^{\log n}, \quad g(n) = n$
h) $f(n) = (\log n)^{\log n}, \quad g(n) = 2^{\log^2 n}$
i) $f(n) = \sqrt{n}, \quad g(n) = \log^3 n$
j) $f(n) = n^{\log n}, \quad g(n) = 2^n$
k) $f(n) = n2^n, \quad g(n) = 3^n$
l) $f(n) = 2^n, \quad g(n) = 2^{n/2}$
m) $f(n) = n!, \quad g(n) = 2^n$
n) $f(n) = n!, \quad g(n) = n^n$
o) $f(n) = \sum_{i=1}^{n} i^k, \quad g(n) = n^{k+1}$

9. Poređajte sledeće funkcije po brzini njihovog rasta za velike vrednosti argumenta n:

$$n, \log n, \log\log n, \log^2 n, \frac{n}{\log n}, n!, \sqrt{n}\log^2 n, \left(\sqrt{2}\right)^{\log n}, (1/3)^n, (3/2)^n, 17$$

10. Ako su x i y pozitivni celi brojevi, dokažite sledeće činjenice o njihovom najvećem zajedničkom deliocu.

a) Ako je $x = 2a$ i $y = 2b$, onda je $\text{nzd}(x, y) = 2 \cdot \text{nzd}(a, b)$.
b) Ako je $x = 2^k a$ i $y = 2^k b$ za $k \geq 0$, onda je $\text{nzd}(x, y) = 2^k \cdot \text{nzd}(a, b)$.
c) Ako je x paran i y neparan, onda je $\text{nzd}(x, y) = \text{gcd}(x/2, y)$.
d) Ako je $x \geq y$, onda je $\text{nzd}(x, y) = \text{nzd}(x - y, y)$.

11. Koristeći poslednju činjenicu iz prethodnog zadatka napišite algoritam za izračunavanje najvećeg zajedničkog delioca pozitivnih celih brojeva x i y. (*Savet:* Ako je $x = y$, $\text{nzd}(x, y) = x$; ako je $x > y$, $\text{nzd}(x, y) = \text{nzd}(x - y, y)$; ako je $x < y$, $\text{nzd}(x, y) = \text{nzd}(y - x, x)$.) Pored toga, odredite vreme izvršavanja tog algoritma.

12. Razmotrimo drugu verziju Euklidovog algoritma za izračunavanje najvećeg zajedničkog delioca pozitivnih celih brojeva x i y. Najpre se izračunava najveći stepen broja 2 koji deli oba broja x i y — neka je to 2^k. Zatim se x zamenjuje sa rezultatom deljenja x sa 2^k i y se zamenjuje sa rezultatom deljenja y sa 2^k. Nakon ove „prethodne obrade", ponavlja se sledeći postupak:

1. Zameniti brojeve ako je to neophodno da bismo imali $x \geq y$.
2. Ako je $y > 1$, onda proveriti parnost brojeva x i y. Ako je jedan od tih brojeva paran i drugi neparan, zameniti paran broj njegovom polovinom, tj. x sa $x/2$ ako je x paran ili y sa $y/2$ ako je y paran. Ako su x i y neparni brojevi, zameniti x sa $x - y$. Nakon toga, u svakom slučaju, ponoviti postupak od koraka 1.
3. Ako je $y = 1$, onda vratiti 2^k kao rezultat.
4. Ako je $y = 0$, onda vratiti $2^k x$ kao rezultat.

Za ovu verziju Euklidovog algoritma uradite sledeće:

a) Izračunajte $\text{nzd}(18, 8)$ primenjujući prethodni postupak korak po korak.
b) Čini se da smo u koraku 2 zaboravili slučaj kada su oba broja x i y parna. Pokažite da se to nikad ne može desiti.
c) Pokažite da se ovaj modifikovan Euklidov algoritam uvek završava tačnim odgovorom.
d) Pokažite da se proizvod dva aktuelna broja u promenljivim x i y smanjuje za faktor dva u bar jednoj od svake dve sukcesivne iteracije prethodnog postupka. Drugim rečima, ako je (x_0, y_0) par brojeva u koraku 1 neke iteracije tog postupka i (x_2, y_2) par brojeva

dobijen u koraku 2 nakon dve sukcesivne iteracije tog postupka, tada je $x_2 \cdot y_2 \leq x_0 \cdot y_0/2$.

e) Izvedite gornju granicu za broj iteracija ovog algoritma.

13. *Prošireni Euklidov algoritam.* Euklidovim algoritmom se izračunava više nego samo najveći zajednički delilac dva pozitivna cela broja x i y. Naime, ako primenimo Euklidov algoritam za izračunavanje $\text{nzd}(x, y)$, svi međubrojevi dobijeni tokom izračunavanja se mogu predstaviti kao zbir dva proizvoda: jedan sabirak je proizvod broja x i nekog celog broja, a drugi sabirak je proizvod broja y i nekog drugog celog broja. Koristeći ovo zapažanje dokažite sledeću činjenicu:

TEOREMA $\text{nzd}(x, y)$ *se može predstaviti kao linearna kombinacija brojeva* x *i* y *u obliku* $\text{nzd}(x, y) = ax + by$*, gde su* a *i* b *celi brojevi (koji ne moraju biti pozitivni).*

Napišite prošireni Euklidov algoritam koji izračunava ne samo $\text{nzd}(x, y)$, nego i cele brojeve a i b iz prethodne teoreme za koje važi $\text{nzd}(x, y) = ax + by$.

GLAVA

4

Osnovne strukture podataka

U ovom poglavlju izučavamo neke od najfundamentalnijih struktura podataka u programiranju. Tu spadaju najpre nizovi i liste, koje predstavljaju sekvence elemenata. Zatim izučavamo dva specijalna slučaja listi: stekove, kod kojih se elementi dodaju i uklanjaju samo sa jednog kraja, kao i redove za čekanje, kod kojih se elementi dodaju na jednom kraju i uklanjaju sa drugog kraja. Za svaku od ovih struktura podataka opisaćemo nekoliko konkretnih realizacija i uporediti njihove dobre i loše strane.

4.1 Opšte napomene

Svaki viši programski jezik obezbeđuje neke primitivne tipove podataka. Njihov repertoar se razlikuje od jezika do jezika, ali većina jezika sadrži celobrojni, realni, logički, znakovni i pokazivački tip podataka. Primitivni tipovi podataka predstavljaju osnovne gradivne elemente za konstruisanje složenih tipova podataka. Svaki programski jezik propisuje i pravila za konstruisanje složenih tipova podataka polazeći od primitivnih tipova, koja se takođe razlikuju od jezika do jezika. Najjednostavniji način za strukturiranje podataka u većini viših programskih jezika je niz.[1]

Realizacija *apstraktnog tipa podataka* u računaru zahteva prevođenje njegovog logičkog opisa (specifikacije) u naredbe određenog programskog

[1] Još jedan način za grupisanje podataka u svim programskim jezicima je datoteka (engl. *file*), ali datoteke nećemo koristiti u ovoj knjizi.

jezika. Ovo prevođenje obuhvata upotrebu primitivnih tipova podataka programskog jezika kao gradivnih elemenata i korišćenje raspoloživih mehanizama strukturiranja za njihovo grupisanje. Na taj način se dobija odgovarajuća *struktura podataka*, odnosno kolekcija elemenata povezanih na određen način i skup operacija dozvoljenih nad tom kolekcijom elemenata. Pod elementom se ovde podrazumeva podatak kome tip nije preciziran zato što nije bitan.

Operacije koje su definisane nad strukturama podataka se veoma razlikuju od strukture do strukture, ali sve strukture imaju bar dve zajedničke operacije: jednu za dodavanje novih elemenata kolekciji postojećih elemenata koji sačinjavaju odgovarajuću strukturu podataka, kao i onu za uklanjanje starih elemenata iz kolekcije elemenata. Ovde bi mogli dodati i operaciju za konstruisanje potpuno nove, prazne strukture podataka. Međutim, ona umnogome zavisi od konkretnog programskog jezika, a pošto su detalji prilično trivijalni i uglavnom nas interesuju vremena izvršavanja složenijih operacija, obično ćemo preskočiti ovu operaciju „konstruktora" strukture podataka. Pored toga, iz našeg idealizovanog ugla gledanja zanemarićemo i ostale implementacione detalje, kao što su provera grešaka i oslobađanje memorije onih elemenata koji više nisu potrebni.

U ostale tipične operacije nad strukturama podataka spadaju određivanje da li se dati element nalazi u nekoj strukturi podataka, preuređivanje elemenata strukture podataka u nekom redosledu, određivanje broja elemenata strukture podataka (tj. njene veličine) i tako dalje. Uopšteno govoreći, operacije nad strukturama podataka se mogu podeliti u dve kategorije: (a) *upite*, koji kao rezultat daju neku informaciju o strukturi podataka ne menjajući je, i (b) *modifikujuće operacije*, koje menjaju sadržaj strukture podataka.

Vreme za izvršavanje neke operacije nad strukturom podataka izražava se u funkciji od veličine strukture podataka. To ima smisla, jer je prirodno pretpostaviti da vreme za neku operaciju zavisi od broja elemenata u strukturi podataka nad kojima se primenjuje. Na primer, ako niz ima n elemenata, vreme za operaciju kojom se određuje da li se dati element nalazi u nizu zavisi od dužine niza n.

Dodatne konvencije pseudo jezika. Kako bismo mogli da pokažemo konkretne implementacije složenijih struktura podataka, u pseudo jeziku pretpostavljamo da nam na raspolaganju stoji praktično sve ono što (direktno ili indirektno) obezbeđuju moderni programski jezici. To obuhvata pokazivački tip podataka i mogućnost rada sa nizovima.

U prezentaciji neke strukture podataka uglavnom primenjujemo proceduralni pristup, mada koristimo i neke rudimentarne koncepte objektno orijentisanog programiranja. Naime, pošto se elementi struktura podataka obično sastoje od više delova, pogodno ih je smatrati objektima koji sadrže unutrašnja polja za različite podatke. (Međutim, mogućnost da objekti sadrže metode nećemo koristiti.)

Ako neki pokazivač ukazuje na objekat, taj objekat i njegov pokazivač radi jednostavnosti često poistovećujemo u izlaganju. Tako, umesto rogobatne fraze „pokazivač x koji ukazuje na neki objekat (element)" kažemo prostije „objekat (element) x" misleći zapravo na objekat (element) na koga ukazuje pokazivač x. Isto tako, ako pokazivač x ukazuje na objekat koji sadrži polje y, sadržaj tog polja označavamo zapisom `x.y`. Specijalna vrednost pokazivača kojom se ne ukazuje ni na jedan objekat označava se sa `null`.[2]

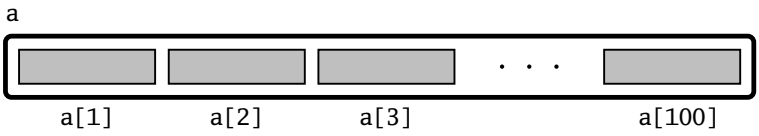
4.2 Nizovi

Niz (engl. *array*) je osnovni način strukturiranja podataka u programiranju i zato se koristi u skoro svim programskim jezicima. Pretpostavljamo da su se čitaoci već sretali u programiranju sa nizovima i zbog toga o njima ovde nećemo govoriti na elementarnom nivou. U stvari, nizove smo već intezivno koristili u prethodnom poglavlju i nadamo se da čitaoci bez predznanja o nizovima nisu imali problema da razumeju njihovu jednostavnu prirodu. Ipak, s obzirom na to da su primene nizova brojne i raznovrsne, ovde ćemo radi kompletnosti istaći njihove konceptualne osobine. Pored toga, u ovom odeljku ćemo pokazati još dva interesantna primera njihove primene, a u drugim odeljcima njihovu primenu u realizaciji složenijih struktura podataka.

Niz je struktura podataka koja predstavlja sekvencu elemenata istog tipa. Sekvencijalna uređenost niza ogleda se u numeraciji njegovih elemenata, redom, od prvog do poslednjeg. Redni broj nekog elementa u nizu se naziva *indeks* elementa. *Ime* niza grupno označava sve elemente niza kao jednu celinu. Za označavanje pojedinačnih elemenata niza koristi se zapis koji se sastoji od imena niza i indeksa odgovarajućeg elementa niza. (Podsetimo se da u ovoj knjizi indeksi niza počinju od 1 i da koristimo zapis indeksa u uglastim zagradama.) Ukupan broj elemenata niza se naziva

[2] Čitaoci koji poznaju Javu mogu ovde prepoznati elemente tog programskog jezika.

dužina ili *veličina* niza. Na primer, niz a dužine 100 se sastoji od 100 elemenata istog tipa čiji je izgled prikazan na slici 4.1.



SLIKA 4.1: Niz a od 100 elemenata.

Dve glavne osobine nizova su:

1. Svi elementi niza moraju biti istog tipa. Ova osobina homogenosti, međutim, ne znači da postoji neko ograničenje za taj tip, odnosno tip pojedinačnih elemenata niza može biti bilo koji, ma kako složen.
2. Dužina niza mora biti unapred zadata i ne može se menjati.

Upravo ove osobine nizova omogućavaju njihovu jednostavnu fizičku reprezentaciju u memoriji računara. Naime, ta memorijska reprezentacija je isključivo neka vrsta sekvencijalne alokacije, odnosno elementi niza zauzimaju neprekidni niz susednih memorijskih lokacija. Zato se praktično i ne razdvaja logički koncept niza od njegove fizičke memorijske reprezentacije.

Primetimo da se ukupna veličina potrebne memorije za neki niz lako određuje kao proizvod broja elemenata i veličine jednog elementa tog niza. Ove dve vrednosti su unapred poznate, jer se dužina niza mora unapred naznačiti, a veličina jednog elementa je poznata na osnovu tipa elemenata niza. Ali ono što je još važnije, lokacija svakog elementa niza se lako izračunava, nezavisno od indeksa elementa: ako je prvi element na lokaciji a i veličina jednog elementa je b bajtova, onda se i-ti element niza nalazi na lokaciji $a + (i - 1)b$. Ovo znači da je vreme pristupa svim elementima niza konstantno, odnosno da se u algoritmima operacija čitanja ili pisanja nekog elementa niza može smatrati jediničnom instrukcijom.

Prema tome, niz je koncepcijski jednostavna i efikasna struktura podataka koju treba koristiti kad god je to moguće. Treba ipak imati u vidu da je struktura niza nepraktična u slučajevima kada se radi o elementima različitih tipova ili njihov približan ukupan broj nije unapred poznat. Ovaj drugi nedostatak se može prevazići zadavanjem neke maksimalne dužine niza, ali to obično dovodi do loše iskorišćenosti memorijskog prostora rezervisanog za niz. Pored toga, operacije dodavanja ili uklanjanja elemenata na proizvoljnoj poziciji niza nisu efikasne, jer dovode do pomeranja svih elemenata od date lokacije do kraja niza za jedno mesto udesno ili ulevo.

Zbog toga struktura niza nije pogodna ni u slučajevima kada su ovakve operacije dominantne.

Primer: linearni algoritam za maksimalnu sumu podniza

Sa problemom maksimalne sume podniza upoznali smo se na strani 29: treba odrediti najveću sumu svih podnizova datog niza brojeva. Preciznije, za dati niz a od n brojeva $a_1, a_2, \ldots, a_n$, ukoliko sa $S_{i,j} = \sum_{k=i}^{j} a_k$ označimo sumu svih elemenata podniza sa indeksima od i do j, onda treba odrediti maksimalnu vrednost M ovih suma:

$$M = \max_{1 \le i \le j \le n} S_{i,j}.$$

Podsetimo se da se kvadratni algoritam za rešenje ovog problema sastoji od dva koraka. U prvom koraku se određuje drugi niz b od n elemenata $b_1, b_2, \ldots, b_n$, pri čemu je element b_i jednak parcijalnom maksimumu suma elementa od i-te pozicije niza:

$$b_i = \max\{S_{i,i}, S_{i,i+1}, \ldots, S_{i,n}\} = \max_{j=i,\ldots,n} S_{i,j}.$$

U drugom koraku se zatim određuje vrednost M:

$$M = \max\{b_1, b_2, \ldots, b_n\} = \max_{i=1,\ldots,n} b_i.$$

Malo poboljšanje ove opšte strategije u kvadratnom algoritmu sastoji se u tome da se dva prethodna koraka ne razdvajaju, nego da se za svako $i = 1, \ldots, n$ redom izračunavaju sume $S_{i,i}, S_{i,i+1}, \ldots, S_{i,n}$ koristeći jednakost $S_{i,j} = S_{i,j-1} + a_j$ i istovremeno određuje njihov maksimum. Znatno poboljšanje, međutim, zahteva bolji uvid u izračunavanje maksimuma M elementa niza b kako bi se to postiglo u jednom prolazu ispitujući ulazni niz sleva na desno. Dobijeni linearni algoritam je tipični primer opšte zakonomernosti: efikasniji algoritmi moraju iskoristiti prirodu samog problema na dublji način. Primetimo i da je zato ispravnost efikasnijih algoritama mnogo manje očigledna i da stoga zahteva pažljiviji dokaz.

Kvadratno vreme za nalaženje maksimalne sume podniza možemo drastično smanjiti do linearnog vremena ukoliko obratimo veću pažnju na sume $S_{i,i}, S_{i,i+1}, \ldots, S_{i,n}$ koje se izračunavaju u i-tom koraku radi određivanja elementa b_i niza b. Ove sume koje se redom izračunavaju zadovoljavaju jedan od ova dva uslova: ili su one sve nenegativne ili su neke od

njih negativne. Prvo zapažanje je da u oba slučaja možemo izbeći mnoga naredna izračunavanja elemenata niza b.

LEMA 1 *Za svako* $i \in \{1, 2, \ldots, n\}$,

1) *Ako su sve sume* $S_{i,i}, S_{i,i+1}, \ldots, S_{i,n}$ *nenegativne, odnosno* $S_{i,j} \geq 0$ *za svako* $j = i, i+1, \ldots, n$, *onda je*

$$M = \max\{b_1, \ldots, b_i\}.$$

2) *Ako su neke od suma* $S_{i,i}, S_{i,i+1}, \ldots, S_{i,n}$ *negativne, neka je* j *najmanji (prvi) indeks tako da je* $S_{i,j} < 0$. *Onda je*

$$M = \max\{b_1, \ldots, b_i, b_{j+1}, \ldots, b_n\}.$$

Dokaz: Oba slučaja se slično dokazuju i zasnivaju se na prostoj činjenici: suma nekog podniza može samo da se smanji ukoliko se podniz smanji za elemente na njegovom početku čija je suma pozitivna. Formalnije, u prvom slučaju ćemo pokazati da je $b_k \leq b_i$ za svako $k = i+1, \ldots, n$, odakle onda sledi da je

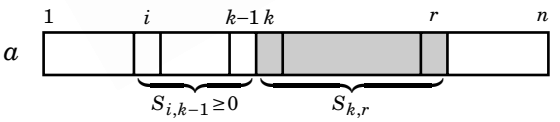
$$M = \max\{b_1, \ldots, b_i, b_{i+1}, \ldots, b_n\} = \max\{b_1, \ldots, b_i\},$$

jer su svi elementi $b_{i+1}, \ldots, b_n$ manji ili jednaki b_i.

Zato za element b_k sa indeksom takvim da $i < k \leq n$ pretpostavimo

$$b_k = \max\{S_{k,k}, S_{k,k+1}, \ldots, S_{k,n}\} = S_{k,r}$$

za neko $r \geq k$, kao što je to ilustrovano na sledećoj slici:



Onda je

$$\begin{aligned} b_k &= S_{k,r} \\ &\leq S_{i,k-1} + S_{k,r}, \quad \text{jer } S_{i,k-1} \geq 0 \text{ po pretpostavci} \\ &= S_{i,r} \\ &\leq b_i, \qquad\qquad \text{jer } b_i = \max_{j=i,\ldots,n} S_{i,j}, \end{aligned}$$

što je trebalo pokazati.

U drugom slučaju je dokaz skoro identičan, osim što se pokazuje da je $b_k \le b_i$ za svako $k = i+1, \ldots, j$, gde je j prvi indeks za koji je $S_{i,j} < 0$. Naime, ako za $i < k \le j$ pretpostavimo $b_k = S_{k,r}$ za neko $r \ge k$, kako je j prvi indeks za koji je $S_{i,j} < 0$, to je $S_{i,k-1} \ge 0$, pa zato

$$b_k = S_{k,r} \le S_{i,k-1} + S_{k,r} = S_{i,r} \le b_i\,.$$

Prema tome,

$$M = \max\{b_1, \ldots, b_i, b_{i+1}, \ldots, b_j, b_{j+1}, \ldots, b_n\} = \max\{b_1, \ldots, b_i, b_{j+1}, \ldots, b_n\},$$

jer su svi elementi $b_{i+1}, \ldots, b_j$ manji ili jednaki b_i. ■

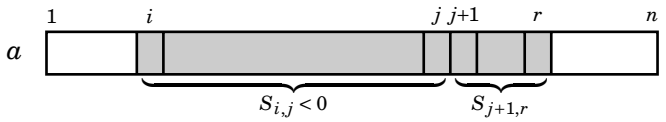
Primetimo da na osnovu prvog zapažanja još ne možemo mnogo da ubrzamo kvadratni algoritam. Prvi slučaj prethodne leme nije problem, jer od i-te pozicije redom ispitujemo elemente niza a do kraja tog niza i nikad se ne vraćamo: kada se ustanovi da su sve sume $S_{i,i}$,$S_{i,i+1}$, …, $S_{i,n}$ nenegativne, postupak se može završiti jer su dalja izračunavanja elemenata $b_{i+1}, \ldots, b_n$ nepotrebna.

U drugom slučaju kada se iza i-te pozicije otkrije prva pozicija j za koju je $S_{i,j} < 0$, onda se mogu preskočiti izračunavanja elemenata $b_{i+1}, \ldots, b_j$ i nastaviti sa izračunavanjem od elementa b_{j+1}. Međutim, problem je što element b_i do pozicije j nije još određen i zato se moraju ispitivati svi elementi niza a do kraja radi izračunavanja suma $S_{i,j+1}$,$S_{i,j+2}$, …, $S_{i,n}$.

U drugom slučaju dakle prolazi se od i-te pozicije niza a do njegovog kraja, a zatim se vraća i nastavlja sličan postupak od pozicije $j+1$. To u najgorem slučaju može opet proizvesti kvadratno vreme. Srećom, drugo zapažanje je da u ovom slučaju, da bismo odredili element b_i, zapravo ne moramo ispitivati sve elemente niza a do kraja radi izračunavanja suma $S_{i,j+1}$,$S_{i,j+2}$, …, $S_{i,n}$.

LEMA 2 *Za svako $i \in \{1, 2, \ldots, n\}$, ako je j najmanji (prvi) indeks tako da je $S_{i,j} < 0$ i $b_i \ne \max\{S_{i,i}, S_{i,i+1}, \ldots, S_{i,j-1}\}$, onda je $b_i \le b_{j+1}$.*

Dokaz: Pretpostavimo $b_i = S_{i,r}$. Kako je $b_i \ne \max\{S_{i,i}, S_{i,i+1}, \ldots, S_{i,j-1}\}$ po pretpostavci, to je $r > j$. (Primetimo da $r \ne j$ pošto $S_{i,j} < 0$.) Ova situacija je ilustrovana na sledećoj slici:



Onda je

$$\begin{aligned} b_i &= S_{i,r} \\ &= S_{i,j} + S_{j+1,r} \\ &\le S_{j+1,r}, && \text{jer } S_{i,j} < 0 \text{ po pretpostavci} \\ &\le b_{j+1}, && \text{jer } b_{j+1} = \max\{S_{j+1,j+1}, \ldots, S_{j+1,n}\} \end{aligned}$$

što je trebalo pokazati. ■

Ova lema implicira da, tokom izračunavanja bilo kog elementa b_i, prvi put kada otkrijemo $S_{i,j} < 0$ možemo bezbedno prekinuti to izračunavanje i nastaviti sa izračunavanjem elemenata b_{j+1}, …, b_n. Naime, ukoliko je element b_i jednak jednoj od suma $S_{i,i}, S_{i,i+1}, \ldots, S_{i,j-1}$, onda svakako ne moramo ispitivati dalje sume $S_{i,j+1}, S_{i,j+2}, \ldots, S_{i,n}$. U suprotnom slučaju, ukoliko je element b_i jednak nekoj daljoj sumi od pozicije j, onda je prema prethodnoj lemi $b_i \le b_{j+1}$. To znači da je element b_i nevažan za izračunavanje maksimuma niza b, pa zato izračunavanje elementa b_i možemo potpuno zanemariti i nastaviti sa izračunavanjem elemenata b_{j+1}, …, b_n.

Postupak koji se zasniva na prethodnim zapažanjima na pseudo jeziku je:

```
// Ulaz:  niz a, broj elemenata n niza a
// Izlaz: maksimalna suma podniza M
algorithm mcss(a, n)

   M = 0;   // maksimalna suma podniza
   i = 1;
   while i < n do   // izračunati b_i
      s = 0;
      for j = i to n do   // izračunati S_{i,j}
         s = s + a[j];        // S_{i,j} = S_{i,j-1} + a_j
         if (s < 0) then      // da li je S_{i,j} < 0?
            break;            // ako jeste, prekinuti sa b_i, b_{i+1}, ..., b_j
         else if (M < s) then // ako nije, ažurirati maksimalnu sumu
            M = s;
      i = j + 1;    // nastaviti sa b_{j+1}, ..., b_n

   return M;
```

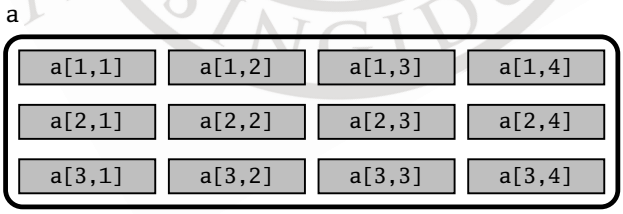
Mada i ova verzija algoritma `mcss` sadrži dve ugnježđene petlje, njegovo vreme izvršavanja je linearno. Naime, primetimo da u svakom koraku spoljašnje `while` petlje povećavamo njen brojač i do pozicije ulaznog niza koja je za jedan veća od pozicije u kojoj je unutrašnja `for` petlja stala. To znači da brojač j unutrašnje petlje dobija sve vrednosti od 1 do n tačno

jednom. Zato se telo unutrašnje petlje izvršava tačno n puta, a kako je za jedno izvršavanje tog tela očigledno potrebno konstantno vreme, to je ukupno vreme izvršavanja ove verzije algoritma $T(n) = O(n)O(1) = O(n)$.

Matrice

Do sada smo govorili, u stvari, o najčešće korišćenom obliku niza koji se opštije naziva *jednodimenzionalni niz* (ili ponekad i *vektor*). Generalizacija tog oblika se može dobiti ukoliko se uzme da su sami elementi jednodimenzionalnog niza opet jednodimenzionalni nizovi. (Podsetimo se da za tip elemenata nizova ne postoji nikakvo ograničenje u vezi sa njihovom složenošću.) Tako se dobija *dvodimenzionalni niz* ili *matrica* elemenata.

Matrica elemenata predstavlja koncepcijski tabelu tih elemenata koja ima određen broj vrsta i kolona. Elementi matrice imaju zato dvostruke indekse: prvi indeks ukazuje na vrstu i drugi na kolonu u kojima se element nalazi u matrici. Pored toga, za matricu elemenata se unapred moraju zadati brojevi njenih vrsta i njenih kolona koji se nazivaju *veličina* matrice. Veličina matrice ne može se menjati, što proizilazi iz nepromenljivosti veličine jednodimenzionalnog niza. Na primer, na slici 4.2 je prikazana matrica elemenata veličine 3×4 koja ima tri vrste i četiri kolone.



SLIKA 4.2: Matrica a veličine 3×4.

Postupak uopštavanja osnovnog koncepta niza kojim se dobija matrica može se slično nastaviti. Ukoliko elementi jednodimenzionalnog niza predstavljaju matrice, dobija se trodimenzionalni niz, odnosno niz matrica. Naravno, ovaj postupak se može dalje nastaviti da bi se dobili *višedimenzionalni nizovi*, ali njihova upotreba u praksi je ograničena na vrlo specijalne primene. Kako se ni konceptualno ništa novo ne dobija daljim uopštavanjem, to o njima u ovoj knjizi više nećemo govoriti.

Primer: problem stabilnih brakova

Problem stabilnih brakova u svom osnovnom obliku su predstavili američki ekonomisti Gejl (*Gale*) i Šejpli (*Shapley*) u vezi sa problemom upisa studenata na univerzitete. Taj početni oblik problema je pretrpeo mnoge modifikacije i njegove razne varijante su našle široku primenu u računarstvu, matematici, ekonomiji, teoriji igara i operacionim istraživanjima. Postupak za rešavanje problema stabilnih brakova otkriva bogatu strukturu tog problema i u računarstvu služi kao primer algoritma čiji je dokaz ispravnosti složeniji nego analiza njegovog vremena izvršavanja.

U opisu problema stabilnih brakova sledimo tradiciju po kojoj se taj problem živopisno definiše kao problem ženidbe mladića sa devojkama (ili udaje devojaka za mladiće). Pretpostavimo da imamo dva disjunktna skupa M i D, oba veličine n, čiji se elementi poistovećuju sa mladićima i devojkama. Prosto rečeno, skup M se sastoji od n mladića i skup D se sastoji od n devojaka. Pretpostavimo dalje da mladići i devojke iz ova dva skupa traže svoje životne partnere u socijalnom okruženju koje predstavlja monogamno i heteroseksualno društvo, što znači da svaka osoba za supražnika želi tačno jednu osobu suprotnog pola. Vezivanje kojim se svakom mladiću iz skupa M dodeljuje različita devojka iz skupa D naziva se *savršeno vezivanje*. Slikovito rečeno, jedno savršeno vezivanje predstavlja jedno moguće kolektivno venčanje svih mladića i devojaka iz dva data skupa. Radi dalje analogije, ako mladiću $m \in M$ u savršenom vezivanju odgovara devojka $d \in D$, to se označava kao (bračni) par (m,d).

Jasno je da za date skupove M i D veličine n postoji veliki broj ($n!$) mogućih savršenih vezivanja, pa bez ikakvih ograničenja to nije mnogo interesantno. Ograničenje koje savršeno vezivanje mora zadovoljiti za problem stabilnih brakova se definiše u duhu toga da sve veze mladića i devojaka budu dugotrajne, odnosno da njihovi brakovi budu stabilni bez razvoda.

Zato pretpostavljamo da svaka osoba ima listu naklonosti svih osoba suprotnog pola koja odražava prvenstvo po kojem bi data osoba želela da stupi u brak sa drugom osobom. Drugim rečima, svaki mladić rangira sve devojke od one koja mu se najviše sviđa pa do one koja mu se najmanje sviđa, a to isto čine i devojke za mladiće. Ovo rangiranje se podrazumeva da je bez jednakih naklonosti tako da se lista naklonosti svake osobe može predstaviti sortiranim nizom u striktno opadajućem redosledu svih n osoba suprotnog pola. Podrazumeva se i da se naklonosti osoba ne menjaju tokom vremena.

Na primer, ako su $M = \{X,Y,Z,U\}$ i $D = \{x,y,z,u\}$ dva skupa po četiri

mladića i devojaka, liste naklonosti mladića u skupu M i devojaka u skupu D mogu biti:

$$X : z,y,x,u \qquad Y : y,x,z,u \qquad Z : x,u,z,y \qquad U : u,z,x,y$$
$$x : X,Y,Z,U \qquad y : U,Z,Y,X \qquad z : Y,Z,X,U \qquad u : Z,U,X,Y$$

Liste naklonosti svih mladića i devojaka je pogodnije predstaviti dvema tabelama, jednom za mladiće i jednom za devojke. Tabelarni oblik svih lista naklonosti u prethodnom primeru je:

mladići	1	2	3	4
X	z	y	x	u
Y	y	x	z	u
Z	x	u	z	y
U	u	z	x	y

devojke	1	2	3	4
x	X	Y	Z	U
y	U	Z	Y	X
z	Y	Z	X	U
u	Z	U	X	Y

Ako je data lista naklonosti osobe p, kažemo da se osobi p *više sviđa* osoba q od osobe r suprotnog pola ukoliko se q nalazi pre r na listi naklonosti osobe p. Tako u prethodnom primeru, devojci z se više sviđa Y od U i mladiću Y se više sviđa x od z.

Nalaženje savršenog vezivanja mladića i devojaka tako da se potpuno zadovolje njihove liste naklonosti je težak zadatak, a u nekim slučajevima je i nemoguć. Odmah je jasno da se ne može obezbediti da svako za bračnog druga dobije svoj prvi izbor: ako je isti mladić prvi izbor za više devojaka, samo jedna devojka mu može postati žena, dok se druge moraju zadovoljiti mužem koji nije prvi na njihovoj listi. Zbog toga, umesto da se u savršenom vezivanju traži najveća sreća za svakoga, postavlja se skromniji društveni cilj da dobijeni brakovi budu stabilni.

Ovaj skromniji cilj podrazumeva da treba pronaći savršeno vezivanje koje sprečava potencijalno saglasno brakolomstvo. To je ono vezivanje koje ne pravi nijedne dve osobe suprotnog pola nezadovoljnim u smislu da se obe osobe međusobno više sviđaju jedno drugom nego njihovi dodeljeni bračni drugovi u vezivanju. Preciznije, neki mladić $m \in M$ i neka devojka $d \in D$ se nazivaju *nezadovoljni par* u savršenom vezivanju ako m i d nisu vezani jedno za drugo u tom vezivanju, odnosno u tom vezivanju su određeni bračni parovi (m,δ) i (μ,d) za neke druge $\delta \in D$ i $\mu \in M$, ali se mladiću m više sviđa d od njegove žene δ i devojci d se više sviđa m od njenog muža μ. (Primetimo da su mišljenja mladića μ i devojke δ nebitna).

Prirodno je pretpostaviti da niko ne bi „švrljao" i stupio u vezu sa nekim ko mu se manje sviđa od dodeljenog partnera, ali ništa ne sprečava neki nezadovoljni par mladića i devojke da raskinu brakove sa svojim dodeljenim bračnim drugovima i međusobno stupe u brak. Radi opšte dobrobiti društva treba zato naći savršeno vezivanje bez nezadovoljnih parova, odnosno takozvano *stabilno vezivanje*. Prethodni zadatak se upravo naziva *problem stabilnih brakova*: za date liste naklonosti svih mladića iz skupa M i svih devojaka iz skupa D odrediti jedno stabilno vezivanje.

Razmotrimo savršeno vezivanje $(X,x),(Y,y),(Z,z),(U,u)$ za prethodni primer listi naklonosti. Mladić Z i devojka u su nezadovoljni par u ovom vezivanju pošto se mladiću Z više sviđa u od njegove žene z, a devojci u se više sviđa Z od njenog muža U. Prema tome to vezivanje nije stabilno.

Ako pokušamo da uklonimo ovu nestabilnost vezivanjem ovog nezadovoljnog para (zamislimo da su se bračni parovi (Z,z) i (U,u) razveli), i zatim vežemo U i z pošto je to jedini izbor za njih, dobijamo novo vezivanje (X,x), (Y,y), (Z,u), (U,z). Međutim, ovo vezivanje je takođe nestabilno zbog X i z. Sada se mladiću X više sviđa z od x, a devojci z se više sviđa X od U. Ovaj postupak sukcesivnih razvoda i vezivanja nezadovoljnih parova možemo nastaviti, ali nije teško smisliti primer u kojem se ovim pristupom vrtimo u krug i nikad ne dobijamo stabilno vezivanje.

Posle ovoga se odmah postavlja pitanje postojanja stabilnog vezivanja u svakoj instanci problema stabilnih brakova. To jest, ako su date proizvoljne liste naklonosti svih mladića i devojaka, da li uvek postoji bar jedno stabilno vezivanje? Odgovor na ovo pitanje je potvrdan i, donekle iznenađujuće, dokaz toga je sasvim konstruktivan. U stvari, dokaz se sastoji od samog algoritma kojim se konstruiše stabilno vezivanje za ulazne liste naklonosti.

Mada je nalaženje bračnog druga i ostajanje s njim u stabilnoj vezi daleko komplikovanije u stvarnom životu, rešenje računarskog problema stabilnih brakova je relativno jednostavno. Glavna ideja originalnog Gejl-Šejpli algoritma može se popularno sažeti u obliku „mladići prose devojke, a devojke biraju ponude". To znači da u toku rada algoritma neki parovi postaju verenici usled prosidbi mladića, a ponekad devojke raskidaju veridbe ako dobiju bolju priliku. Algoritam u glavnim crtama može se opisati na sledeći način:

- Na početku, svi mladići i devojke su slobodni (nisu vereni), a zatim algoritam radi u iteracijama.
- Na početku svake iteracije se bira slobodan mladić koji prosi prvu

(najpoželjniju) devojku na svojoj listi naklonosti od onih koje nikad ranije nije zaprosio. (Primetimo da ovo implicira da nijedan mladić ne prosi istu devojku dvaput.)

- Ako je *slobodna* devojka zaprošena, ona uvek prihvata ponudu, a mladić koji ju je zaprosio i ta devojka postaju verenici (i nisu više slobodni). Ako je *zauzeta* devojka zaprošena, ona proverava svoju listu naklonosti i prihvata ponudu samo ako joj se više sviđa mladić koji ju je zaprosio od svog aktuelnog verenika. Ako je to slučaj, mladić koji je zaprosio devojku i zaprošena devojka postaju novi par verenika, a stari verenik devojke postaje slobodan mladić. Ako se zauzetoj devojci više sviđa aktuelni verenik od mladića koji ju je zaprosio, ona odbija ponudu i ništa se ostalo ne menja.
- Algoritam zatim prelazi u narednu iteraciju sve dok svi mladići ne postanu vereni (neslobodni), kada se sve veridbe proglašavaju brakovima. To jest, svi završni parovi verenika su rezultat koji se vraća kao stabilno vezivanje.

Ovaj neformalni opis izražen malo preciznije na pseudo jeziku je:[3]

```
// Ulaz:  liste naklonosti mladića, liste naklonosti devojaka
// Izlaz: stabilno vezivanje mladića i devojaka
algorithm smp(liste naklonosti mladića i devojaka)

   Svi mladići i devojke su slobodni;
   while (postoji slobodan mladić m) do
      d = prva devojka na m-ovoj listi od onih koje nije zaprosio;
      // m prosi d
      if (d slobodna) then                          // d prihvata m
         m i d su verenici;
      else if (d više voli m od verenika μ) then    // d prihvata m
         m i d su verenici;
         μ je slobodan;
      else                                          // d ne prihvata m
         ništa;

   return veridbe;   // proslaviti stabilne brakove
```

Ako za trenutak zanemarimo programsku realizaciju algoritma `smp`, pažljiviji čitaoci su možda primetili da on ima tri potencijalne greške. Pre svega, nije jasno da li se algoritam uopšte završava, a kamoli da li je njegov rezultat zaista jedno stabilno vezivanje. Naime, u svakoj iteraciji `while`

[3] Ime algoritma `smp` je skraćenica od engleskog termina za problem stabilnih brakova: *the stable marriage problem*.

petlje imamo slobodnog mladića koji može ostati slobodan ukoliko ga odbije njegova najpoželjnija zauzeta devojka koju prosi. Čak i ako ga ona prihvati, njen stari verenik postaje slobodan, tako da se broj slobodnih mladića neće smanjiti. Zato je zamislivo da će uvek postojati slobodan mladić na početku svake iteracije, pa će se algoritam izvršavati beskonačno.

Druga potencijalna greška algoritma je zbog toga što izabrani slobodni mladić na samom početku neke iteracije možda nema devojku na svojoj listi naklonosti koju nije već zaprosio. Naime, možda je moguće da je neki slobodan mladić m već zaprosio sve devojke. Ekvivalentno, ne postoji devojka u listi naklonosti mladića m koju on već nije zaprosio, pa m nema prvu takvu devojku koju bi zaprosio.

Konačno, opis algoritma `smp` nije u potpunosti deterministički, pošto se u uslovu nastavka `while` petlje proverava samo da li postoji slobodan mladić. Ako je to slučaj, u algoritmu nije navedeno kako se bira jedan takav mladić ako postoje više kandidata za aktuelnu iteraciju. Zato se s pravom može postaviti pitanje da li je redosled po kojem se oni biraju važan za rezultujuće vezivanje, odnosno da li možda neki „pogrešan" redosled izbora slobodnih mladića ne bi proizveo nestabilno vezivanje.

Ovo ukazuje da ispravnost algoritma `smp` nije očigledna i da je potrebno pažljivije pokazati njegova svojstva. U nastavku ćemo zato kroz niz formalnih zapažanja odgovoriti na sve prethodne dileme.

LEMA 1 *Za n mladića i n devojaka, algoritam `smp` se završava u najviše n^2 iteracija.*

Dokaz: Primetimo da ovde tvrdimo više od toga da algoritam `smp` ne radi beskonačno — on se završava i to u najviše n^2 iteracija. Da bismo ovo pokazali, moramo naći dobru meru napredovanja `while` petlje. (Videli smo da to nije broj slobodnih mladića.) Pokazaćemo da je to broj prosidbi koje mladići izvedu.

Zaista, nijedan mladić ne prosi istu devojku dvaput, jer neki mladić uvek prosi devojku koju nikad ranije nije zaprosio. To znači da svaki mladić izvede najviše n prosidbi tokom rada algoritma `smp`, odnosno svih n mladića zajedno izvedu najviše n^2 prosidbi ukupno. U svakoj iteraciji `while` petlje se izvodi tačno jedna nova prosidba, pa zato može biti najviše n^2 iteracija. ■

Pre nego što pokažemo da u algoritmu `smp` svaki slobodan mladić ima devojku na svojoj listi koju još nije zaprosio, primetimo jedno interesantno svojstvo tog algoritma. Svaki put kada mladić prosi neku devojku, on to

radi na svoju veću žalost: prvi put prosi svoju najveću ljubav, zatim prosi devojku koja je njegov drugi izbor, pa treći izbor i tako dalje. Devojke sa druge strane svaki put kada promene verenike, rade to na svoju veću sreću. Slobodna devojka od trenutka kada prihvati ponudu nekog mladića koji ju je zaprosio, nikad više ne postaje slobodna — ona može samo promeniti svog verenika ako dobije bolju ponudu iz njenog ugla gledanja, odnosno ako je zaprosi mladić koji joj se više sviđa od aktuelnog verenika.

LEMA 2 *Svaki slobodan mladić na početku* `while` *petlje u algoritmu* `smp` *ima devojku koju još nije zaprosio.*

Dokaz: Primetimo najpre da je na početku izvršavanja svake iteracije `while` petlje u algoritmu `smp` broj slobodnih mladića jednak broju slobodnih devojaka. To je, u stvari, invarijanta te petlje, odnosno uslov koji ostaje nepromenjen nakon izvršavanja svake iteracije petlje. (Čitaoci se o invarijantama petlji mogu podsetiti na strani 20.)

U to se možemo uveriti na sledeći način. Jasno je da je taj uslov zadovoljen na početku prve iteracije, pošto su onda svi mladići i devojke slobodni. Pod pretpostavkom da je taj uslov tačan na početku neke iteracije, izvršavanjem te iteracije se broj slobodnih mladića i broj slobodnih devojaka ili ne menjaju ili oba smanjuju za jedan (kada slobodna devojka prihvati ponudu slobodnog mladića), pa je taj uslov tačan i na početku sledeće iteracije.

Sada možemo pokazati da važi tvrđenje leme. Neka je m slobodan mladić na početku `while` petlje u algoritmu `smp`. Prema invarijanti te petlje, mora postojati i bar jedna slobodna devojka, recimo d. To znači da ovu devojku d do tada još niko nije zaprosio. U suprotnom, da ju je neki mladić ranije zaprosio, ona bi morala da prihvati njegovu ponudu i nikad više ne bi bila slobodna. Prema tome, specifično ni mladić m nije još zaprosio (slobodnu) devojku d. To znači da je bar ta devojka jedna od onih koje slobodni mladić nije zaprosio, što je trebalo pokazati. ■

LEMA 3 *Algoritam* `smp` *kao rezultat daje stabilno vezivanje.*

Dokaz: Neka su broj mladića i broj devojaka jednaki n. Prvo ćemo pokazati da algoritam `smp` proizvodi savršeno vezivanje, odnosno tačno n bračnih parova u kojima se svaki mladić i devojka pojavljuju tačno jednom. Kada se `while` petlja završi, ne postoji slobodan mladić. To znači da su svi mladići vereni bar sa jednom devojkom. Očigledno je da neki mladić može biti veren sa najviše jednom devojkom, jer mladić može biti veren samo kada je slobodan. Stoga, kada se `while` petlja završi, svaki mladić je veren sa tačno jednom devojkom. Sa druge strane, na osnovu invarijante

`while` petlje ne postoji ni slobodna devojka, jer je broj slobodnih mladića (tj. nula) jednak broju slobodnih devojaka. To znači da je i svaka devojka verena bar sa jednim mladićem. Kako je i očigledno da neka devojka može biti verena sa najviše jednim mladićem, to znači je rezultat algoritma `smp` jedno savršeno vezivanje.

Da bismo pokazali da je rezultujuće savršeno vezivanje stabilno, primenićemo način zaključivanja po kontradikciji. Pretpostavimo dakle suprotno, da postoji nezadovoljni par, recimo mladić m i devojka d, u dobijenom vezivanju. To znači da u tom vezivanju imamo bračne parove (m,δ) i (μ,d), ali se mladiću m više sviđa d od njegove žene δ i devojci d se više sviđa m od njenog muža μ. Pošto je (m,δ) bračni par, m je morao da zaprosi δ, jer neki mladić može biti veren samo ako zaprosi devojku. Ali tada je m morao da zaprosi i sve devojke koje su pre njegove žene δ na njegovoj listi naklonosti, jer mladići prose devojke po redu u kojem se one nalaze na njihovoj listi naklonosti. Specifično, tokom rada algoritma m je morao zaprositi i devojku d, jer mu se ona više sviđa od njegove žene δ (i pri tome je m zaprosio d pre nego što je zaprosio svoju ženu δ).

Sa druge strane, pošto je (μ,d) bračni par, suprug μ je po mišljenju devojke d najbolji (najviše rangirani) mladić koji ju je ikad zaprosio, jer devojke menjaju verenike samo ukoliko dobiju bolje ponude. Specifično, pošto je m prosio d, to znači da je μ ispred m na listi naklonosti devojke d. Ali ovo je u suprotnosti sa našom pretpostavkom da je par m i d nezadovoljan, odnosno da se devojci d više sviđa m od njenog muža μ. Dakle naša pretpostavka da postoji nezadovoljni par je neodrživa, jer to vodi u kontradikciju. Ovo pokazuje da ne postoji nijedan nezadovoljni par u dobijenom vezivanju. ■

Preostalo je još da se uverimo da nedeterminizam algoritma nije bitna, odnosno da izbor slobodnog maladića na početku `while` petlje može biti proizvoljan.

LEMA 4 *Algoritam `smp` proizvodi isto stabilno vezivanje bez obzira na to kako se biraju slobodni mladići u svakoj iteraciji `while` petlje.*

Dokaz: Na prosidbu slobodnog mladića ni na koji način ne utiče to šta neki drugi mladić radi. Slično, na izbor nijedne devojke ne utiče to šta druge devojke rade. Prema tome, u svakoj iteraciji nije bitno kojim redom se prosidbe obavljaju niti kojim redom devojke biraju mladiće. ■

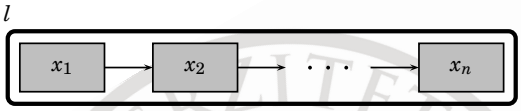
Analizirajmo sada vreme izvršavanja algoritma `smp`. Ako su date liste naklonosti n mladića i n devojaka, znamo da je broj iteracija `while` petlje

najviše n^2. Vreme izvršavanja ovog algoritma biće zato u najgorem slučaju proporcionalno n^2 ukoliko izaberemo njegovu implementaciju tako da se svaka iteracija izvršava za konstantno vreme. Bez mnogo detaljisanja (videti 6. zadatak na kraju poglavlja), u nastavku ćemo samo u grubim crtama navesti moguće strukture podataka koje omogućavaju da se svaki korak u telu `while` petlje realizuje za konstantno vreme:

- Za predstavljanje n mladića i n devojaka možemo koristiti obične brojeve $1, 2, \ldots, n$ za njihova imena.
- Za predstavljanje njihovih veridbi možemo koristiti dva niza a i b od n elemenata. Prirodno, $a_i \in \{1, \ldots, n\}$ označava devojku sa kojom je mladić i veren, a $a_i = 0$ ako je mladić i slobodan. Slično, $b_i \in \{1, \ldots, n\}$ označava mladića za koga je devojka i verena, a $b_i = 0$ ako je devojka i slobodna.
- Za predstavljanje slobodnih mladića možemo koristiti niz s od n elemenata i voditi računa o njegovoj stvarnoj dužini. Postojeći slobodan mladić se uzima sa kraja tog niza, a novi slobodan mladić se takođe dodaje na kraj tog niza. (Čitaoci koji bolje poznaju strukture podataka mogu prepoznati da ovim simuliramo stek, mada se može koristiti i red za čekanje.)
- Za predstavljanje listi naklonosti svih mladića možemo koristiti $n \times n$ matricu m tako da i-ti red te tabele predstavlja listu naklonosti mladića i, sortiranu od najviše do najmanje željene devojke za mladića i. Tako $m_{i,j} = k$ znači da je devojka k na j-tom mestu u listi naklonosti mladića i.
- Za predstavljanje ranga devojke koju je neki mladić poslednje zaprosio možemo koristiti niz r od n elemenata tako da $r_i = j$ označava da se poslednja devojka koju je mladić i zaprosio nalazi na j-tom mestu u njegovoj listi naklonosti, odnosno to je devojka $m_{i,j}$. Na ovaj način možemo lako odrediti prvu devojku koju neki mladić treba da zaprosi.
- Za predstavljanje listi naklonosti svih devojaka možemo koristiti $n \times n$ „matricu rangiranja" d tako da $d_{i,j}$ označava rang mladića j za devojku i u njenoj listi naklonosti. Tako $d_{i,j} = k$ označava da se mladić j nalazi na k-tom mestu u listi naklonosti devojke i. Razlog ove asimetričnosti u odnosu na liste naklonosti mladića je u tome što na ovaj način možemo lako proveriti da li se devojci i više sviđa mladić j_1 od mladića j_2.

4.3 Liste

Lista (tačnije *povezana lista*) je struktura podataka koja je slična nizu jer predstavlja sekvencu elemenata istog tipa. Međutim, za razliku od niza kod koga se sekvencijalna uređenost dobija indeksiranjem elemenata niza, linearni poredak elemenata liste se određuje pokazivačem u svakom elementu na sledeći element u listi. Ova konceptualna slika povezane liste $l = \langle x_1, x_2, \ldots, x_n \rangle$ prikazana je na slici 4.3.



SLIKA 4.3: Povezana lista l.

Obratite pažnju na to da pojam pokazivača u listi kao apstraktnom tipu podataka ne mora da bude isti onaj kao kod programskih jezika. Na apstraktnom nivou je to jednostavno neka indikacija u jednom elementu liste na to koji je sledeći element u logičkom poretku liste. Poslednji element liste sadrži specijalni pokazivač koji ne ukazuje ni na jedan element.

Terminologija u vezi sa listama je slična onoj za nizove. Broj elemenata liste se naziva *veličina* ili *dužina* liste, a lista bez elemenata se naziva *prazna* lista. Za dva susedna elementa liste se kaže da je drugi *sledbenik* prvom ili da je prvi *prethodnik* drugom. Prvi element liste, koji se obično naziva *glava liste*, nema prethodnika; poslednji element liste, koji se obično naziva *rep liste*, nema sledbenika.

Glavna razlika listi u odnosu na nizove je u tome što dužina liste nije unapred određena. Svaka lista je na početku prazna, a zatim se njena dužina povećava ili smanjuje dodavanjem novih ili uklanjanjem postojećih elemenata. Zato lista kao apstraktni tip podataka ima prednost nad nizom u primenama kod kojih ukupan broj nekih istorodnih elemenata nije poznat, ili operacije dodavanja i uklanjanja tih elemenata ne slede unapred poznat obrazac.

Sa druge strane, nizovi imaju prednost kada je potreban efikasan pristup proizvoljnom elementu — pristup svakom elementu kod liste mora početi od prvog elementa i sledeći njegovog sledbenika doći do drugog elementa, zatim sledeći njegovog sledbenika do trećeg i tako dalje, dok se ne stigne do željenog elementa. Prema tome, pristup proizvoljnom elementu

liste nije operacija konstantnog vremena, nego zavisi od njegove pozicije u listi.

Nepotpun spisak tipičnih operacija nad listom je:

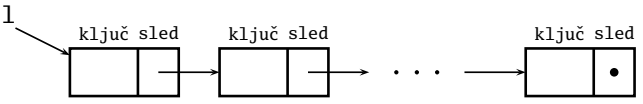
- Konstruisanje prazne liste;
- Dodavanje novog elementa u listu;
- Uklanjanje datog elementa iz liste;
- Pretraga liste.

Implementacija liste pomoću pokazivača

U programskim jezicima koji obezbeđuju rad sa pokazivačima, struktura podataka liste realizuje se na prirodan način: svaki element liste predstavlja se jednim objektom sa dva polja. Ovaj objekat se obično naziva *čvor* liste, a prvo polje čvora sadrži relevantan podatak odgovarajućeg elementa liste, dok drugo polje čvora sadrži pokazavač na sledeći čvor liste.

Pošto sadržaj elemenata liste mogu biti podaci bilo kog (istog) tipa, prvo polje mora biti odgovarajućeg tipa tih podataka koji zavisi od konkretne primene liste. Međutim, na opštem nivou se sadržaj elemenata liste posmatra neutralno i obično se naziva ključem čvora liste. Prema tome, prvo polje u svakom čvoru liste (oznaka `ključ`) poistovećuje se sa pravim sadržajem odgovarajućeg elementa liste, dok drugo pokazivačko polje (oznaka `sled`) ukazuje na sledbenika elementa liste.

Listi l se grupno pristupa preko posebnog spoljašnjeg pokazivača (oznaka `l`) koji nije deo liste i koji pokazuje na prvi čvor liste. Pogodno je pretpostaviti i postojanje specijalne vrednosti pokazivača `null` koja nikad ne ukazuje ni na šta. Na taj način možemo uniformno tretirati čvorove liste i smatrati da poslednji čvor liste u pokazivačkom polju sadrži tu specijalnu vrednost pokazivača. Pored toga, ako pokazivač `l` ima vrednost `null`, onda smatramo da je lista l prazna. Na slici 4.4 je ilustrovana reprezentacija liste pomoću pokazivača.



SLIKA 4.4: Reprezentacija liste l pomoću pokazivača.

Da bismo ilustrovali tipične operacije nad listom, u nastavku ćemo predstaviti algoritme za svaku od njih i analizirati njihovo vreme izvršavanja.

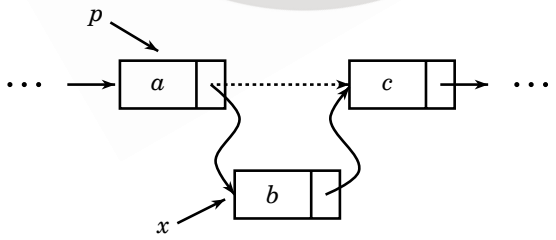
Konstruisanje prazne liste. Konstruisanje prazne liste sastoji se prosto od inicijalizacije njenog pristupnog pokazivača vrednošću `null`:

```
// Ulaz:  lista l
// Izlaz: prazna lista l
algorithm list-make(l)

   l = null;

   return l;
```

Vreme izvršavanja algoritma `list-make` je očigledno konstantno.

Dodavanje novog čvora u listu. Dodavanje elemenata u listu (kao i uklanjanje elemenata iz nje) modifikuje sadržaj liste, pa je potrebno lokalno preurediti listu kako bi se zadržala sekvencijalna uređenost liste sa novim sadržajem. Da bi to bilo moguće, kod dodavanja novog čvora u listu mora se znati čvor liste iza koga se dodaje novi čvor.

Ako je dat (pokazivač na) novi čvor x i (pokazivač na) čvor p liste iza koga se dodaje novi čvor, veza između čvora p i njegovog sledbenika u listi mora se raskinuti i između njih umetnuti čvor x. Na slici 4.5 su prikazane neophodne manipulacije pokazivačima radi dodavanja čvora x u listu iza čvora p.



SLIKA 4.5: Dodavanje čvora x u listu iza čvora p.

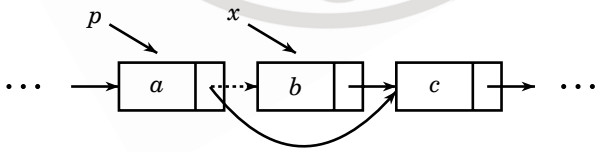
U posebnom slučaju kada novi čvor treba dodati na početak liste, pošto to mesto nema prethodnika pretpostavljamo da je p jednako `null`. Algoritam na pseudo jeziku za dodavanje novog čvora x u listu l iza čvora p te liste je:

```
// Ulaz:  novi čvor x, čvor p u listi l
// Izlaz: lista l sa čvorom x iza čvora p
algorithm list-insert(x, p, l)

   if (p == null) then  // novi čvor dodati na početak
      x.sled = l;
      l = x;
   else                 // novi čvor dodati iza p
      x.sled = p.sled;
      p.sled = x;

   return l;
```

Pošto su sve manipulacije pokazivačima jedinične instrukcije, vreme izvršavanja ovog algoritma za dodavanje čvora u listu je očigledno konstantno.

Uklanjanje čvora iz liste. Slično kao kod dodavanja čvora u listu, za uklanjanje čvora iz liste mora se znati čvor liste iza koga se uklanja postojeći čvor kako bi se lokalne veze mogle preurediti i zadržati sekvencijalna uređenost liste.

Ako je dat (pokazivač na) čvor p liste čijeg sledbenika treba ukloniti, veza između tog čvora i njegovog sledbenika x u listi se raskida, a uspostavlja se nova veza između čvora p i sledbenika starog sledbenika x. Na slici 4.6 su prikazane neophodne manipulacije pokazivačima radi uklanjanja čvora x iz liste koji se nalazi iza čvora p.



SLIKA 4.6: Uklanjanje čvora x iz liste iza čvora p.

Mada za uklanjanje nekog čvora iza datog čvora logički ne treba ništa dodatno uraditi od onog što je prikazano na slici 4.6, praktično je potrebno i osloboditi računarske resurse koje zauzima uklonjeni čvor. To oslobađanje zauzetih resursa se kod nekih programskih jezika mora „ručno” uraditi. U pseudo jeziku za opis algoritama u ovoj knjizi usvajamo lakši pristup i koristimo mogućnost drugih programskih jezika kod kojih se to obavlja automatski postupkom „sakupljanja otpadaka”.

Ako opet pretpostavimo da je p jednako null kada treba ukloniti čvor na početku liste, algoritam na pseudo jeziku za uklanjanje čvora iz liste l iza čvora p te liste je:

```
// Ulaz:  čvor p u listi l iza koga treba ukloniti čvor
// Izlaz: lista l bez uklonjenog čvora
algorithm list-delete(p, l)

   if (p == null) then  // ukloniti čvor na početku
      l = l.sled;
   else                 // ukloniti čvor iza p
      p.sled = p.sled.sled;

   return l;
```

Očigledno kao kod dodavanja čvora u listu, kako su sve manipulacije pokazivačima jedinične instrukcije, vreme izvršavanja ovog algoritma za uklanjanje čvora iz liste je konstantno.

Pretraga liste. Slično problemu pretrage niza, operacijom pretrage liste se određuje da li se u datoj listi l nalazi čvor sa datim ključem k. Ako je to slučaj, algoritam za ovu operaciju u nastavku kao rezultat vraća (pokazivač na) prvi pronađeni čvor sa datim ključem. U suprotnom slučaju, ako čvor sa datim ključem nije pronađen u listi, kao rezultat se vraća pokazivač null.

```
// Ulaz:  lista l, ključ k
// Izlaz: čvor x u listi sa ključem k ili vrednost null
algorithm list-search(l, k)

   x = l;
   while ((x != null) && (x.ključ != k)) do
      x = x.sled;

   return x;
```

Očigledno je da u najgorem slučaju moramo proći kroz celu listu da bismo otkrili da se čvor sa datim ključem ne nalazi u listi. Zbog toga je vreme izvršavanja algoritma list-search u funkciji broja čvorova liste n jednako $O(n)$.

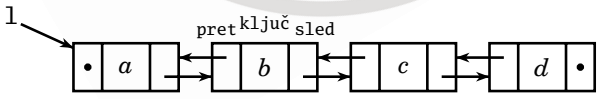
Obratite pažnju na to da smo kod dodavanja ili uklanjanja čvora pretpostavljali da tačno znamo mesto u listi, naime iza datog čvora, na kojem se dešava odgovarajuća promena. Često međutim želimo da dodamo ili uklonimo čvor iza nekog čvora u listi sa datim ključem. U takvom slučaju

zato ne možemo neposredno iskoristiti date algoritme za dodavanje ili uklanjanje čvora, nego najpre moramo koristiti algoritam `list-search` radi nalaženja čvora sa datim ključem i zatim prethodnim algoritmima dodati ili ukloniti čvor iza pronađenog čvora. Prema tome, vreme izvršavanja ovih operacija dodavanja ili uklanjanja čvora za listu od n čvorova bilo bi $O(n)$, a ne $O(1)$.

Dvostruko povezane liste

Liste o kojima smo do sada govorili su *jednostruko povezane*, pošto svaki čvor ima jedno pokazivačko polje koje ukazuje na sledbenika čvora. Jednostruko povezane liste se mogu prelaziti samo u jednom smeru sleva na desno, što smo mogli primetiti u algoritmu `list-search`. Međutim, u nekim primenama je potrebno efikasno prelaziti listu u oba smera ili je za dati čvor potrebno efikasno odrediti njegovog prethodnika, a ne samo njegovog sledbenika.

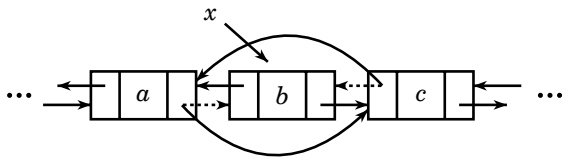
U takvim slučajevima možemo koristiti *dvostruko povezanu listu* u kojoj svaki čvor ima polje ključa i dva pokazivačka polja sa oznakama `sled` i `pret`: polje `sled` kao i ranije ukazuje na sledbenika čvora, dok polje `pret` ukazuje na prethodnika čvora. Prvi čvor dvostruko povezane liste u polju `pret` i poslednji čvor u polju `sled` imaju vrednost `null`. Kao kod jednostruko povezane liste, dvostruko povezanoj listi se pristupa preko posebnog spoljašnjeg pokazivača koji pokazuje na prvi čvor liste, a koji nije deo liste. Na slici 4.7 je prikazan primer dvostruko povezane liste l.



SLIKA 4.7: Dvostruko povezana lista l.

Druga prednost dvostruko povezane liste je u tome što dodavanje čvora možemo izvoditi iza ili ispred datog čvora. Pored toga, za uklanjanje čvora možemo kao parametar direktno koristiti željeni čvor, a ne manje prirodno njegovog prethodnika. Na primer, na slici 4.8 je prikazano kako se neki čvor uklanja iz dvostruko povezane liste na ovaj način.

Algoritam `dlist-delete` u nastavku služi za uklanjanje čvora x iz dvostruko povezane liste l. Jednostavni algoritmi za ostale operacije pre-



SLIKA 4.8: Uklanjanje čvora x iz dvostruko povezane liste.

puštaju se čitaocima za vežbu, kao i ocena vremena izvršavanja svih algoritama.

```
// Ulaz:  čvor x koji treba ukloniti iz l
// Izlaz: lista l bez uklonjenog čvora x
algorithm dlist-delete(x, l)

   if (x.pret != null) then   // x nije prvi čvor
      x.pret.sled = x.sled;
   else
      l = x.sled;
   if (x.sled != null) then   // x nije poslednji čvor
      x.sled.pret = x.pret;

   return l;
```

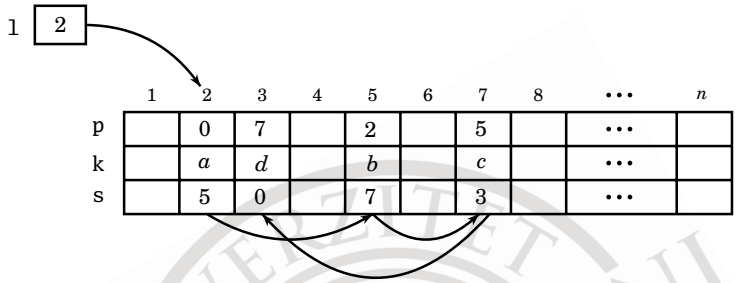
Implementacija liste pomoću nizova

Implementacija liste pomoću pokazivača je vrlo prirodna i jednostavna. Neki programski jezici međutim ne omogućavaju direktan rad sa pokazivačima, pa ćemo u ovom odeljku kratko opisati kako se liste mogu predstaviti pomoću nizova i kako se pokazivači mogu simulirati pomoću indeksa elemenata nizova.[4]

Čvorovi liste (ili opštije bilo koji objekti) sa istim poljima mogu se najjednostavnije predstaviti pomoću posebnog niza za svako polje. Specifično, jednostruko povezana lista se predstavlja pomoću dva niza, a dvostruko povezana lista pomoću tri niza. Kod dvostruko povezane liste na primer, jedan niz, recimo k, sadrži vrednosti polja ključa čvorova liste, a preostala dva niza, recimo s i p, simuliraju polja pokazivača na sledbenika i prethodnika. Na slici 4.9 je prikazano moguće stanje ovih nizova za dvostruku

[4] Reč *simulirati* treba shvatiti uslovno, pošto se u računarima pokazivači zaista realizuju pomoću indeksa (memorijskih adresa) elemenata jednog velikog niza (glavne memorije).

povezanu listu sa slike 4.7. U opštem slučaju, element dvostruko povezane liste se grupno predstavlja elementima tri niza k_i, s_i i p_i na i-toj poziciji. Pri tome, vrednost pokazivača x se predstavlja zajedničkim indeksom elemenata tri niza k_x, s_x i p_x. Indeksi nizova u ovoj interpretaciji se ponekad nazivaju *kursori*.



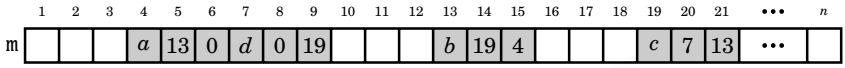
SLIKA 4.9: Reprezentacija dvostruko povezane liste pomoću tri niza.

Element liste a zauzima drugu grupnu poziciju nizova na slici 4.9, pa je $k_2 = a$. Njegov sledbenik u listi, element b, nalazi se u grupnoj poziciji 5, pa zato $k_5 = b$. Pored toga, poredak ova dva elementa liste je određen time što $s_2 = 5$ i $p_5 = 2$. Drugi elementi liste su na sličan način povezani kursorima umesto pokazivačima. Pokazivač `null` se obično predstavlja brojem (na primer, 0 ili -1) koji sigurno ne može biti stvarni indeks nizova. Pokazivač preko koga se pristupa listi predstavlja se običnom promenljivom `l` koja sadrži indeks prvog elementa liste.

Povezane liste možemo predstaviti i pomoću samo jednog niza, ukoliko primenimo metod sličan onom koji se koristi za smeštanje objekata u glavnoj memoriji stvarnih računara. Naime, jedan objekat zauzima susedne lokacije u glavnoj memoriji računara. Pokazivač na neki objekat predstavlja adresu prve memorijske lokacije tog objekta, a polja unutar tog objekta mogu se indeksirati dodavanjem njihove relativne pozicije na početni pokazivač.

Isti princip možemo primeniti za predstavljanje objekata u programskim jezicima koji nemaju ugrađeni pokazivački tip podataka. Na slici 4.10 je prikazano moguće stanje jednog niza m koji se koristi za predstavljanje dvostruko povezane liste sa slike 4.7. Jedan čvor liste zauzima mesto neprekidnog podniza $m_i, \ldots, m_j$ niza m. Pokazivač na ovaj čvor je određen početnim indeksom i podniza, a svako polje ovog čvora je određeno relativnom pozicijom iz opsega 0 do $j - i$. Na primer, relativne pozicije

polja `ključ`, `sled` i `prev` u nekom čvoru na slici 4.10 su 0, 1 i 2. Stoga, za dobijanje vrednosti polja `x.prev` ukoliko je data vrednost pokazivača `x`, treba najpre dodati relativnu poziciju 2 polja `prev` na vrednost pokazivača `x`, a zatim treba pročitati sadržaj elementa niza `m[x+2]`.



SLIKA 4.10: Reprezentacija dvostruko povezane liste pomoću jednog niza.

Prednost upotrebe jednog niza za predstavljanje struktura podataka je u tome što se na taj način mogu predstaviti i kolekcije heterogenih objekata različite veličine. Sa druge strane, rad sa takvom reprezentacijom je komplikovaniji.

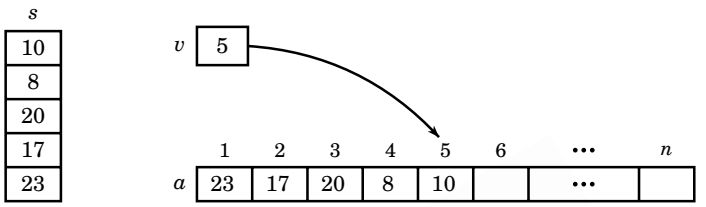
4.4 Stekovi

Stek (engl. *stack*) je specijalna vrsta liste kod koje se dodavanja i uklanjanja elemenata obavljaju samo na jednom kraju koji se naziva *vrh steka*. Intuitivni model strukture podataka steka su naslagani poslužavnici u restoranu ili komadići mesa poređani na ražnjiću. U oba primera je pogodno ukloniti samo onaj element (poslužavnik ili komadić mesa) koji se nalazi na vrhu steka, kao i dodati novi element samo na vrh steka. Drugim rečima, poredak elemenata po kojem se oni uklanjaju sa steka je obrnut onom po kojem su oni dodati na stek. Operacija dodavanja elemenata na stek se tradicionalno zove *push*, a operacija uklanjanja elemenata iz steka zove se *pop*.

Sve implementacije liste o kojima smo govorili u prethodnom odeljku mogu se iskoristiti i za stek, pošto je stek sa svojim operacijama specijalni slučaj liste sa svojim operacijama. U programskom okruženju koje obezbeđuje rad sa pokazivačima, implementacija liste praktično ne mora da se menja za implementaciju steka. Pri tome, operacije za dodavanje elemenata na početak liste i za uklanjanje elemenata sa početka liste su, u stvari, druga imena za operacije *push* i *pop*.

Sa druge strane, za implementaciju steka pomoću nizova može se iskoristiti činjenica da se dodavanja i uklanjanja elemenata vrše samo na vrhu steka. Zato možemo koristiti samo jedan niz, izjednačiti dno steka sa početkom tog niza, a zatim uvećavati stek prema kraju tog niza. Ovaj

pristup je ilustrovan na slici 4.11 na kojoj je levo prikazan logički izgled jednog steka s i desno njegova realizacija pomoću niza a.



SLIKA 4.11: Stek s predstavljen jednim nizom a.

Kao što je prikazano na slici 4.11, stek s od najviše n elemenata se predstavlja nizom a dužine n. Steku se pristupa preko promenljive v čiji sadržaj ukazuje na vrh steka, odnosno na poziciju elementa niza koji je poslednje dodat na stek. Elementi steka zauzimaju pozicije $1,\ldots,v$ u nizu a, pri čemu je a_1 element na dnu steka i a_v element na vrhu steka.

Ukoliko je $v = 0$, stek nema elemenata i onda se kaže da je stek *prazan*. Ako se greškom uklanja element iz praznog steka, takva programska greška se naziva *potkoračenje* steka (engl. *underflow*). Ukoliko je $v = n$, stek je *pun*. Ako se dodaje element na pun stek, dolazi do programske greške *prekoračenja* steka (engl. *overflow*).

Tipične operacije za rad sa stekom, pored tri uobičajene operacije konstruisanja (praznog) steka, dodavanja elementa na vrh steka i uklanjanja elementa sa vrha steka, uključuju i upite da li je stek prazan ili pun kako bi se izbegle greške potkoračenja i prekoračenje steka. U nastavku je data jednostavna algoritamska implementacija ovih operacija nad stekom predstavljenog jednim identifikujućim objektom s koji obuhvata promenljivu v i niz a dužine n.

```
// Ulaz:  stek s
// Izlaz: početno stanje steka s
algorithm stack-make(s)

   s.v = 0;

   return;
```

```
// Ulaz:  stek s
// Izlaz: tačno ako je s prazan, inače netačno
algorithm stack-empty(s)
```

```
   if (s.v == 0)
      return true;
   else
      return false;
```

```
// Ulaz:  stek s
// Izlaz: tačno ako je s pun, inače netačno
algorithm stack-full(s)

   if (s.v == n)
      return true;
   else
      return false;
```

```
// Ulaz:  novi element x, stek s
// Izlaz: stek s sa elementom x na vrhu
algorithm push(x, s)

   // Pretpostavlja se da stek nije pun
   s.v = s.v + 1;
   s.a[s.v] = x;

   return;
```

```
// Ulaz:  stek s
// Izlaz: element x uklonjen sa vrha steka s
algorithm pop(s)

   // Pretpostavlja se da stek nije prazan
   x = s.a[s.v];
   s.v = s.v - 1;

   return x;
```

Čitaoci se lako mogu uveriti da se svi navedeni algoritmi izvršavaju za konstantno vreme.

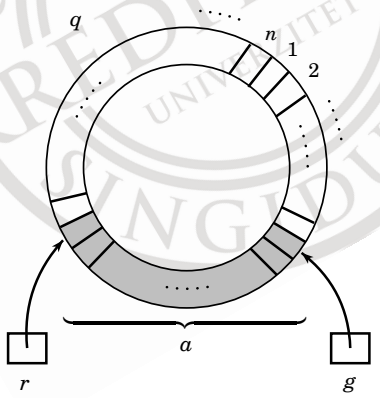
4.5 Redovi za čekanje

Red za čekanje (engl. *queue*) je druga specijalna vrsta povezane liste kod koje se elementi dodaju na jednom kraju liste i uklanjaju sa drugog kraja liste. Mesto na kojem se elementi dodaju logički predstavlja kraj reda, a mesto gde se elementi dodaju označava početak reda. Prema tome, kada

se novi element dodaje u red za čekanje, taj element se uvek priključuje na kraj reda. Sa druge strane, kada se element uklanja iz reda za čekanje, izbacuje se uvek prvi element reda. Intuitivni model za strukturu podataka reda za čekanje je red ljudi koji čeka ispred bioskopa da kupi karte, pri čemu ljudi na početku reda odlaze iz reda pošto kupe karte, a novi ljudi staju na kraj reda da čekaju. (Ali u strukturi podataka reda za čekanje nema „padobranaca”!) Operacija dodavanja elementa u red za čekanje se tradicionalno naziva *enqueue*, a operacija uklanjanja se naziva *dequeue*.

Kao i za stekove, sve implementacije liste mogu se iskoristiti i za redove za čekanje. U programskom okruženju sa pokazivačkim tipom podataka, implementacija reda za čekanje je skoro identična implementaciji liste. Primetimo samo da tu možemo dodatno koristiti pokazivač na kraj reda da bismo operaciju *enqueue* učinili efikasnijom. Na taj način naime, svaki put kada dodajemo novi element ne moramo da prolazimo kroz celu listu od početka do kraja.

Kod implementacije reda za čekanje q pomoću nizova, jedno interesantno rešenje koristi „kružni” niz a veličine n, kao što je to prikazano na slici 4.12.



SLIKA 4.12: Red za čekanje q u obliku „kružnog” niza.

Niz a zamišljamo u kružnom obliku tako da prvi element niza sledi iza poslednjeg. Red za čekanje q se nalazi unutar kruga u susednim lokacijama, pri čemu je kraj reda negde iza početka reda u smeru kretanja kazaljke na satu. Obratite pažnju da se „susedne lokacije” uzimaju u kružnom smislu. Red q ima kursor g koji ukazuje na prvi element (glavu), kao i kursor r koji ukazuje na poslednji element (rep). Na taj način, elementi reda se nalaze u lokacijama $g, g+1, \ldots, r$, koje su „susedne” u smislu da

lokacija 1 sledi odmah iza lokacije n u kružnom redosledu.

Kada se dodaje neki element, kursor r se uvećava za jedan i novi element se upisuje u poziciju na koju ukazuje kursor. Kada se neki element uklanja, kursor g se samo uvećava za jedan. Prema tome, kako se elementi dodaju i uklanjaju, ceo red za čekanje se tokom vremena pomera u nizu u smeru kretanja kazaljke na satu.

U reprezentaciji reda za čekanje koja je prikazana na slici 4.12, kao i u manjim varijacijama na tu temu (na primer, kada kursor r ukazuje na prvu lokaciju iza poslednjeg elementa, a ne na sâm poslednji element), nailazimo na jednu teškoću. Naime, problem je da ne možemo razlikovati prazan od punog reda, ukoliko eksplicitno ne pratimo dužinu reda. (Zainteresovani čitaoci se lako mogu uveriti da je ovo zaista problem.)

Zato se obično koristi dodatna promenljiva, recimo l, koja sadrži aktuelni broj elemenata reda u nizu a. Drugi način za rešavanje ovog problema je ne dozvoliti da red ima više od $n-1$ elemenata, ali imati jedan neupotrebljiv element u nizu izgleda manje prirodno nego imati jednu promenljivu za praćenje dužine reda.

Prema tome, red za čekanje se predstavlja identifikujućim objektom q koji obuhvata tri promenljive g, r i l, čija je uloga prethodno objašnjena, kao i niz a dužine n za smeštanje elemenata reda. Ako se koristi sabiranje po modulu za realizaciju kružnog izgleda niza a, algoritamska implementacija operacije nad redom za čekanje, koja je data u nastavku, nije uopšte komplikovana. Čitaoci se lako mogu uveriti da se svi navedeni algoritmi izvršavaju za konstantno vreme.

```
// Ulaz:  red q
// Izlaz: početno stanje reda q
algorithm queue-make(q)

   q.g = 1;
   q.r = n;
   q.l = 0;

   return;
```

```
// Ulaz:  red q
// Izlaz: tačno ako je q prazan, inače netačno
algorithm queue-empty(q)

   if (q.l == 0)
      return true;
   else
```

```
      return false;
```

```
// Ulaz:  red q
// Izlaz: tačno ako je q pun, inače netačno
algorithm queue-full(q)

   if (q.l == n)
      return true;
   else
      return false;
```

```
// Ulaz:  novi element x, red q
// Izlaz: red q sa elementom x na kraju
algorithm enqueue(x, q)

   // Pretpostavlja se da red nije pun
   q.r = q.r % n + 1;
   q.a[q.r] = x;
   q.l = q.l + 1;

   return;
```

```
// Ulaz:  red q
// Izlaz: element x uklonjen iz reda q na početku
algorithm dequeue(q)

   // Pretpostavlja se da red nije prazan
   x = q.a[q.g];
   q.g = q.g % n + 1;
   q.l = q.l - 1;

   return x;
```

Zadaci

1. *Virtuelna inicijalizacija niza.* Ako je dat niz $a_1,\ldots,a_n$ dužine n, naivni način za inicijalizaciju svih elemenata niza nekom vrednošću x je:

```
algorithm array-init(a, n, x)

   for i = 1 to n do
      a[i] = x;

   return a;
```

Ovaj algoritam očigledno zahteva vreme $O(n)$ za izvršavanje i konstantni memorijski prostor (pored prostora potrebnog za sam niz). Pri tome, obe operacije za pristup nekom elementu niza a koje su realizovane algoritmima `array-read(a,i)`, kojim se čita vrednost i-tog elementa niza a, i `array-write(a,i,x)`, kojim se upisuje vrednost x u i-ti element niza a, izvršavaju se za konstantno vreme:

```
algorithm array-read(a, i)

   return a[i];
```

```
algorithm array-write(a, i, x)

   a[i] = x;

   return a;
```

U nekim primenama je linearno vreme za inicijalizaciju niza neprihvatljivo dugo, pa je potrebno „virtuelno" inicijalizovati niz za konstantno vreme. To je posebno značajno u slučajevima kada se tokom rada programa koristi zapravo samo mali deo nekog velikog niza. Napišite algoritme `array-init(a,n,x)`, `array-read(a,i)` i `array-write(a,i,x)` koji rade tačno ono što treba, ali svaki od njih se izvršava za konstantno vreme. (*Savet:* Koristite dva dodatna niza veličine n, pri čemu jedan od njih simulira stek.)

2. Pokažite da je ispravna ova verzija linearnog algoritma `mcss` koja ima samo jednu petlju:

```
// Ulaz:  niz a, broj elemenata n niza a
// Izlaz: maksimalna suma podniza M
algorithm mcss(a, n)

   M = 0;   // maksimalna suma podniza
   s = 0;
   for j = i to n do
      s = s + a[j];
      if (s < 0) then
         s = 0;
      else if (M < s) then
         M = s;

   return M;
```

3. Ako su date sledeće liste naklonosti mladića i devojaka, pronađite sva moguća stabilna vezivanja. (*Savet:* Ima ih tačno pet.)

mladići	1	2	3	4	5
X	x	y	z	u	v
Y	y	z	u	v	x
Z	z	u	v	x	y
U	u	v	x	y	z
V	v	x	y	z	u

devojke	1	2	3	4	5
x	Y	Z	U	V	X
y	Z	U	V	X	Y
z	U	V	X	Y	Z
u	V	X	Y	Z	U
v	X	Y	Z	U	V

4. Pokažite da se u algoritmu `smp` zaista izvršava skoro n^2 iteracija u nekim instancama problema stabilnih brakova. (*Savet:* Uzmite da je lista naklonosti svakog od n mladića u obliku $1,2,\ldots,n-1,n$, uzmite da je lista naklonosti svake od n devojaka u obliku $n,n-1,\ldots,2,1$, i pretpostavite da se slobodni mladići biraju kružnim redom $1,2,\ldots,n$ i zatim opet od 1, kao da se nalaze u redu za čekanje.)

5. Pokažite da je Gejl-Šejpli algoritam „polno nefer" jer favorizuje mladiće (kao i da je suprotno tačno ukoliko mladići i devojke zamene uloge tako što devojke prose mladiće i mladići biraju bolje ponude). Specifično, navedite primer lista naklonosti mladića i devojaka tako da je dobijeno stabilno vezivanje najbolje moguće za mladiće i najgore moguće za devojke.

6. Koristeći strukture podataka iz ovog poglavlja, napišite detaljniji algoritam `smp` na pseudo jeziku. Još bolje, na svom omiljenom programskom jeziku napišite i testirajte pravi program za rešavanje problema stabilnih brakova. (*Savet:* Koristite strukture podataka koje su spomenute kod analize vremena izvršavanja algoritma `smp`.)

7. Napišite algoritam `list-length(l)` kojim se izračunava dužina liste l pretpostavljajući da su liste implementirane pomoću pokazivača. Analizirajte vreme izvršavanja svog algoritma.

8. Napišite algoritam `list-copy(l1,l2)` kojim se konstruiše lista l_2 tako da bude identična kopija liste l_1, a lista l_1 ostaje nepromenjena. Pretpostavite da su liste implementirane pomoću pokazivača i analizirajte vreme izvršavanja svog algoritma.

9. Napišite algoritam `list-concat(l1,l2,l3)` kojim se spajaju (konka-

teniraju) dve liste l_1 i l_2 u treću listu l_3. Pretpostavite da su liste implementirane pomoću pokazivača i analizirajte vreme izvršavanja svog algoritma.

10. Napišite algoritam `list-deldup(l)` kojim se uklanjaju svi duplikati iz liste l. Pretpostavite da su liste implementirane pomoću pokazivača i analizirajte vreme izvršavanja svog algoritma.

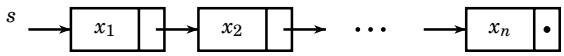
11. Napišite algoritam `dlist-insert(x,l,p)` kojim se čvor x dodaje iza čvora p u dvostruko povezanu listu l. Pretpostavite da su liste implementirane pomoću pokazivača i analizirajte vreme izvršavanja svog algoritma.

12. Jedan niz se može iskoristiti za predstavljanje dva steka, jedan koji raste udesno od početka niza i drugi koji raste ulevo od kraja niza. Implementirajte operacije nad stekom koristeći ovu reprezentaciju dva steka pomoću jednog niza. Analizirajte vreme izvršavanja tih operacija.

13. Pokažite da se red za čekanje može implementirati pomoću dva steka, odnosno napišite algoritme `enqueue(q)`, `dequeue(q)` i `queue-empty(q)` koristeći jedino operacije *push*, *pop* i *stack-empty* nad dva steka koji predstavljaju jedan red za čekanje. Analizirajte vreme izvršavanja tih operacija nad redom za čekanje.

14. Pokažite da se stek može implementirati pomoću dva reda za čekanje, odnosno napišite algoritme `push(x,s)`, `pop(s)` i `stack-empty(s)` koristeći jedino operacije *enqueue*, *dequeue* i *queue-empty* nad dva reda za čekanje koji predstavljaju jedan stek. Analizirajte vreme izvršavanja tih operacija nad stekom.

15. *Skup* kao struktura podataka je konačna kolekcija *elemenata* (ili *članova*) istog tipa. Dve konceptualne osobine skupa su da su svi njegovi elementi međusobno različiti i da ne postoji nikakva relacija uređenosti među elementima skupa. Jedan lak način za predstavljanje skupa je pomoću povezane liste njegovih elemenata. Na primer, na slici 4.13 je prikazan skup $S = \{x_1, x_2, \ldots, x_n\}$ predstavljen povezanom listom s.



SLIKA 4.13: Skup $S = \{x_1, x_2, \ldots, x_n\}$ predstavljen listom s.

Tipične skupovne operacije su dodavanje datog elementa skupu, uklanjanje datog elementa iz skupa i provera da li dati element pripada skupu.

a) Pretpostavljajući implementaciju skupa pomoću liste njegovih elemenata, napišite algoritme `set-insert(x,s)`, `set-delete(x,s)` i `set-member(x,s)` kojima se realizuju prethodne skupovne operacije. Za sve algoritme odredite vreme njihovog izvršavanja.

b) Ako se elementi skupa mogu međusobno upoređivati, druga mogućnost za predstavljanje skupa pomoću liste je da se skupovni elementi u listi smeštaju u rastućem redosledu. Ponovite prethodni zadatak pretpostavljajući ovakvu implementaciju skupa pomoću sortirane liste njegovih elemenata.

16. Napišite algoritme za operacije unije i preseka dva skupa pretpostavljajući implementaciju skupova pomoću a) nesortiranih listi i b) sortiranih listi.

UNIVERZITET
PRIVATNI
AKREDITOVANI
UNIVERZITET
SINGIDUNUM

GLAVA

5

Rekurzivni algoritmi

U rekurzivnom algoritmu se do rešenja problema dolazi pristupom „od opšteg ka pojedinačnom". To se neformalno može opisati na sledeći način: dati problem se rešava tako što se najpre podeli u nekoliko manjih potproblema, zatim se nezavisno rešava svaki od tih potproblema i, na kraju, kombinovanjem njihovih rešenja se dolazi do rešenja polaznog problema. Ako su manji potproblemi slični polaznom problemu, njihova rešenja se mogu dobiti ponovnom (rekurzivnom) primenom istog postupka. Naravno, postupak deljenja u manje probleme se mora završiti posle konačno mnogo koraka i dobiti trivijalni potproblemi čija se rešenja mogu neposredno dobiti bez daljeg deljenja. U suprotnom slučaju dolazi do programske greške, jer bi rekurzivni algoritam nastavio proces deljenja potproblema u beskonačnost i ne bi došao do rešenja.

Rekurzivni način rešavanja problema u programiranju je vrlo važan i efikasan metod za rešavanje složenih problema. Takvi problemi se često mogu mnogo prirodnije i jednostavnije rešiti primenom rekurzije nego klasičnim, iterativnim putem. Pošto rekurzivno rešavanje problema zahteva drugačiji pristup u načinu programiranja, u ovom poglavlju ćemo ilustrovati osnovne principe „rekurzivnog razmišljanja" kroz brojne reprezentativne primere. Pored toga, analiza rekurzivnih algoritama se ne sastoji od pukog prebrojavanja jediničnih instrukcija, nego se zasniva na rešavanju tzv. rekurentnih jednačina. O tome će takođe biti reči u ovom poglavlju.

5.1 Iterativni i rekurzivni algoritmi

Rekurzivni metod je alternativni oblik programske kontrole u odnosu na petlje, ali se takođe sastoji od ponavljanja nekog istog postupka. Kada se u iterativnom algoritmu koristi petlja, postupak koji se ponavlja navodi se u formi tela petlje, a broj ponavljanja tela petlje kontroliše se uslovom prekida (nastavka) petlje. Kod rekurzivnih algoritama, postupak koji se ponavlja je sâm algoritam koji sam sebe poziva radi višestrukog izvršavanja. Broj ponavljanja tog postupka kontroliše se naredbom grananja u rekurzivnom algoritmu kojom se određuje da li isti algoritam treba pozvati ili ne. U suštini dakle, rekurzivni i iterativni algoritmi predstavljaju različite zapise za opis višestrukog ponavljanja jednog istog postupka.

Svaki problem koji se može rešiti rekurzivno može se rešiti i iterativno, kao i obrnuto. Obratite ipak pažnju na to da postupak koji se ponavlja u rekurzivnom i iterativnom algoritmu za isti problem ne mora biti identičan.

Razmotrimo, na primer, izračunavanje stepena x^n, gde je x neki realan broj različit od nule i n ceo broj veći ili jednak nuli. Iterativni način za rešenje ovog problema sastoji se od uzastopnog množenja broja x sa parcijalno izračunatim stepenima x^i za $i = 0, 1, \ldots, n-1$. Naredni iterativni algoritam power1 koristi for petlju radi realizacije ovog postupka:

```
// Ulaz:  realan broj x, ceo broj n ≥ 0
// Izlaz: broj x^n
algorithm power1(x, n)

   y = 1;
   for i = 1 to n do
      y = x * y;

   return y;
```

Pored ovog klasičnog iterativnog načina, za izračunavanje stepena možemo primeniti drugačiji, rekurzivni pristup ukoliko uočimo da je $x^0 = 1$ za stepen $n = 0$ i $x^n = x \cdot x^{n-1}$ za stepen n veći od nule. Drugim rečima, ako je stepen $n = 0$, rezultat je uvek 1, dok za stepen $n > 0$ rezultat dobijamo ako x pomnožimo sa $(n-1)$-im stepenom broja x. Naredni rekurzivni algoritam power2 koristi ovaj način za izračunavanje n-tog stepena broja x:

```
// Ulaz:  realan broj x, ceo broj n ≥ 0
// Izlaz: broj x^n
algorithm power2(x, n)
```

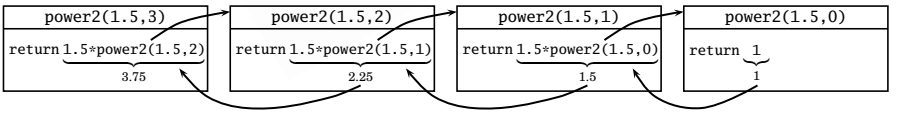
```
if (n == 0)
   return 1;
else
   return x * power2(x,n-1);
```

Obratite pažnju na to da, za izračunavanje n-tog stepena broja x, rekurzivni algoritam power2 u drugoj naredbi return poziva sam sebe radi izračunavanja $(n-1)$-og stepena broja x.

Rekurzivni algoritam je algoritam koji poziva sam sebe, bilo direktno ili indirektno. Algoritam direktno poziva sam sebe (kao recimo power2) ukoliko se u njegovoj definiciji nalazi poziv samog algoritma koji se definiše. Algoritam indirektno poziva sebe ukoliko se u njegovoj definiciji nalazi poziv drugog algoritma koji sa svoje strane poziva polazni algoritam koji se definiše (bilo direktno ili indirektno).

Mehanizam pozivanja rekurzivnih algoritama se ne razlikuje od standardnog mehanizma za obične algoritme: argumenti u pozivu algoritma se najpre dodeljuju parametrima algoritma kao početne vrednosti i zatim se izvršava telo pozvanog algoritma. Međutim, ovde je važno primetiti da poziv rekurzivnog algoritma generiše lanac poziva istog algoritma za rešavanje niza prostijih zadataka sve dok se ne dođe do najprostijeg zadatka čije je rešenje očigledno. Tada se lanac rekurzivnih poziva prekida i započeti pozivi se završavaju jedan za drugim u obrnutom redosledu njihovog pozivanja, odnosno složenosti zadataka koje rešavaju (od najprostijeg zadatka ka složenijim zadacima). Na taj način, na kraju se dobija rešenje najsloženijeg polaznog zadatka.

Na primer, na slici 5.1 je ilustrovano rekurzivno izvršavanje poziva power2(1.5,3) kojim se izračunava $1.5^3 = 3.75$.



SLIKA 5.1: Lanac rekurzivnih poziva za power2(1.5,3).

Na slici 5.1 gornje strelice pokazuju da poziv power2(1.5,3) najpre proizvodi drugi poziv power2(1.5,2); ovaj drugi poziv zatim proizvodi treći poziv power2(1.5,1); i na kraju, treći poziv dovodi do poslednjeg poziva power2(1.5,0). U ovom lancu jedan poziv dovodi do drugog poziva jer je u svim slučajevima argument poziva za stepen različit od nule. Zato

se u telu algoritma power2 izvršava deo else naredbe if, odnosno naredba return u kojoj se argument za broj x množi sa odgovarajućim rezultatom poziva algoritma. Tek u poslednjem pozivu je argument za stepen jednak nuli, pa se izvršavanjem dela then naredbe if kao rezultat dobija broj 1. Nakon toga se lanac rekurzivnih poziva „odmotava" u obrnutom smeru, kako to pokazuju donje strelice na slici 5.1. Pri tome se izvršavanje aktuelnog poziva sastoji od množenja rezultata odgovarajućeg završenog poziva sa vrednošću parametra x i dobijena vrednost se kao rezultat vraća prethodnom pozivu.

Generalno, rekurzivni algoritmi rešavaju neki zadatak svođenjem tog zadatka na sličan prostiji zadatak (ili više njih). Na primer, u algoritmu power2 se problem izračunavanja n-tog stepena nekog broja rešava svođenjem na problem izračunavanja $(n-1)$-og stepena istog broja, koji je prostiji u smislu da je stepen broja koji se izračunava za jedan manji. Prostiji zadatak se zatim rešava rekurzivnim pozivom algoritma koji se definiše. To sa svoje strane dovodi do svođenja tog prostijeg zadatka na još prostiji zadatak, a ovaj se onda opet rešava rekurzivnim pozivom. Na primer, u algoritmu power2 se problem izračunavanja $(n-1)$-og stepena broja svodi na problem izračunavanja $(n-2)$-og stepena istog broja.

Ovaj postupak uprošćavanja polaznog zadatka se i dalje slično nastavlja, pa bi se bez ikakve kontrole dobio beskonačan lanac poziva istog algoritma, što je programska greška slična beskonačnoj petlji. Da bi se lanac rekurzivnih poziva prekinuo na kraju, u svakom rekurzivnom algoritmu mora se nalaziti *bazni slučaj* za najprostiji zadatak čije je rešenje unapred poznato i koje se ne dobija daljim rekurzivnim pozivima. To je u algoritmu power2 slučaj kada je stepen $n = 0$, jer se onda rezultat 1 neposredno dobija bez daljeg rekurzivnog rešavanja.

Radi boljeg razumevanja ovih osnovnih elemenata rekurzivne tehnike programiranja, u nastavku poglavlja ćemo pokazati još neke primere rekurzivnih algoritama.

Primer: rekurzivni Euklidov algoritam

U odeljku 3.2 na strani 60 pokazali smo efikasan iterativni (Euklidov) algoritam za izračunavanje najvećeg zajedničkog delioca $\text{nzd}(x, y)$ dva pozitivna cela broja x i y. Podsetimo se da se taj iterativni algoritam zasniva na činjenici da je $\text{nzd}(x, y) = \text{nzd}(y, x\%y)$ za $x \geq y$. Ista činjenica se može iskoristiti i za (efikasan) rekurzivni algoritam za izračunavanje $\text{nzd}(x, y)$ ukoliko to izračunavanje podelimo u dva slučaja:

- bazni slučaj: ako je $x \% y = 0$, onda je očigledno $\text{nzd}(x, y) = y$;
- opšti slučaj: ako je $x \% y \neq 0$, onda je $\text{nzd}(x, y) = \text{nzd}(y, x \% y)$.

Prema tome, $\text{nzd}(x, y)$ se može rekurzivno definisati na sledeći način:

$$\text{nzd}(x, y) = \begin{cases} y, & \text{ako je } x \% y = 0 \\ \text{nzd}(y, x \% y), & \text{ako je } x \% y \neq 0. \end{cases}$$

Na osnovu ove rekurzivne definicije sada je lako napisati rekurzivni Euklidov algoritam za izračunavanje $\text{nzd}(x, y)$:

```
// Ulaz:  celi brojevi x i y takvi da x ≥ y > 0
// Izlaz: nzd(x,y)
algorithm euclid(x, y)

   if (x % y == 0) then  // bazni slučaj
      return y;
   else                  // opšti slučaj
      return nzd(y,x % y);
```

U ovom algoritmu u drugoj naredbi `return` primetimo da su u rekurzivnom pozivu $\text{nzd}(y, x \% y)$ oba argumenta manja od polaznih brojeva x i y. To znači da se tim pozivom zaista rešava prostiji zadatak izračunavanja najvećeg zajedničkog delioca za par manjih brojeva. Ostatak pri celobrojnom deljenju u lancu rekurzivnih poziva se dakle stalno smanjuje i zato u jednom trenutku mora biti nula. Tada će se lanac prekinuti, jer se onda rezultat neposredno dobija kao vrednost drugog argumenta poslednjeg poziva u lancu.

Primer: Fibonačijev niz brojeva

Fibonačijev niz brojeva ima brojne primene u umetnosti, matematici i računarstvu. Prva dva broja Fibonačijevog niza brojeva su 1 i 1, a svaki sledeći broj u nizu se zatim dobija kao zbir prethodna dva broja niza. Tako, treći broj niza je zbir drugog i prvog, odnosno $1 + 1 = 2$; četvrti broj niza je zbir trećeg i drugog, odnosno $2 + 1 = 3$; peti broj niza je zbir četvrtog i trećeg, odnosno $3 + 2 = 5$; i tako dalje. Prema tome, početni deo Fibonačijevog niza brojeva je:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

Za izračunavanje n-tog broja Fibonačijevog niza može se početi od prva dva broja tog niza i zatim primeniti iterativni postupak u kojem se u svakom koraku redom izračunava naredni broj u nizu dok se ne dobije traženi

n-ti broj. U nastavku je predstavljen iterativni algoritam u kojem se koriste tri promenljive koje u svakom koraku sadrže prethodna dva broja niza (promenljive x i y) i naredni broj niza (promenljiva z):

```
// Ulaz:  n ≥ 1
// Izlaz: n-ti broj Fibonačijevog niza
algorithm fib1(n)

   x = 1; y = 1; z = 1;
   for i = 3 to n do
      z = x + y;
      x = y;
      y = z;

   return z;
```

Ako brojeve Fibonačijevog niza označimo sa $f_1, f_2, f_3, \ldots$, onda je očevidno rekurzivna definicija za n-ti broj Fibonačijevog niza:

$$f_1 = 1, f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2}, \quad n > 2.$$

Drugim rečima, prva dva broja f_1 i f_2 su jednaki 1, a n-ti broj za $n > 2$ je jednak zbiru prethodna dva, tj. $(n-1)$-og i $(n-2)$-og, broja Fibonačijevog niza.

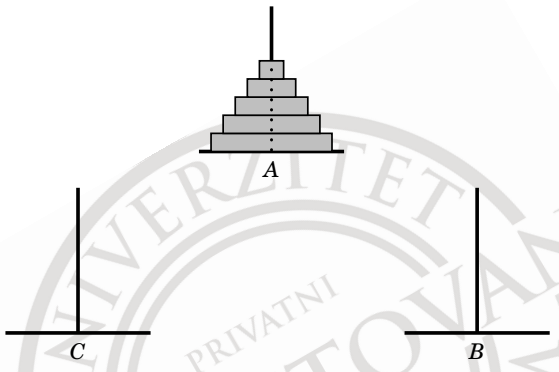
Za izračunavanje n-tog broja Fibonačijevog niza možemo lako napisati rukurzivni algoritam ukoliko iskoristimo prethodnu rekurzivnu definiciju tog broja:

```
// Ulaz:  n ≥ 1
// Izlaz: n-ti broj Fibonačijevog niza
algorithm fib2(n)

   if (n <= 2) then  // bazni slučaj
      return 1;
   else              // opšti slučaj
      return fib(n-1) + fib(n-2);
```

U ovom algoritmu primetimo da se zadatak izračunavanja n-tog broja Fibonačijevog niza svodi na dva slična prostija zadatka, odnosno izračunavanje $(n-1)$-og i $(n-2)$-og broja Fibonačijevog niza. Zato se pomoću dva rekurzivna poziva `fib(n-1)` i `fib(n-2)` izračunavaju vrednosti $(n-1)$-og i $(n-2)$-og broja Fibonačijevog niza, a njihov zbir se kao rezultat vraća za n-ti broj Fibonačijevog niza.

Primer: Hanojske kule

Problem Hanojskih kula je jednostavna igra u kojoj su na početku n diskova različite veličine poređani na jednom od tri stuba u opadajućem redosledu svoje veličine od podnožja tog stuba ka vrhu. Na slici 5.2 je prikazana početna konfiguracija ove igre za $n = 5$ diskova ukoliko zamislimo da su tri stuba trougaono raspoređeni.



SLIKA 5.2: Početna konfiguracija igre Hanojskih kula.

Cilj igre je da se svi diskovi premeste sa početnog stuba na drugi stub koristeći treći stub kao pomoćni. Pri tome, samo disk sa vrha jednog stuba se može premestiti na vrh drugog stuba i nikad se veći disk ne sme postaviti iznad manjeg diska.

Preciznije rečeno, igra Hanojskih kula se sastoji od tri stuba A, B i C, kao i od n diskova različite veličine prečnika $d_1 < d_2 < \cdots < d_n$. Svi diskovi su početno poređani na stubu A u opadajućem redosledu svoje veličine od podnožja stuba A, odnosno disk najvećeg prečnika d_n se nalazi na dnu i disk najmanjeg prečnika d_1 se nalazi na vrhu stuba A. Cilj igre je premestiti sve diskove na stub C koristeći stub B kao pomoćni i poštujući dva pravila:

- Samo se disk sa vrha jednog stuba može premestiti na vrh drugog stuba.
- Nikad se veći disk ne može nalaziti iznad manjeg diska.

Primetimo da ova pravila impliciraju da se samo po jedan disk može premeštati, kao i da se svi diskovi u svakom trenutku moraju nalaziti samo na stubovima.

Bez mnogo objašnjenja navodimo rešenje igre koje se dobija iterativnim postupkom korak po korak: u svakom neparnom koraku, premestiti najmanji disk na naredni disk u smeru kretanja kazaljke na satu; u svakom parnom koraku, legalno premestiti jedini mogući disk koji nije najmanji. Ovaj postupak je ispravan, kao što se u to pažljivi čitaoci mogu sami uveriti, ali je teško razumeti zašto je to tako i još je teže doći do njega.

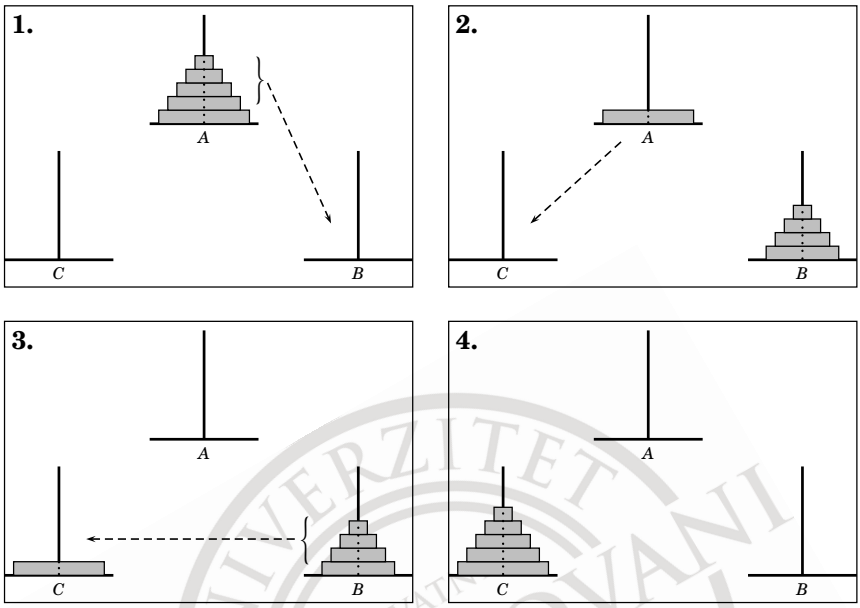
Igra Hanojskih kula je primer problema čije se rešenje mnogo lakše i prirodnije dobija rekurzivnim postupkom. Zadatak premeštanja n diskova sa stuba A na stub C koristeći B kao pomoćni stub možemo svesti na dva slična prostija zadatka premeštanja $(n-1)$-og diska. Prvi zadatak je premeštanje $n-1$ najmanjih diskova sa stuba A na stub B koristeći C kao pomoćni stub. Drugi zadatak je premeštanje $n-1$ najmanjih diskova sa stuba B na stub C koristeći A kao pomoćni disk. (Ovo su prostiji zadaci u smislu da se rešava sličan problem za manji broj diskova.) Polazni problem za n diskova onda možemo rešiti rekurzivnim postupkom koji se sastoji od tri koraka:

1. Primenjujući rekurzivni postupak za rešenje prvog zadatka, najpre premeštamo $n-1$ najmanjih diskova sa stuba A na stub B koristeći stub C kao pomoćni.
2. Na taj način nam ostaje samo najveći disk na stubu A. Zatim premeštamo taj najveći disk sa stuba A na slobodni stub C.
3. Na kraju, primenjujući isti rekurzivni postupak za rešenje drugog zadatka, svih $n-1$ najmanjih diskova koji se nalaze na stubu B premeštamo na stub C koristeći stub A kao pomoćni.

Ovaj rekurzivni postupak je ilustrovan na slici 5.3. Na toj slici su prikazani rezultati prethodna tri koraka od početne konfiguracije za $n = 5$ diskova.

Obratite pažnju na to koliko je rekurzivno rešenje problema Hanojskih kula konceptualno jednostavnije za razumevanje. Možda još važnije, verovatno su i sami čitaoci relativno lako došli do njega. Ova jednostavnost rekurzivnog postupka je prevashodno ono što čini rekurzivnu tehniku važnom, mada su u mnogim slučajevima dobijeni rekurzivni algoritmi i efikasniji od klasičnih.

Pretpostavimo da na raspolaganju imamo algoritam `move(x,y)` kojim se jedan disk na vrhu stuba `x` premešta na vrh diskova na stubu `y`. Ovaj algoritam je trivijalan, ali ga ne navodimo jer njegova implementacija veoma zavisi od konkretnog programskog jezika. Onda se sledećim jednostavnim



SLIKA 5.3: Rekurzivno rešenje problema Hanojskih kula.

rekurzivnim algoritmom određuje niz poteza kojima se premeštaju n diskova sa stuba A na stub C koristeći B kao pomoćni stub:[1]

```
// Ulaz:  n ≥ 1, početni, pomoćni i ciljni stub
// Izlaz: niz poteza za n diskova problema Hanojskih kula
algorithm toh(n, a, b, c)

   if (n == 1)  // bazni slučaj
      move(a,c);
   else         // opšti slučaj
      toh(n-1,a,c,b);
      move(a,c);
      toh(n-1,b,a,c);

   return;
```

[1]Ime algoritma toh je skraćenica od engleskog naziva za igru Hanojskih kula: *the Towers of Hanoi*.

5.2 Analiza rekurzivnih algoritama

U poglavlju 3 smo pokazali kako se analiziraju iterativni algoritmi i objasnili smo da se to svodi, u suštini, na prebrojavanje jediničnih instrukcija koje se izvode za vreme izvršavanja takvih algoritama. Kod rekurzivnih algoritama je dinamika njihovog izvršavanja složenija i puko prebrojavanje jediničnih instrukcija može biti prilično komplikovano. Zbog toga se vreme izvršavanja rekurzivnih algoritama određuje na drugačiji način i predstavlja se u obliku *rekurentnih jednačina*. Te jednačine opisuju ukupno vreme izvršavanja rekurzivnog algoritma za ulaz veličine n na osnovu vremena izvršavanja istog algoritma za ulaze veličine manje od n. Pošto za rešavanje rekurentnih jednačina postoji razvijen matematički aparat, time se na relativno lak način mogu dobiti i ocene performansi rekurzivnih algoritama.

Radi ilustracije, razmotrimo algoritam `toh` iz prethodnog odeljka za rešavanje problema Hanojskih kula i neka je $T(n)$ njegovo vreme izvršavanja za n diskova. (Čitaoci bi do sada trebalo da prepoznaju da je veličina ulaza za ovaj problem predstavljena brojem diskova.) Nije se teško ubediti da se algoritam `move` u svakoj konkretnoj realizaciji izvršava za konstantno vreme, pa ga radi jednostavnosti možemo smatrati jediničnom instrukcijom. Kako se u algoritmu `toh` za $n = 1$ izvršava samo algoritam `move`, to je $T(1) = 1$. Za $n > 1$ se u algoritmu `toh` izvršavaju dva rekurzivna poziva istog algoritma za $n - 1$ disk, kao i algoritam `move`, pa je u tom slučaju ukupno vreme $T(n) = 2T(n-1) + 1$. Vreme izvršavanja algoritma `toh` može se dakle kraće zapisati u obliku jednačine:

$$T(n) = \begin{cases} 1, & \text{ako je } n = 1 \\ 2T(n-1) + 1, & \text{ako je } n > 1. \end{cases} \tag{5.1}$$

Ovo je primer *rekurentne jednačine* u kojoj se vrednost funkcije $T(n)$ definiše tako što se navodi početni uslov za određenu vrednost argumenta n, a za druge vrednosti argumenta n se $T(n)$ definiše na osnovu vrednosti iste funkcije kada je argument manji od n. Na primer, u konkretnom slučaju rekurentne jednačine (5.1) za vreme izvršavanja algoritma `toh`, početni uslov je $T(1) = 1$, dok se $T(n)$ za $n > 1$ predstavlja izrazom u kojem učestvuje vrednost za $T(n-1)$.

Rešavanje rekurentne jednačine u opštem slučaju sastoji se u određivanju zatvorenog izraza za $T(n)$ u funkcijskom obliku samo od argumenta n, bez ijednog pojavljivanja vrednosti $T(m)$ za neko $m < n$. Jedan način za

dobijanje rešenja neke rekurentne jednačine je primena tzv. postupka višestruke zamene u kojem se sukcesivno zamenjuje opšti izraz za $T(n)$ umesto vrednosti $T(m)$ za neko $m < n$ na desnoj strani rekurentne jednačine, sve dok se ne eliminišu sve takve vrednosti i ne dobije početni uslov.

Na primer, ukoliko se primeni ovaj postupak na rekurentnu jednačinu (5.1) za vreme izvršavanja algoritma `toh`, opšti izraz za $T(n)$ treba zameniti umesto $T(n-1), T(n-2), \ldots, T(2)$ na desnoj strani rekurentne jednačine (5.1) za $n > 1$. Na kraju treba zameniti početni uslov $T(1)$ i pojednostaviti rezultujuću formulu. Na taj način se dobija sledeće rešenje za $T(n)$:

$$\begin{aligned}
T(n) &= 2 \cdot T(n-1) + 1 \\
&= 2 \cdot \big(2T(n-2) + 1\big) + 1 = 2^2 \cdot T(n-2) + 2 + 1 \\
&= 2^2 \cdot \big(2T(n-3) + 1\big) + 2 + 1 = 2^3 \cdot T(n-3) + 2^2 + 2 + 1 \\
&\vdots \\
&= 2^{n-1} \cdot T(1) + 2^{n-2} + \cdots + 2^2 + 2 + 1 \\
&= 2^{n-1} \cdot 1 + \sum_{i=0}^{n-2} 2^i = 2^{n-1} + \frac{2^{n-1} - 1}{2 - 1} = 2 \cdot 2^{n-1} - 1 \\
&= 2^n - 1 .
\end{aligned}$$

Primetimo da je rešenje $T(n) = 2^n - 1$ zapravo broj poteza u rešenju problema Hanojskih kula za n diskova, jer izvršavanje algoritma `move` predstavlja upravo jedan potez u rešenju.

Da bismo bolje uvežbali rekurzivni način rešavanja problema i analizu odgovarajućih rekurzivnih algoritama, u nastavku ćemo pokazati još neke primere rekurzivne tehnike.

Primer: sortiranje objedinjavanjem (*merge-sort*)

Mada smo problem sortiranja definisali za nizove brojeva, algoritam za njegovo rešavanje o kome govorimo u ovom odeljku se najbolje može objasniti preko liste brojeva. Ne postoje suštinske prepreke za primenu tog algoritma na nizove, nego je prosto lakše opisati i razumeti algoritam na apstraktnijem nivou liste brojeva. Naime, u slučaju nizova bismo morali da nepotrebno posvetimo pažnju nevažnim detaljima manipulisanja indeksa nizova, što nije slučaj kod listi.

Metod sortiranja neuređene (jednostruko povezane) liste brojeva, koji se popularno naziva *sortiranje objedinjavanjem* (engl. *merge-sort*), zasniva

se na tri glavna koraka. U prvom koraku se najpre deli data lista l od n brojeva u dve manje liste l_1 i l_2 od otprilike $n/2$ elemenata u svakoj. U drugom koraku se zatim rekurzivno sortiraju ove dve polovične liste posebno. Na kraju, u trećem koraku se dve sortirane polovične liste dobijene u drugom koraku objedinjuju u završnu sortiranu celu listu. Sada ćemo ove korake redom detaljnije objasniti.

Da bismo datu listu l podelili u dve polovične liste l_1 i l_2 u prvom koraku, redom uklanjamo glavu date liste i dodajemo je naizmenično na početak prve ili druge liste. Ako iskoristimo operacije za rad sa listom iz odeljka 4.3, realizacija ovog postupka na pseudo jeziku je:

```
// Ulaz:  lista l
// Izlaz: liste l_1 i l_2 formirane od naizmeničnih elemenata liste l
algorithm list-split(l, l1, l2)

   l1 = null; l2 = null;
   pd = true;  // indikator dodavanja prvoj ili drugoj listi

   while (l != null) do
      x = l;       // uklanjanje glave liste l
      l = list-delete(null,l);
      if (pd) then  // dodavanje glave liste l na početak liste l_1
         l1 = list-insert(x,null,l1);
      else         // dodavanje glave liste l na početak liste l_2
         l2 = list-insert(x,null,l2);
      pd = !pd;

   return l1, l2;
```

Objediniti dve sortirane liste u trećem koraku sortiranja objedinjavanjem znači proizvesti jedinstvenu sortiranu listu koja sadrži sve elemente datih listi (i nijedan drugi element). Na primer, ako su date dve sortirane liste $\langle 1,2,3,7,8\rangle$ i $\langle 1,2,6,9\rangle$, jedinstvena sortirana lista dobijena njihovim objedinjavanjem je $\langle 1, 1, 2, 2, 3, 6, 7, 8, 9\rangle$. Obratite pažnju na to da se kod objedinjavanja dve liste uvek podrazumeva da su one već sortirane.

Jednostavan postupak za objedinjavanje dve sortirane liste jeste da se obe liste paralelno ispituju redom od početka do kraja. Pri tome se u svakom koraku upoređuju glave dve liste, bira se manji element od njih (ako su oba elementa jednaka, uzima se bilo koji) za naredni element objedinjene liste i, na kraju, uklanja se izabrani element iz njegove liste. Realizacija ovog postupka na pseudo jeziku je:

```
// Ulaz:  sortirane liste l_1 i l_2
// Izlaz: sortirana lista l formirana objedinjavanjem l_1 i l_2
```

```
algorithm list-merge(l1, l2, l)

   l = null;  // glava liste l
   r = null;  // rep liste l
   while ((l1 != null) && (l2 != null)) do
      if (l1.ključ <= l2.ključ) then
         x = l1;
         l1 = list-delete(null,l1);
      else
         x = l2;
         l2 = list-delete(null,l2);
      l = list-insert(x,r,l);
      r = x;
   while (l1 != null) do
      x = l1;
      l1 = list-delete(null,l1);
      l = list-insert(x,r,l);
      r = x;
   while (l2 != null) do
      x = l2;
      l2 = list-delete(null,l2);
      l = list-insert(x,r,l);
      r = x;

   return l1, l2;
```

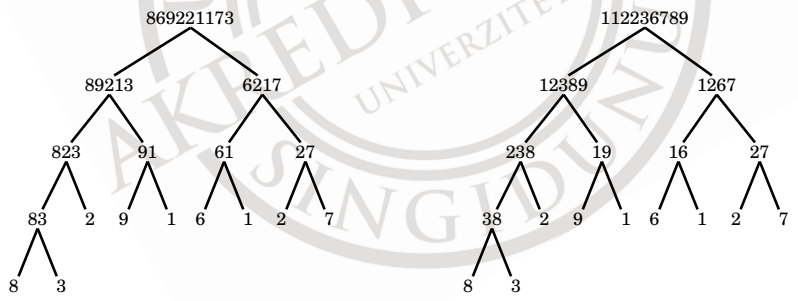
Upotrebom prethodna dva algoritma u prvom i trećem koraku postupka za sortiranje objedinjavanjem i dva rekurzivna poziva za sortiranje manjih lista brojeva u drugom koraku, kompletan algoritam za sortiranje objedinjavanjem na pseudo jeziku je:

```
// Ulaz:  neuređena lista l
// Izlaz: sortirana lista l
algorithm merge-sort(l)

   if (l.sled != null) then  // lista ima više od jednog elementa
      l1, l2 = list-split(l,l1,l2);
      l1 = merge-sort(l1);
      l2 = merge-sort(l2);
      l = list-merge(l1,l2,l);

   return l;
```

Pokažimo kako radi algoritam `merge-sort` na jednom primeru liste jednocifrenih brojeva $\langle 8,6,9,2,2,1,1,7,3\rangle$. Radi kraćeg zapisa, da bismo predstavili liste u ovom primeru, izostavljamo zagrade i zapete između cifara. Tako, ulazna lista je $l = 869221173$. Ova lista se najpre algoritmom

`list-split` deli u dve liste l_1 i l_2. Ove liste se sastoje od svakog drugog elementa polazne liste, odnosno prva lista sadrži neparne, a druga parne elemente polazne liste. Konkretno, $l_1 = 89213$ i $l_2 = 6217$. Zatim se ove polovične liste rekurzivno sortiraju istim algoritmom, što kao rezultat daje liste $l_1 = 12389$ i $l_2 = 1267$. Na kraju, algoritam `list-merge` proizvodi objedinjenu sortiranu listu $l = 112236789$.

Nema, naravno, ništa magičnog kod sortiranja dve manje liste, već se to dobija sistematičnom primenom rekurzivnog algoritma. Naime, algoritmom `merge-sort` se najpre deli ulazna lista na dve manje liste, a zatim se nastavlja ovo rekurzivno deljenje manjih lista sve dok se ne dobije to da svaka lista ima po jedan element. Na slici 5.4(a) je prikazano ovo rekurzivno deljenje listi iz primera. Zatim se dobijene manje liste objedinjuju u parovima, idući odozdo uz stablo, dok se cela lista ne dobije sortirana. Ovaj proces je prikazan na slici 5.4(b). Međutim, vredno je uočiti da je ovo deljenje i objedinjavanje manjih listi isprepletano, tj. sva objedinjavanja ne slede iza svih deljenja. Na primer, za manju listu $l_1 = 89213$ se izvršavaju sva deljenja i objedinjavanja pre nego što se bilo šta uradi sa drugom manjom listom $l_2 = 6217$.



(a) deljenje (b) objedinjavanje

SLIKA 5.4: Rekurzivno deljenje i objedinjavanje listi.

Analiza algoritma `merge-sort`. Neka je $T(n)$ vreme izvršavanja algoritma `merge-sort` za ulaznu listu l od n elemenata. Na osnovu dizajna tog algoritma možemo zaključiti da vreme njegovog izvršavanja zavisi od dva slučaja: kada ulazna lista ima jedan element ($n = 1$) i kada ima više od jednog elementa ($n > 1$).

Ako je $n = 1$, u tom algoritmu se izvršava konstantan broj instrukcija za proveravanje uslova naredbe `if` i izvršavanje naredbe `return`. Ukoliko

u ovom slučaju pojednostavimo analizu zanemarivanjem tačnog broja tih jediničnih instrukcija, pod pretpostavkom da se sve one zbirno izvršavaju za konstantno vreme c_1 dobijamo da je $T(1) = c_1$.

Ako je $n > 1$, u algoritmu `merge-sort` se izvršava deo `then` naredbe `if`, odnosno izvršavaju se algoritmi `list-split` i `list-merge` i dva rekurzivna poziva samog algoritma `merge-sort` za sortiranje dve manje liste od otprilike $n/2$ elemenata. Nije se teško uveriti, što ostavljamo čitaocima da provere, da je vreme izvršavanja oba algoritma `list-split` i `list-merge` proporcionalno dužini ulazne liste n. Prema tome, ako su faktori te proporcionalnosti konstante c_2 i c_3, onda su njihova vremena izvršavanja $c_2 n$ i $c_3 n$.

Na osnovu dosadašnje analize može se dakle zaključiti da je rekurentna jednačina za $T(n)$ data izrazom:

$$T(n) = \begin{cases} c_1, & \text{ako je } n = 1 \\ 2T(n/2) + c_2 n + c_3 n, & \text{ako je } n > 1. \end{cases}$$

U ovoj jednačini tačne vrednosti konstanti c_1, c_2 i c_3 nisu bitne, jer nas interesuje asimptotsko vreme izvršavanja u najgorem slučaju. Zbog toga možemo pretpostaviti da su one sve jednake, tj. $c_1 = c_2 = c_3 = c$ (uzimajući recimo najveću od njih), čime se dobija rekurentna jednačina za $T(n)$ koja je prostija za rešavanje:

$$T(n) = \begin{cases} c, & \text{ako je } n = 1 \\ 2T(n/2) + 2cn, & \text{ako je } n > 1. \end{cases} \tag{5.2}$$

Uzgred, obratite pažnju na još jedno suptilno pojednostavljenje u jednačini (5.2). Naime, dužina listi l_1 i l_2 u rekurzivnim pozivima algoritma `merge-sort` nije uvek jednaka $n/2$. Pažljivi čitaoci mogu lako proveriti da su te dužine zapravo $\lceil n/2 \rceil - 1$ i $\lfloor n/2 \rfloor$, pa bi u jednačini (5.2) za $n > 1$ na desnoj strani trebalo da stoji $T\big(\lceil n/2 \rceil - 1\big) + T\big(\lfloor n/2 \rfloor\big) + 2cn$. Kako je uvek $\lceil n/2 \rceil - 1 \leq \lfloor n/2 \rfloor$, ovu desnu stranu možemo pojednostaviti pišući $2T\big(\lfloor n/2 \rfloor\big) + 2cn$. (Ovde prirodno pretpostavljamo da je $T(n)$ rastuća funkcija i, radi jednostavnosti, koristimo jednakost umesto preciznije nejednakosti $\leq$.) U stvari, ovo ćemo dalje pojednostaviti i potpuno zanemariti rad sa celobrojnim delovima realnih vrednosti, kako u jednačini (5.2) tako i u preostalom delu knjige. To ima opravdanja jer se pažljivijom analizom može pokazati da to možemo uvek uraditi bez gubitka opštosti, pa radi jednostavnosti namerno biramo da budemo malo neprecizniji nauštrb formalne tačnosti.

Ako se sada vratimo rešavanju rekurentne jednačine (5.2), podsetimo se da metodom višestruke zamene treba opšti oblik za $T(n)$ zamenjivati na desnoj strani umesto vrednosti funkcije T za argumente manje od n, sve dok se ne dođe do početnog uslova. Tako, pod pretpostavkom $n/2 > 1$ i obavljajući jednu takvu zamenu u jednačini (5.2) na desnoj strani, dobijamo:

$$\begin{aligned} T(n) &= 2\left(2T\left(n/2^2\right) + 2c \cdot n/2\right) + 2cn \\ &= 2^2 T\left(n/2^2\right) + 4cn\,. \end{aligned}$$

U ovoj jednačini zamenjujući dalje opštu jednačinu za $T\left(n/2^2\right)$ na desnoj strani dobijamo novu jednačinu:

$$\begin{aligned} T(n) &= 2^2\left(2T\left(n/2^3\right) + 2c \cdot n/2^2\right) + 4cn \\ &= 2^3 T\left(n/2^3\right) + 6cn\,. \end{aligned}$$

Nastavljajući ovaj postupak zamene na desnoj strani prethodnih jednačina, nije teško prepoznati obrazac po kojem se dobija opšti oblik na desnoj strani izraza za $T(n)$:

$$T(n) = 2^i T\left(n/2^i\right) + 2icn, \quad i = 1, 2, 3, \ldots$$

Početni uslov $T(1) = c$ na desnoj strani ove jednačine dobija se kada je $n/2^i = 1$, odnosno $n = 2^i$, ili $i = \log n$. Zamenom tog uslova na desnoj strani najzad dobijamo rešenje jednačine (5.2) u zatvorenom obliku:

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

Ako se podsetimo asimptotske notacije iz odeljka 3.2, nije teško pokazati da se ovo rešenje može dalje izraziti O-zapisom u obliku $T(n) = O(n\log n)$. Drugim rečima, vreme izvršavanja algoritma `merge-sort` je linearno-logaritamsko, što je asimptotski mnogo brže od kvadratnog vremena izvršavanja jednostavnih algoritama za sortiranje o kojima smo govorili u odeljku 2.3 na strani 34.

Primer: sortiranje razdvajanjem (*quick-sort*)

U ovom odeljku ćemo pokazati još jedan rekurzivan način za rešavanje problema sortiranja. Taj metod za sortiranje neuređenog niza brojeva, koji se popularno naziva *sortiranje razdvajanjem* (engl. *quick-sort*), jedan je od najčešće korišćenih metoda u praksi. Mada u najgorem slučaju nije bolji od sporih, kvadratnih metoda sortiranja, u praktičnim primenama se pokazao

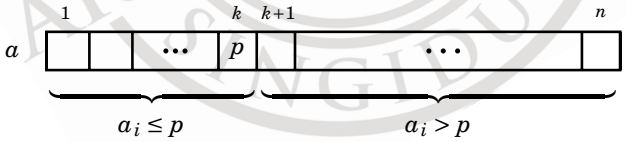
toliko efikasnim da je čak na engleskom dobio ime koje u prevodu znači „brzo sortiranje".

Algoritam za sortiranje niza razdvajanjem sastoji se od dva glavna koraka. U prvoj fazi se ulazni niz koji treba sortirati deli u dva podniza tako što se bira jedan element datog niza oko kojeg se dati niz razdvaja. Taj izabrani element se naziva *pivot*, a najbolji slučaj se dobija kada se ulazni niz preuredi tako da pivot podeli taj niz u dva približno jednaka dela.

Konkretno, neka je ulazni niz a od n elemenata koji treba sortirati. Ako je p vrednost izabranog pivot elementa, tada se pivot smešta na svoje pravo mesto u sortiranom nizu, svi elementi niza koji su manji ili jednaki pivotu premeštaju se tako da se nalaze levo od njega, a svi elementi koji su veći od pivota se premeštaju tako da se nalaze desno od njega. To formalno znači da se elementi niza permutuju tako da budu zadovoljena sledeća tri uslova:

- $a_k = p$ za neko k između 1 i n;
- $a_i \leq p$ za svako i tako da je $1 \leq i \leq k$; i
- $a_i > p$ za svako i tako da je $k+1 \leq i \leq n$.

Na slici 5.5 je prikazan rezulat ove permutacije elemenata u prvoj fazi algoritma.



SLIKA 5.5: Podela niza kod sortiranja razdvajanjem.

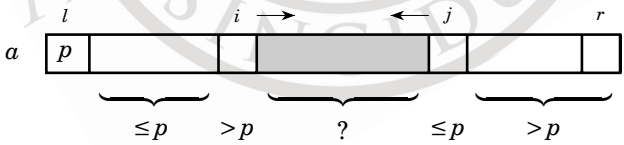
U drugoj fazi sortiranja razdvajanjem se rekurzivno sortiraju dve oblasti originalnog niza koje se nalaze sa obe strane pivota. Time se, u stvari, automatski dobija ceo niz potpuno sortiran, jer su svi elementi u prvoj oblasti levo od pivota manji od svih elemenata u drugoj oblasti desno od pivota.

Izbor dobrog pivota je ključni korak u algoritmu, jer ukoliko je veličina dve oblasti sa obe strane pivota podjednaka, vreme potrebno za njihovo sortiranje će biti otprilike isto. To onda izbalansirano sa vremenom za podelu niza dovodi do efikasnog sortiranja. Najbolji kandidat od elemenata niza da bude pivot, jeste, dakle, onaj element niza koji je veći od otprilike

polovine elemenata i manji od otprilike druge polovine elemenata niza. Taj element koji je u sredini sortiranog niza naziva se *medijana* niza. Međutim, nalaženje medijane niza nije jednostavno i zahteva više vremena nego što se to isplati. Zbog toga, pivot se obično bira na jednostavan način, u nadi da je izabrani element blizu medijane. Ono što je iznenađujuće je da se ovaj pristup pokazao dobrim u praksi, jer je sortiranje razdvajanjem vrlo efikasno u proseku.

Mogući načini za jednostavan izbor pivota koji se primenjuju u praksi obuhvataju izbor prvog, poslednjeg, srednjeg, ili čak slučajnog elementa (pod)niza. U nastavku ovog odeljka ćemo opisati postupke za razdvajanje podniza izborom njegovog prvog i poslednjeg elementa za pivota, dok se izbor srednjeg elementa niza za pivota ostavlja čitaocima za laku vežbu.

Razmotrimo najpre razdvajanje niza kada se bira njegov prvi element za pivota. Pretpostavimo da želimo da podelimo podniz $a_l,\ldots,a_r$, gde je $1 \le l \le r \le n$, i neka je pivot prvi element tog podniza $p = a_l$. Da bismo preuredili elemente oko pivota p, elemente podniza redom ispitujemo samo jednom, ali to radimo od oba kraja. Ako je i indeks ispitanih elemenata idući od levog kraja i j indeks ispitanih elemenata idući od desnog kraja, to znači da oni na početku imaju vrednosti $l+1$ i r, tim redom. Zatim se indeks i uvećava za 1 sve dok ne naiđemo na element takav da je $a_i > p$, a indeks j se smanjuje za 1 sve dok ne naiđemo na element takav da je $a_j \le p$. Na slici 5.6 je ilustrovan ovaj postupak.



SLIKA 5.6: Pivotiranje sa prvim elementom podniza.

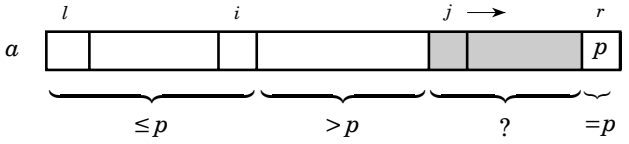
Prema tome, od levog kraja podniza se preskaču svi elementi koji su manji ili jednaki p dok se ne otkrije element a_i koji je veći od p. Zatim od desnog kraja podniza se preskaču svi elementi koji su veći od p dok se ne otkrije element a_j koji je manji ili jednak p. Dodatno, da bi se elementi a_i i a_j premestili na pravu stranu u odnosu na pivota, međusobno se zamenjuju njihove vrednosti. Ovaj isti postupak se ponavlja sve dok je $i < j$. Kada se indeksi i i j preklope (tj. $i \ge j$), to znači da je ispitan ceo podniz. Ono što preostaje je da se međusobno zamene elementi a_l i a_j da bi se pivot element premestio na njegovo pravo mesto.

Ovaj postupak na pseudo jeziku je u nastavku predstavljen algoritmom partition kojim se razdvaja podniz $a_l,\ldots,a_r$ oko njegovog prvog elementa izabranog za pivota i vraća indeks tog pivota u rezultujućem preuređenom podnizu.

```
// Ulaz:  podniz niza a ograničen indeksima od l do r
// Izlaz: preuređen podniz i njegov indeks u kome se nalazi pivot
algorithm partition(a, l, r)

   // Pretpostavlja se da je a_{r+1} = +∞
   p = a[l];
   i = l + 1; j = r;
   while (i < j) do
      while (a[i] <= p) do i = i + 1;
      while (a[j] > p)  do j = j - 1;
      swap(a[i],a[j]);
   swap(a[l],a[j]);

   return j;
```

U algoritmu partition imamo jedan tehnički problem koji smo razrešili tako što smo ga ignorisali. Naime, kada se po prvi put redom ispituje podniz, počinjući od levog kraja i tražeći element koji je veći od pivota p, postoji mogućnost da se pređe preko desnog kraja podniza. To se dešava ukoliko je pivot najveći element podniza ili su svi elementi podniza jednaki. Da bismo u algoritmu partition izbegli proveravanje ovih uslova i tako pokvarili njegovu jednostavnost, unapred smo pretpostavili da uvek postoji granični element podniza na desnom kraju tako da je $a_{r+1} = +\infty$. Aktuelni detalji rešenja ovog tehničkog problema se prepuštaju čitaocima za laku vežbu.

Razmotrimo sada razdvajanje niza kada se bira njegov poslednji element za pivota. U ovom slučaju se niz redom ispituje samo u jednom smeru (sleva na desno) i deli u četiri (moguće prazne) oblasti kao što je to prikazano na slici 5.7.



SLIKA 5.7: Pivotiranje sa poslednjim elementom podniza.

Svaka od četiri oblasti niza na slici 5.7 zadovoljava određen uslov koji

je naznačen ispod odgovarajuće oblasti. Imajući to na umu, nije se teško uveriti da naredna verzija algoritma `partition` ispravno razdvaja dati podniz oko njegovog poslednjeg elementa izabranog za pivota, jer ovi uslovi zapravo predstavljaju invarijantu `for` petlje u algoritmu.

```
// Ulaz:  podniz niza a ograničen indeksima od l do r
// Izlaz: preuređen podniz i njegov indeks u kome se nalazi pivot
algorithm partition(a, l, r)

   p = a[r];
   i = l - 1;
   for j = l to r-1 do
      if (a[j] <= p) then
         i = i + 1;
         swap(a[i],a[j]);
   swap(a[i+1],a[r]);

   return i+1;
```

Bilo koju od dve prethodne verzije algoritma `partition` možemo koristiti da bismo napisali algoritam `quick-sort(a,l,r)` koji razdvajanjem elemenata sortira podniz $a_l,\ldots,a_r$, gde je $1 \le l \le r \le n$. Eksplicitne granice podniza su potrebne kao parametri ovog algoritma zbog njegove rekurzivne prirode. Naravno, ukoliko želimo da sortiramo ceo niz a dužine n, to možemo u programu uraditi pozivom `quick-sort(a,1,n)`.

```
// Ulaz:  podniz niza a ograničen indeksima od l do r
// Izlaz: sortiran podniz
algorithm quick-sort(a, l, r)

   if (l < r) then
      k = partition(a,l,r);
      quick-sort(a,l,k-1);
      quick-sort(a,k+1,r);

   return a;
```

Analiza algoritma `quick-sort`. Izvršavanje algoritma `quick-sort` zavisi od toga da li su razdvojeni delovi podnizova uravnoteženi po veličini, što sa svoje strane zavisi od *ranga*[2] elementa izabranog za pivota. Ako su podnizovi na svakom nivou stabla rekurzije otprilike jednako podeljeni, sortiranje razdvajanjem se asimptotski izvršava kao i sortiranje objedinjavanjem za vreme $O(n \log n)$. Sa druge strane, ako je razdvajanje podnizova

[2]Rang nekog elementa jednak je njegovoj poziciji u sortiranom nizu.

vrlo neuravnoteženo, vreme za sortiranje razdvajanjem može degradirati do $O(n^2)$. Srećom, u proseku, „dobre" podele preovlađuju u odnosu na „loše" podele, pa je ovo „brzo" sortiranje zaista efikasno i tipično vreme izvršavanja iznosi $O(n \log n)$.

Pošto vreme izvršavanja sortiranja razdvajanjem nije uvek isto, razmotrimo najbolji i najgori mogući slučaj. Pre svega, primetimo da se obe verzije algoritma `partition` izvršavaju za vreme koje je proporcionalno dužini ulaznog podniza. Naime, u oba slučaja, svaki element ulaznog podniza se ispituje samo jednom. U prvom slučaju kada se za pivota bira prvi element podniza, elementi podniza se ispituju od oba kraja, ali se pri tome staje čim se odgovarajući indeksi preklope. U drugom slučaju kada se za pivota bira poslednji element podniza, elementi podniza se prosto ispituju redom od levog kraja do desnog kraja. Dakle, vreme izvršavanja algoritma `partition(a,l,r)` je $c(r-l+1)$. Specifično, početno razdvajanje ($l = 1$, $r = n$) celog ulaznog niza se izvršava za vreme cn.

Najgori slučaj za performanse sortiranja razdvajanjem jeste kada u svim slučajevima razdvajanja podniza izabrani pivot bude najveći element podniza. To se dešava u prvoj verziji algoritma `partition` kada je ulazni niz početno sortiran u opadajućem redosledu, a u drugoj verziji kada je ulazni niz već sortiran u rastućem redosledu. Tada u svakom rekurzivnom pozivu, podniz delimo na dva dela od kojih se jedan sastoji od jednog (pivot) elementa, a drugi od svih ostalih elemenata podniza. Prema tome, rekurentna jednačina koja opisuje vreme izvršavanja $T(n)$ algoritma `quick-sort` u najgorem slučaju je:

$$\begin{aligned} T(n) &= T(n-1) + T(1) + cn \\ &= T(n-1) + cn\,, \end{aligned}$$

jer je vreme izvršavanja algoritma $T(1)$ za jedan element očigledno neka konstanta. Lako je dobiti, na primer, postupkom višestruke zamene da je rešenje ove rekurentne jednačine $O(n^2)$. Prema tome, vreme izvršavanja za sortiranje razdvajanjem u najgorem slučaju nije ništa bolje od vremena izvršavanja jednostavnijih postupaka sortiranja.

U najboljem slučaju razdvajanja podnizova, kada se na svakom rekurzivnom nivou algoritma ulazni podniz deli na otprilike dve jednake polovine, rekurentna jednačina za vreme izvršavanja postaje:

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

Ova jednačina je identična rekurentnoj jednačini za vreme izvršavanja kod sortiranja objedinjavanjem, pa vreme izvršavanja za sortiranje razdvajanjem u najboljem slučaju iznosi $O(n \log n)$.

Prosečan slučaj vremena izvršavanja sortiranja razdvajanjem je mnogo bliži najboljem slučaju nego najgorem slučaju. Pošto formalni dokaz ove činjenice zahteva složeniji matematički aparat, ovde ćemo samo intuitivno objasniti zašto je sortiranje razdvajanjem efikasan metod u praksi. Pretpostavimo da algoritam `partition` uvek deli ulazni podniz u odnosu, recimo, 90-10 procenata. To izgleda kao vrlo neuravnoteženo deljenje i za vreme izvršavanja algoritma `quick-sort` dobijamo rekurentnu jednačinu:

$$T(n) = T(9n/10) + T(n/10) + cn\,.$$

Ali rešenje ove rekurentne jednačine je i dalje $T(n) = O(n \log n)$, što se može proveriti indukcijom, mada sa većom konstantom skrivenom u asimptotskoj notaciji. Dakle, čak i kada se u postupku sortiranja razdvajanjem dobijaju naizgled vrlo nejednake veličine podnizova, asimptotsko vreme izvršavanja je i dalje $O(n \log n)$ — isto ono koje se dobija kada su podele tačno po sredini. U stvari, sve podele izvršene u konstantnoj proporciji na svakom nivou rekurzije, čak i za 99-1 procenata, daju vreme izvršavanja $O(n \log n)$.

Iako nije mnogo verovatno da ćemo za slučajni ulazni niz uvek dobiti proporcionalne podele, očekujemo da će neke podele biti dobro uravnotežene i da će neke biti prilično neuravnotežene. Drugim rečima, dobre i loše podele će se smenjivati na svim nivoima rekurzije. Međutim, očekujemo da dobijemo mnogo više dobrih podela nego loših, pošto naša neformalna analiza nagoveštava da su naizgled loše podele zapravo dobre. Dakle, čak i kada se neka vrlo loša podela dobije pre dobrih podela, odgovarajući podniz će se brzo sortirati. Naime, dodatni posao zbog loše podele se ne odražava na dodatne rekurzivne pozive, nego je dominiran dodatnim razdvajanjem koje se apsorbuje u ukupno vreme izvršavanja.

5.3 Rekurentne jednačine

U primerima u prethodnom odeljku uverili smo se da se u analizi rekurzivnih algoritama obično srećemo sa rekurentnim jednačinama koje na neki način treba rešiti. Drugim rečima, ukoliko funkcija $T(n)$ kod tih algoritama označava maksimalno vreme izvršavanja za ulaz veličine n, onda se $T(n)$ izražava jednačinom u kojoj učestvuju vrednosti te iste funkcije za

veličine ulaza manje od n. Zadatak je naći formulu (ili bar gornju granicu) za $T(n)$ izraženu samo pomoću veličine ulaza n.

Kroz primere u prethodnom odeljku smo upoznali i metod *višestruke zamene* za rešavanje nekih rekurentnih jednačina. Pokazali smo da se metod višestruke zamene sastoji od zamenjivanja vrednosti $T(m)$ za $m < n$ opštim oblikom rekurentnog dela jednačine za $T(n)$. Pri tome nakon nekoliko zamena očekujemo da možemo odrediti opšti obrazac za $T(n)$ i da možemo odrediti kada taj opšti oblik prelazi u bazni slučaj.

Sa matematičke tačke gledišta, postoji trenutak kada kod metoda višestruke zamene dolazi do malo opasnog logičkog skoka. To se događa u trenutku kada o opštem obliku rešenja zaključujemo na osnovu obrasca koji dobijamo nakon niza zamena. Često je taj logički skok sasvim opravdan, ali ponekad to nije baš očigledno i može biti podložno greškama. Da bismo bili potpuno sigurni u takvim slučajevima, opšti oblik rešenja treba dalje verifikovati, obično indukcijom. Kombinovan sa ovom verifikacijom, metod višestruke zamene je sasvim korektan način za dobijanje rešenja rekurentnih jednačina.

Pošto se u opštem slučaju analize rekurzivnih algoritama mogu dobiti rekurentne jednačine koje su teže za rešavanje, u ovom odeljku ćemo upoznati još neke matematičke metode za rešavanje rekurentnih jednačina rekurzivnih algoritama.

Metod stabla rekurzije

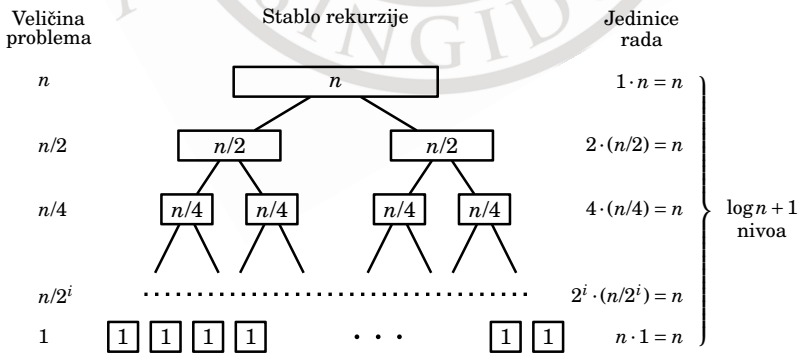
Metod višestruke zamene se sastoji od zamenjivanja rekurentnog dela jednačine za $T(n)$, a zatim od često složenog sređivanja dobijenog izraza u oblik iz koga možemo prepoznati opšti obrazac. Međutim, kod težih primera se lako možemo izgubiti u svom tom simboličkom manipulisanju, pa se zato upotrebom stabla rekurzije obezbeđuje slikovitiji način za praćenje šta se događa na svakom nivou rekurzije. Kod primene metoda *stabla rekurzije*, rekurentna jednačina se predstavlja stablom u kojem nas svaka zamena rekurentnog dela polazne jednačine dovodi do jednog nivoa dublje u stablu.

Glavnu ideju metoda stabla rekurzije za rešavanje rekurentnih jednačina objasnićemo na primeru rekurentne jednačine (5.2) za vreme izvršavanja algoritma `merge-sort` na strani 121 (koju smo dalje pojednostavili

uzimajući da je konstanta $c = 1$):

$$T(n) = \begin{cases} 1, & \text{ako je } n = 1 \\ 2T(n/2) + n, & \text{ako je } n > 1. \end{cases} \tag{5.3}$$

Podsetimo se algoritamske interpretacije ove jednačine: da bismo niz dužine n sortirali objedinjavanjem, moramo sortirati objedinjavanjem dva podniza dužine $n/2$ i uraditi n jedinica dodatnog posla za njihovo nerekurzivno objedinjavanje. Za rekurentnu jednačinu sortiranja objedinjavanjem crtamo stablo rekurzije sa korenom koji predstavlja polazni problem i koren označavamo vremenom za rešavanje tog problema u nerekurzivnom delu, koje iznosi n. Činjenicu da polazni problem delimo u dva potproblema predstavljamo crtanjem dvoje dece korenog čvora. Tu decu takođe označavamo vremenom za njihovo reševanje u nerekurzivnom delu, koje sada iznosi $n/2$. Na sledećem nivou imamo četiri čvora, pošto se svaki od dva potproblema na prethodnom nivou dalje deli u dva potproblema. Svaki od ova četiri čvora ima oznaku $n/4$, jer je to vreme potrebno za rešavanje svakog od ovih potproblema u nerekurzivnom delu. Ovaj postupak se nastavlja sve do poslednjeg nivoa na dnu stabla, koje sadrži čvorove za probleme veličine 1 i koji su označeni vremenom za njihovo rešavanje iz baznog slučaja. Na slici 5.8 je prikazano rezultujuće stablo rekurzije za rekurentnu jednačinu (5.3) koja opisuje funkciju vremena izvršavanja za sortiranje objedinjavanjem.



SLIKA 5.8: Stablo rekurzije za rekurentnu jednačinu (5.3).

Primetimo da smo u krajnje desnoj koloni na slici 5.8 naveli ukupno vreme za svaki nivo rekurzije. Ako nivo na samom vrhu označimo da bude nivo 0, onda na nivou i imamo 2^i potproblema veličine $n/2^i$ i dodatno vreme

za rešavanje svakog od njih u nerekurzivnom delu je $n/2^i$. To znači da je za i-ti nivo potrebno ukupno vreme jednako $2^i \cdot (n/2^i) = n$. Pošto stablo ima $\log n + 1$ nivoa $(0, 1, 2, \ldots, \log n)$, a svaki nivo doprinosi istu vrednost od n jedinica dodatnog vremena, ukupno vreme je $T(n) = n(\log n + 1) = n \log n + n$. Ovo rešenje rekurentne jednačine sortiranja objedinjavanjem je tačno ono koje smo dobili metodom višestruke zamene.

Važno zapažanje je da tačno znamo koliko ima nivoa u stablu rekurzije, kao i to koliko vremena nam je potrebno za svaki nivo. Pošto ovo znamo, ukupno vreme možemo dobiti ukoliko saberemo sva vremena po svim nivoima. To ukupno vreme je rešenje rekurentne jednačine (5.3), pošto stablo rekurzije samo slikovito modelira postupak odvijanja rekurzivnog izvršavanja algoritma. Prema tome, rešenje rekurentne jednačine (5.3) je $T(n) = n \log n + n$.

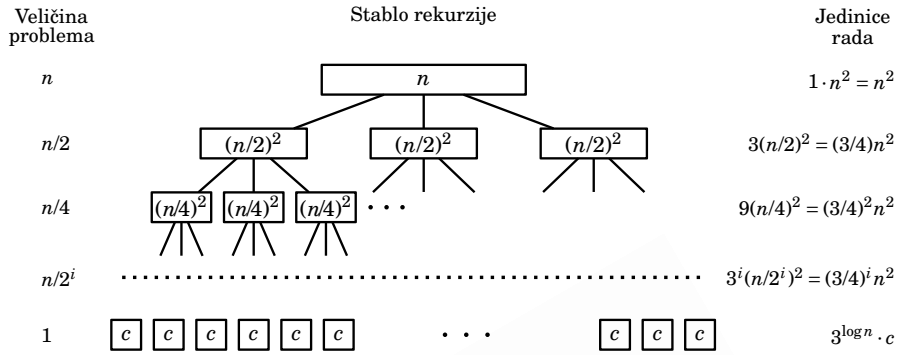
Primetimo da je rešavanje rekurentne jednačine metodom stabla rekurzije ekvivalentno primeni metoda višestruke zamene, iako vizuelno pruža bolju sliku o tome kako se dolazi do rešenja. Da bismo ovo bolje utvrdili, posmatrajmo još jednu rekurentnu jednačinu (gde je $c > 0$ neka konstanta):

$$T(n) = \begin{cases} c, & \text{ako je } n = 1 \\ 3T(n/2) + n^2, & \text{ako je } n > 1. \end{cases} \tag{5.4}$$

Ukoliko zanemarimo bazni slučaj, rekurentnu jednačinu (5.4) možemo interpretirati na sledeći način: da bismo rešili problem veličine n, rešavamo 3 potproblema veličine $n/2$ i radimo n^2 jedinica dodatnog posla. Kada crtamo stablo rekurzije, svaki čvor opet označavamo iznosom dodatnog (nerekurzivnog) vremena za odgovarajuću veličinu potproblema. Na najvišem nivou 0, za dodatni posao treba n^2 jedinica vremena. Na nivou 1 imamo tri čvora po $(n/2)^2$ jedinica dodatnog vremena, što za taj nivo daje ukupno vreme $3(n/2)^2 = (3/4)n^2$. Na nivou 2 imamo devet čvorova po $(n/4)^2$ jedinica dodatnog vremena, što za taj nivo daje ukupno vreme $9(n/4)^2 = (3/4)^2 n^2$. Ovo je lako nastaviti za nivo i, gde imamo 3^i čvorova po $\left(n/2^i\right)^2$ jedinica dodatnog vremena, što za taj nivo daje ukupno vreme $3^i \left(n/2^i\right)^2 = (3/4)^i n^2$.

Nivo na dnu je drugačiji od ostalih nivoa. U ostalim nivoima, vreme po čvoru dobijamo na osnovu rekurzivnog dela $T(n) = 3T(n/2) + n^2$ rekurentne jednačine. U najnižem nivou, vreme po čvoru dobijamo na osnovu baznog slučaja $T(1) = c$ rekurentne jednačine. Na slici 5.9 je prikazano stablo rekurzije koje se dobija za rekurentnu jednačinu (5.4).

Naravno, da bismo dobili ukupno vreme za celo stablo rekurzije, tj. rešenje jednačine (5.4), sabiramo vremena po svim nivoima stabla. Pošto je



SLIKA 5.9: Stablo rekurzije za rekurentnu jednačinu (5.4).

ukupan broj nivoa jednak $\log n + 1$ (to su nivoi $0, 1, 2, \ldots, \log n$) i broj čvorova na najnižem nivou je jednak $3^{\log n}$, zbir vremena po svim nivoima stabla je:

$$\begin{aligned} T(n) &= n^2 + (3/4)n^2 + (3/4)^2 n^2 + \cdots + (3/4)^{\log n - 1} n^2 + c3^{\log n} \\ &= n^2 \sum_{i=0}^{\log n - 1} (3/4)^i + c3^{\log n} \\ &= n^2 \cdot \frac{(3/4)^{\log n} - 1}{3/4 - 1} + c3^{\log n} \\ &= n^2 \cdot \frac{1 - n^{\log 3}/n^2}{1/4} + cn^{\log 3} \\ &= 4n^2 - 4n^{\log 3} + cn^{\log 3} \\ &= 4n^2 + (c-4)n^{\log 3}. \end{aligned}$$

U ovom izvođenju smo koristili jednakost $a^{\log_b n} = n^{\log_b a}$, kao i formulu za zbir geometrijskog reda:

$$\sum_{i=0}^{m} x^i = \frac{x^{m+1} - 1}{x - 1}, \quad (x \neq 1).$$

Prema tome, tačno rešenje rekurentne jednačine (5.4) je $T(n) = 4n^2 + (c-4)n^{\log 3}$. Ako nas interesuje samo asimptotska vrednost tog rešenja, kako je $\log_2 3 \approx 1.58$ to je $T(n) = O(n^2)$.

Metod nagađanja

Mada zvuči nesistematično, *metod nagađanja* je sasvim legalan način za rešavanje rekurentnih jednačina. Osnovna ideja tog pristupa sastoji se

u tome da se najpre promišljeno pretpostavi rešenje date rekurentne jednačine, a zatim da se ta pretpostavka potvrdi, obično indukcijom. Naravno, primenom ovog metoda možda moramo da probamo nekoliko rešenja dok ne dobijemo dobru gornju granicu za datu rekurentnu jednačinu. Drugim rečima, ako ne uspevamo da verifikujemo aktuelno pretpostavljeno rešenje, verovatno treba probati neku funkciju koja brže raste. Ali i ako uspemo u tome uz relativno malo truda, verovatno treba probati i neku funkciju koja sporije raste.

Na primer, posmatrajmo sledeću rekurentnu jednačinu (gde su a i b neke pozitivne konstante):

$$T(n) = \begin{cases} a, & \text{ako je } n = 1 \\ 2T(n/2) + bn\log n, & \text{ako je } n > 1. \end{cases} \tag{5.5}$$

Ovo izgleda vrlo slično rekurentnoj jednačini za sortiranje objedinjavanjem, pa naša prva pretpostavka može biti da je rešenje $T(n) = O(n\log n)$. Zatim, u sledećem koraku želimo da verifikujemo indukcijom da je rešenje rekurentne jednačine (5.5) zaista $T(n) \le cn\log n$ za neku pozitivnu konstantu c i dovoljno veliko n.

Sigurno možemo izabrati dovoljno velike konstante c i n_0 tako da naše pretpostavljeno rešenje zadovoljava bazni slučaj $n = n_0$. Naime, ako uzmemo $c = a + b$ i $n_0 = 2$, tada je:

$$T(2) = 2T(1) + b\cdot 2\cdot\log 2 = 2a + 2b = 2(a+b) = 2c = c\cdot 2\cdot\log 2.$$

Za indukcijski korak kada je $n > n_0$, ako po indukcijskoj pretpostavci uzmemo da je naše prvo rešenje tačno za vrednosti argumenta manjih od n, tada za argument jednak n možemo pisati:

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &\le 2\left(c\frac{n}{2}\log\frac{n}{2}\right) + bn\log n \\ &= cn(\log n - \log 2) + bn\log n \\ &= cn\log n - cn + bn\log n. \end{aligned}$$

Prema tome, da bismo dokazali indukcijski korak, moramo pokazati da važi nejednakost:

$$cn\log n - cn + bn\log n \le cn\log n,$$

odnosno $bn\log n \le cn$, ili ekvivalentno $\log n \le c/b$. Ali poslednja nejednakost nikako ne može biti tačna za svako $n \ge 2$. To znači da naše prvo naslućeno rešenje nije dobro.

Zato kao drugo rešenje za $T(n)$ moramo probati neku funkciju koja brže raste, na primer, $T(n) = O(n^2)$. U tom slučaju, da bismo pokazali $T(n) \leq cn^2$ za neku pozitivnu konstantu c i dovoljno veliko n, možemo uzeti $c = 2(a+b)$ i $n_0 = 1$ i lako videti da bazni slučaj važi. Za indukcijski korak kada je $n > n_0$, ako po indukcijskoj pretpostavci uzmemo da je naše drugo rešenje tačno za vrednosti argumenta manjih od n, tada za argument jednak n imamo:

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &\leq 2 \cdot c\frac{n^2}{4} + bn\log n \\ &\leq c\frac{n^2}{2} + \frac{c}{2} \cdot n^2 \\ &= cn^2 . \end{aligned}$$

Dakle u ovom slučaju indukcija prolazi. Međutim, pošto izgleda da je ovo rešenje lako dobiti, probaćemo još jednu (tj. „pravu") funkciju koja ne raste tako brzo: $T(n) = O(n\log^2 n)$. Sada, da bismo pokazali $T(n) \leq cn\log^2 n$ za neku pozitivnu konstantu c i dovoljno veliko n, možemo uzeti $c = a + b$ i $n_0 = 2$ i opet lako videti da bazni slučaj važi. Za indukcijski korak kada je $n > n_0$ imamo:

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &\leq 2\left(c\frac{n}{2}\log^2\frac{n}{2}\right) + bn\log n \\ &= cn(\log^2 n - 2\log n + 1) + bn\log n \\ &\leq cn\log^2 n - 2cn\log n + cn + cn\log n \\ &= cn\log^2 n - cn\log n + cn \\ &= cn\log^2 n - cn(\log n - 1) \\ &\leq cn\log^2 n . \end{aligned}$$

Ovo dokazuje da je $T(n)$ zaista $O(n\log^2 n)$.

Opšti metod

Prethodni načini za rešavanje rekurentnih jednačina zahtevaju određeno matematičko predznanje (recimo, primenu indukcije) da bi se mogli uspešno primeniti. Za neke čitaoce je možda lakši sledeći metod o kome govorimo, jer taj metod zahteva samo prepoznavanje tri karakteristična

slučaja i neposredno daje rešenje u svakom od njih. Ovaj *opšti metod* sadrži „formulu" na osnovu koje se može odrediti asimptotsko rešenje velike klase rekurentnih jednačina u obliku:

$$T(n) = aT(n/b) + f(n),$$

gde su $a \geq 1$ i $b > 1$ konstante, a $f(n)$ je asimptotski pozitivna funkcija.

Ovaj oblik rekurentne jednačine opisuje vreme izvršavanja rekurzivnog algoritma u kome se problem veličine n deli u a nezavisnih potproblema veličine najviše n/b svaki, zatim rešava svaki od ovih potproblema rekurzivno i, na kraju, kombinuju rešenja ovih potproblema u rešenje celog problema. Ukupno vreme za deljenje polaznog problema u potprobleme i za kombinovanje njihovih rešenja je izraženo funkcijom $f(n)$. Prema tome, ovo je zaista opšti oblik za rekurentne jednačine rekurzivnih algoritama.

Opšti metod zahteva prepoznavanje tri slučaja u jednoj teoremi, ali se zato rešenje mnogih rekurentnih jednačina dobija prilično lako i često se svodi samo na ispisivanje odgovora. Odgovarajući slučaj na osnovu koga se dobija rešenje prepoznaje se poređenjem funkcije $f(n)$ sa specijalnom funkcijom $n^{\log_b a}$. Pomenutu teoremu međutim nećemo navoditi u njenoj punoj opštosti, jer je prilično tehničke prirode i ima više teorijski značaj. U praksi se češće koristi njena pojednostavljena verzija u kojoj je $f(n)$ u suštini polinomska funkcija po n.

TEOREMA *Neka je nenegativna funkcija $T(n)$ definisana rekurentnom jednačinom*

$$T(n) = \begin{cases} d, & n \leq n_0 \\ aT(n/b) + cn^k, & n > n_0 \end{cases}$$

gde je $n_0 \geq 1$ celobrojna konstanta, a $a \geq 1$, $b > 1$, $c > 0$, $d > 0$ i $k \geq 0$ su realne konstante.

Slučaj 1: *Ako je $a = b^k$, tada je $T(n) = \Theta(n^k \log n)$.*

Slučaj 2: *Ako je $a < b^k$, tada je $T(n) = \Theta(n^k)$.*

Slučaj 3: *Ako je $a > b^k$, tada je $T(n) = \Theta(n^{\log_b a})$.*

Dokaz: Radi jednostavnosti pretpostavljamo da je n stepen broja b i $n_0 = 1$, kako ne bismo zamaglili osnovnu sliku nepotrebnim detaljima. Ipak naglašavamo da time ne gubimo na opštosti i ideja dokaza se prenosi za bilo koje n. Ako radi lakšeg pisanja uvedemo oznaku $f(n) = cn^k$, primenom višestruke zamene na datu rekurentnu jednačinu dobijamo:

$$T(n) = aT(n/b) + f(n)$$

$$
\begin{aligned}
&= a\left(aT\left(n/b^2\right) + f(n/b)\right) + f(n)\\
&= a^2T\left(n/b^2\right) + af(n/b) + f(n)\\
&= a^3T\left(n/b^3\right) + a^2f\left(n/b^2\right) + af(n/b) + f(n)\\
&\vdots\\
&= a^{\log_b n}T(1) + \sum_{i=0}^{\log_b n-1} a^i f\left(n/b^i\right)\\
&= n^{\log_b a}\cdot d + \sum_{i=0}^{\log_b n-1} a^i f\left(n/b^i\right),
\end{aligned}
$$

gde smo u poslednjoj jednakosti iskoristili formulu $a^{\log_b n} = n^{\log_b a}$.[3] Ako sada vratimo $f(n) = cn^k$, dobijamo:

$$
\begin{aligned}
T(n) &= dn^{\log_b a} + \sum_{i=0}^{\log_b n-1} a^i \cdot c\left(n/b^i\right)^k\\
&= dn^{\log_b a} + cn^k \cdot \sum_{i=0}^{\log_b n-1} \left(\frac{a}{b^k}\right)^i.
\end{aligned}
$$

Pošto sumiranje na desnoj strani predstavlja geometrijski zbir, tvrđenje teoreme ćemo pokazati posmatranjem slučajeva za vrednost količnika a/b^k.

Prvo, ako je $a = b^k$, tj. $k = \log_b a$, onda je

$$T(n) = dn^k + cn^k \log_b n = \Theta\left(n^k \log n\right).$$

Drugo, ako je $a \neq b^k$, onda je

$$T(n) = dn^{\log_b a} + cn^k \cdot \frac{\left(a/b^k\right)^{\log_b n} - 1}{a/b^k - 1}.$$

Dakle, ako je $a < b^k$, odnosno $k > \log_b a$ i $0 < a/b^k < 1$, onda je

$$T(n) = dn^{\log_b a} + cn^k \cdot \Theta(1) = \Theta\left(n^k\right),$$

pošto je drugi izraz dominantan i $\sum_{i=0}^{m} x^i = \Theta(1)$ za $0 < x < 1$.

U trećem slučaju, ako je $a > b^k$ odnosno $a/b^k > 1$, pošto za $x > 1$ imamo $\sum_{i=0}^{m} x^i = \Theta(x^m)$, dobijamo:

$$T(n) = dn^{\log_b a} + cn^k \cdot \Theta\left(\left(a/b^k\right)^{\log_b n}\right)$$

[3] To je razlog zašto se koristi specijalna funkcija $n^{\log_b a}$ za prepoznavanje rešenja u opštem slučaju.

$$= d n^{\log_b a} + c n^k \cdot \Theta\left(\frac{n^{\log_b a}}{n^k}\right)$$
$$= \Theta\left(n^{\log_b a}\right),$$

čime je dokaz završen. ■

Da bismo rešili neku rekurentnu jednačinu opštim metodom, jednostavno određujemo (ako možemo) slučaj prethodne teoreme koji je u pitanju i odmah dobijamo odgovor. Na primer, razmotrimo rekurentnu jednačinu:

$$T(n) = 4T(n/2) + n\,.$$

Pošto $a = 4$, $b = 2$ i $k = 1$ implicira $a > b^k$, na osnovu prvog slučaja prethodne teoreme odmah možemo zaključiti da je $T(n) = \Theta\left(n^{\log_2 4}\right) = \Theta\left(n^2\right)$.

Kao drugi primer, razmotrimo rekurentnu jednačinu:

$$T(n) = 2T\left(\sqrt{n}\right) + \log n = 2T\left(n^{1/2}\right) + \log n\,.$$

Nažalost, ova jednačina nije u onom obliku u kojem možemo primeniti opšti metod. Međutim, tu jednačinu možemo transformisati u željeni oblik smenom $m = \log n$. Tada je $n = 2^m$, pa možemo pisati:

$$T\left(2^m\right) = 2T\left(2^{m/2}\right) + m\,.$$

Ukoliko označimo $S(m) = T(2^m)$ dobijamo rekurentnu jednačinu za funkciju $S(m)$:

$$S(m) = 2S(m/2) + m\,.$$

Sada je oblik ove jednačine onaj koji omogućava primenu opšteg metoda, pa lako dobijamo rešenje $S(m) = \Theta(m \log m)$. Ponovnom upotrebom smene da bismo se vratili na $T(n)$ dobijamo rešenje $T(n) = \Theta(\log n \log\log n)$.

Primer: množenje velikih celih brojeva

Praktičniju primenu opšteg metoda za rešavanje rekurentnih jednačina ćemo pokazati na problemu množenja dva velika n-bitna cela broja. Pretpostavimo da je broj bitova n vrlo veliki (recimo, $n > 100$), tako da se celi brojevi ne mogu direktno pomnožiti u aritmetičkoj jedinici standardnih računara. Drugim rečima, ne možemo pretpostaviti da uobičajene aritmetičke operacije nad velikim celim brojevima predstavljaju jedinične operacije koje se izvršavaju za konstantno vreme. Očigledno, to vreme zavisi od broja bitova u reprezentaciji velikih celih brojeva u binarnom sistemu.

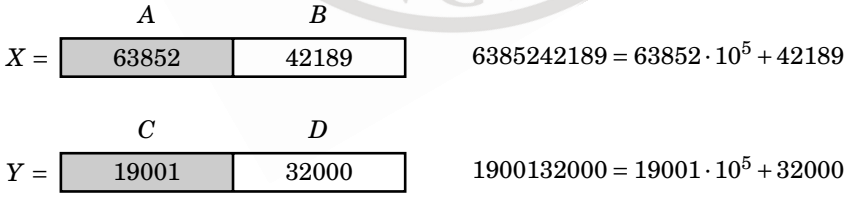
Množenje velikih celih brojeva ima primenu, na primer, u kriptografiji, jer se veliki celi brojevi koriste u raznim postupcima šifrovanja.

Ako su data dva velika cela broja X i Y predstavljena u binarnom sistemu sa po n bitova, njihov zbir $X+Y$ i razliku $X-Y$ možemo lako izračunati koristeći uobičajene postupke za ručno sabiranje i oduzimanje. Jasno je da vreme izvršavanja ovih postupaka iznosi $O(n)$ ukoliko sabiranja i oduzimanja pojedinačnih cifara računamo kao jedinične operacije. Međutim, ako se podsetimo da se uobičajeni postupak za množenje brojeva X i Y sastoji od izračunavanja n delimičnih proizvoda dužine n, nije se teško uveriti da je njegovo vreme izvršavanja $O(n^2)$ ukoliko množenja i sabiranja pojedinačnih cifara računamo kao jedinične operacije. Ali, kao što ćemo pokazati, jednim rekurzivnim algoritmom se može dobiti brži način za množenje dva n-bitna broja.

Radi jednostavnosti, pretpostavimo da je n stepen broja 2. Ako to nije slučaj, brojevima X i Y možemo dodati nevažeće nule sa leve strane. Da bismo primenili rekurzivni metod za množenje brojeva X i Y, n-bitne reprezentacije oba ta broja delimo u dve polovine po $n/2$ bitova. Jedna polovina predstavlja cifre veće važnosti, a druga predstavlja cifre manje važnosti. Ako polovine veće važnosti brojeva X i Y označimo sa A i C, tim redom, i slično polovine manje važnosti sa B i D, možemo pisati:

$$X = A2^{n/2} + B \quad \text{i} \quad Y = C2^{n/2} + D\,.$$

Ovo polovljenje brojeva je ilustrovano na slici 5.10 za $X = 6385242189$ i $Y = 1900132000$. Radi bolje čitljivosti, na toj slici su ovi desetocifreni brojevi prikazani u decimalnom brojnom sistemu, a ne u binarnom.



SLIKA 5.10: Polovljenje velikih brojeva.

Proizvod brojeva X i Y se dakle može napisati u obliku

$$X \cdot Y = \left(A2^{n/2} + B\right) \cdot \left(C2^{n/2} + D\right),$$

ili

$$X \cdot Y = AC \cdot 2^n + (AD + BC) \cdot 2^{n/2} + BD\,. \tag{5.6}$$

Ova formula nagoveštava jednostavan rekurzivni algoritam za izračunavanje proizvoda $A \cdot B$: podeliti n-bitne brojeve X i Y na pola, zatim rekurzivno izračunati proizvode u kojima učestvuju te polovine u (5.6) (naime AC, AD, BC i BD) i, na kraju, dobijene međuproizvode pomnožiti i sabrati prema formuli (5.6). Rekurzivni postupak se završava kada se dođe do množenja dva broja od po jednog bita, što je trivijalno.

Primetimo da je množenje binarnog broja stepenom broja 2, recimo 2^m, vrlo jednostavno — to se može postići prostim pomeranjem broja za m pozicija ulevo (ili dodavanjem m nula na kraju broja). To znači da se množenje n-bitnog celog broja nekim stepenom 2^m može obaviti za vreme $O(n+m)$ ukoliko pomeranje pojedinačnih cifara smatramo jediničnim operacijama.

Prema tome, ako proizvod $X \cdot Y$ rekurzivno računamo prema formuli (5.6), ukupno moramo obaviti četiri množenja $n/2$-bitnih celih brojeva (proizvodi AC, AD, BC i BD u formuli), tri sabiranja celih brojeva od najviše $2n$ bitova (što odgovara znakovima + u formuli) i dva pomeranja brojeva od otprilike n bitova (množenje sa 2^n i $2^{n/2}$ u formuli). Pošto vreme za svako od ovih sabiranja i pomeranja brojeva iznosi $O(n)$, vreme izvršavanja rekurzivnog algoritma za množenje $T(n)$ možemo opisati rekurentnom jednačinom:

$$T(n) = \begin{cases} c, & n = 1 \\ 4T(n/2) + cn, & n > 1, \end{cases}$$

za neku konstantu c.

Da bismo rešili ovu rekurentnu jednačinu, možemo primeniti pojednostavljenu opštu teoremu za $a = 4$, $b = 2$ i $k = 1$. Pošto nam to daje prvi slučaj teoreme, odmah dobijamo da je $T(n) = \Theta\left(n^{\log_2 4}\right) = \Theta\left(n^2\right)$. Nažalost, ovo nije ništa bolje od uobičajenog algoritma za množenje.

Na osnovu opšteg metoda ipak možemo steći uvid u to kako da ubrzamo rekurzivni algoritam. Ako bismo nekako uspeli da smanjimo broj rekurzivnih poziva a, tada bismo dobili sporiju specijalnu funkciju $n^{\log_b a}$ koja učestvuje u rešenju rekurentne jednačine, odnosno vremenu izvršavanja.

Možda zvuči iznenađujuće ako kažemo da to zaista možemo postići, ali posmatrajmo sledeći proizvod dva cela broja od $n/2$ cifara:

$$(A - B) \cdot (D - C) = AD - BD - AC + BC. \tag{5.7}$$

Mora se priznati da zaista deluje čudno zašto se posmatra baš ovaj proizvod, ali upoređujući formulu (5.6) vidimo da proizvod (5.7) u razvijenoj formi sadrži zbir dva međuproizvoda koji treba da izračunamo ($AD + BC$), kao i dva međuproizvoda (BD i AC) koje ionako računamo rekurzivno.

Prema tome, proizvod dva n-bitna cela broja X i Y možemo napisati u obliku:

$$X \cdot Y = AC \cdot 2^n + \Big((A-B)\cdot(D-C) + BD + AC\Big)\cdot 2^{n/2} + BD\,. \qquad (5.8)$$

Iz ove formule na desnoj strani vidimo da je za izračunavanje $X \cdot Y$ potrebno samo tri množenja $n/2$-bitnih celih brojeva (AC, $(A-B)\cdot(D-C)$ i BD), šest sabiranja ili oduzimanja i dva pomeranja brojeva. Pošto vreme za sve ove operacije osim množenja iznosi $O(n)$, vreme izvršavanja $T(n)$ rekurzivnog algoritma za množenje dva n-bitna cela broja prema formuli (5.8) dato je rekurentnom jednačinom:

$$T(n) = \begin{cases} c, & n = 1 \\ 3T(n/2) + cn, & n > 1. \end{cases}$$

Rešenje ove jednačine na osnovu pojednostavljene opšte teoreme je

$$T(n) = O\big(n^{\log_2 3}\big) = O\big(n^{1.585}\big),$$

što je bolje od $O\big(n^2\big)$.

Pisanje aktuelnog rekurzivnog algoritma na pseudo jeziku za množenje dva n-bitna cela broja po formuli (5.8) ostavljamo čitaocima za laku vežbu.

Zadaci

1. Jedna igra sa žetonima se sastoji od ukupno n žetona podeljenih u nekoliko grupa. Pretpostavlja se da je broj žetona n neki stepen broja 2 i da je broj grupa u koje su žetoni podeljeni proizvoljan. Jedan potez ove igre se sastoji od toga da se najpre biraju dve grupe žetona A i B i zatim ako grupa A ima a žetona i grupa B ima b žetona i ako je, recimo, $a \geq b$, premešta se b žetona iz grupe A u grupu B. Cilj igre je da se određenim nizom ovakvih poteza svi žetoni premeste u samo jednu grupu na kraju. Navedite algoritam kojim se rešava problem ove igre i odredite njegovo vreme izvršavanja (tj. broj poteza) izraženo u funkciji broja žetona n.

2. Napišite algoritam `partition` za sortiranje razdvajanjem kojim se za pivota bira srednji element podnizova. Obrazložite činjenicu da ovakav izbor pivot elementa ne utiče na smanjenje vremena izvršavanja algoritma za sortiranje razdvajanjem.

3. Napišite rekurzivni algoritam kojim se određuje najveći i drugi najveći element niza a od n brojeva. Koliko se poređenja parova elemenata niza a obavlja u tom algoritmu?

4. *Problem poznate ličnosti.* Pretpostavimo da se grupa od $n \geq 1$ osoba nalazi u jednoj sobi. *Poznata ličnost* u toj grupi je ona osoba koja ne poznaje nijednu drugu osobu u grupi (osim naravno sebe), a sve druge osobe poznaju tu osobu. Ako su poznanstva osoba predstavljena na odgovarajući način, problem poznate ličnosti se sastoji u nalaženju poznate ličnosti ako takva osoba postoji, ili se otkriva da takva osoba ne postoji. Ako sa $x_1, x_2, \ldots, x_n$ označimo n osoba i njihova poznanstva predstavimo $n \times n$ matricom a tako da je

$$a_{i,j} = \begin{cases} 1, & \text{ako } x_i \text{ poznaje } x_j \\ 0, & \text{ako } x_i \text{ ne poznaje } x_j \end{cases}$$

za $i, j = 1, 2, \ldots, n$, napišite rekurzivni algoritam koji kao rezultat daje indeks poznate ličnosti među ovih n osoba. Ako ne postoji poznata ličnost, rezultat algoritma treba da bude 0. Koliko je vreme izvršavanja tog algoritma? (*Savet:* Primetite da ako x poznaje y, tada x nije poznata ličnost, a u suprotnom slučaju, y nije poznata ličnost.)

5. Najpre pokažite da Fibonačijevi brojevi $f_1, f_2, f_3, \ldots$ zadovoljavaju sledeće jednačine za svako $k \geq 1$:

$$f_{2k-1} = (f_k)^2 + (f_{k-1})^2$$
$$f_{2k} = (f_k)^2 + 2 f_k f_{k-1},$$

a zatim napišite rekurzivni algoritam kojim se izračunava n-ti Fibonačijev broj primenom prethodnih jednačina. Koliko je vreme izvršavanja tog algoritma?

6. Koristeći matricu

$$M = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

i njen n-ti stepen M^n, napišite još jedan rekurzivni algoritam kojim se izračunava n-ti Fibonačijev broj. Koliko je vreme izvršavanja tog algoritma?

7. U neuređenom nizu a od n *različitih* brojeva, par različitih elemenata a_i i a_j čini jednu *inverziju* ako je $i < j$ i $a_i > a_j$. Na primer, u nizu $[5,3,10,2,8,9,20]$ nalazimo 6 inverzija: $(5,3)$, $(5,2)$, $(3,2)$, $(10,2)$, $(10,8)$ i $(10,9)$.

a) Napišite rekurzivni algoritam kojim se izračunava ukupan broj inverzija u datom nizu a dužine n.

b) Navedite i rešite rekurentnu jednačinu za vreme izvršavanja tog algoritma.

8. Napišite i analizirajte rekurzivni algoritam za problem maksimalne sume podniza koji je opisan na strani 29 u poglavlju 2. (*Savet:* Podelite ulazni niz u dve polovine i primetite da se podniz sa maksimalnom sumom nalazi ili potpuno u prvoj polovini, ili potpuno u drugoj polovini, ili mu je početak u prvoj i završetak u drugoj polovini. U poslednjem slučaju, nađite ga od srednjeg elementa idući u oba smera.)

9. Ako je dat skup od n timova u nekom sportu, *kružni turnir* je kolekcija mečeva u kojima se svaki tim sastaje sa svakim drugim timom tačno jedanput. Napišite i analizirajte rekurzivni algoritam kojim se konstruišu mečevi parova timova kružnog turnira pretpostavljajući da je n stepen broja 2.

10. Pretpostavljajući da je n stepen broja 3, rešite rekurentnu jednačinu

$$T(n) = \begin{cases} 1, & n < 3 \\ 4T(n/3) + n^2, & n \geq 3 \end{cases}$$

koristeći metod

a) višestruke zamene;

b) stabla rekurzije;

c) opšte teoreme.

11. Posmatrajmo sledeći algoritam koji izračunava proizvod dva nenegativna cela broja x i y:

```
// Ulaz:  celi brojevi x≥0 i y≥0
// Izlaz: x·y
```

```
algorithm multiply(x, y)

   p = 0;                    // p = x·y
   while (x > 0) do
      if (x % 2 != 0) then   // x je neparan
         p = p + y;
      x = x / 2;
      y = 2 * y;

   return p;
```

a) Pokažite da ovaj algoritam ispravno izračunava proizvod brojeva x i y.

b) Koliko je vreme izvršavanja ovog algoritma ako su ulazni brojevi mali, a koliko je ako su ti brojevi veliki?

c) Napišite rekurzivni algoritam koji koristi istu ideju za množenje brojeva x i y.

d) Koliko je vreme izvršavanja tog rekurzivnog algoritma ako su ulazni brojevi mali, a koliko je ako su ti brojevi veliki?

12. Napišite algoritam za deljenje velikih celih brojeva. Preciznije, ako su dati veliki pozitivni celi brojevi x i y, algoritam treba da izračuna količnik q i ostatak r tako da je $x = qy + r$ uz uslov $0 \leq r < b$. Koliko je vreme izvršavanja tog algoritma?

13. Napišite rekurzivni algoritam `multiply(x,y,n)` kojim se proizvod brojeva X i Y od n decimalnih cifara izračunava primenom formule (5.6).

14. Napišite rekurzivni algoritam `multiply(x,y,n)` kojim se proizvod brojeva X i Y od n decimalnih cifara izračunava primenom formule (5.8).

15. Ako su data dva linearna polinoma $ax + b$ i $cx + d$, njihov se proizvod

$$(ax + b)(cx + d) = acx^2 + (ad + bc)x + bd$$

može izračunati pomoću četiri množenja njihovih koeficijenata: ac, ad, bc i bd.

a) Koristeći ovo zapažanje, napišite rekurzivni algoritam kojim se izračunava proizvod dva polinoma stepena n:

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

i

$$B_n(x) = b_n x^n + b_{n-1} x^{n-1} + \cdots + b_1 x + b_0 .$$

Pretpostavljajući da su polinomi $A_n(x)$ i $B_n(x)$ dati nizovima svojih koeficijenata, rezultat algoritma treba da bude poseban niz koji redom sadrži koeficijente polinoma $A_n(x) \cdot B_n(x)$.

b) Navedite i rešite rekurentnu jednačinu za vreme izvršavanja algoritma pod a).

c) Pokažite kako se proizvod dva linearna polinoma $ax + b$ i $cx + d$ može izračunati sa samo *tri* množenja njihovih koeficijenata.

d) Na osnovu rešenja pod c), navedite drugi rekurzivni algoritam kojim se izračunava proizvod dva polinoma stepena n.

e) Navedite i rešite rekurentnu jednačinu za vreme izvršavanja algoritma pod d).

GLAVA

6

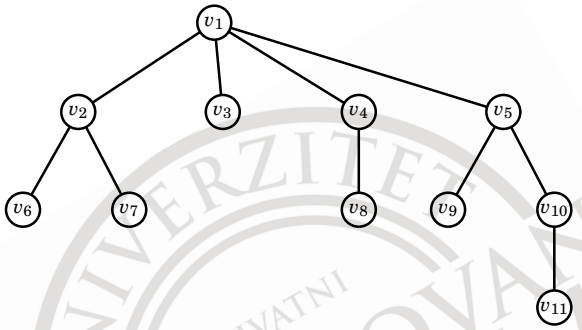
Stabla

Stablo je nelinearna struktura podataka koja predstavlja hijerarhijski odnos između elemenata. Dobro poznati primeri strukture stabla iz svakodnevnog života su izgled sadržaja neke knjige na njenom početku, rodoslovi familija i razne organizacione šeme. Stabla se u raznim oblicima vrlo često koriste i u računarstvu. Na primer, skoro svi operativni sistemi organizuju sistem datoteka i foldera na način koji odgovara strukturi stabla. Druga primena strukture stabla je za predstavljanje izraza u programima prilikom njihovog prevođenja od strane prevodilaca. Ove sintaksne kategorije se kod prevodilaca interno predstavljaju u obliku stabla izraza. Pored ovih neposrednih primena stabala, važna je i njihova indirektna primena za predstavljanje drugih struktura podataka kao što su disjunktni skupovi, redovi sa prioritetima i tako dalje.

6.1 Korenska stabla

Najopštije govoreći, stabla su samo specijalni slučaj grafova o kojima se detaljnije govori u poglavlju 7. Međutim, u ovom poglavlju nećemo punu pažnju posvetiti ovim opštim stablima, nego njihovoj ograničenoj verziji koja se nazivaju binarna stabla. Ipak radi potpunosti, pojmove u vezi sa stablima navodimo u opštijem slučaju korenskih stabala (ili stabala sa korenom). *Korensko stablo* se sastoji od skupa elemenata koji se nazivaju *čvorovi*, kao i skupa uređenih parova različitih čvorova povezanih linijama koje se nazivaju *grane*. Jedan od čvorova korenskog stabla je poseban

po tome što označava *koren* stabla. Na crtežima se čvorovi stabla obično označavaju kružićem, a koren stabla se obično prikazuje na vrhu hijerarhijske strukture koju čini stablo. Na slici 6.1 je prikazan primer korenskog stabla sa 11 čvorova, gde je čvor v_1 koren stabla. Pošto u nastavku ovog poglavlja govorimo isključivo o korenskim stablima, radi jednostavnosti ćemo ih kraće zvati prosto stablima.



SLIKA 6.1: Korensko stablo.

Stablo se neformalno najlakše predstavljaju crtežom njihovih čvorova i grana koje ih povezuju, ali je potrebno razjasniti još neke detalje u vezi sa stablima. Pre svega, jedna grana stabla je uređen par dva različita čvora x i y. U tekstu se ta grana označava sa (x, y), a na crtežima linijom koja spaja čvorove x i y. Primetimo da bi pravilnije bilo da na crtežima koristimo strelicu od x do y, jer je važno koji čvor je prvi a koji drugi u uređenom paru koji predstavlja granu. Ipak se radi lakšeg crtanja koristi linija za granu, a ne strelica, jer se redosled čvorova grane ukazuje time što se prvi čvor na crtežima uvek prikazuje iznad drugog čvora. Tako se na slici 6.1 podrazumeva da je, recimo, (v_2, v_7) grana stabla, a ne obrnuto.

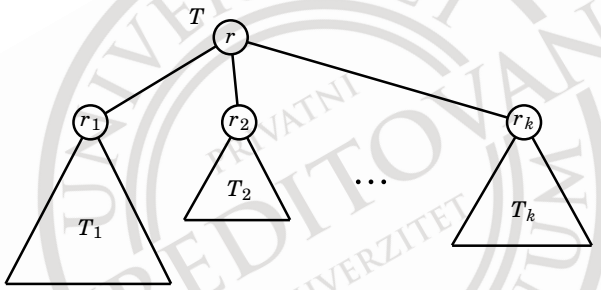
Terminologija u vezi sa stablima je uglavnom pozajmljena iz njihove primene u predstavljanju rodoslova neke porodice, gde se granama uspostavlja međusobni odnos roditelj-dete između pojedinih članova porodice. Tako se u računarstvu za granu (x, y) u stablu kaže da je prvi čvor *x roditelj* čvora y, odnosno da je drugi čvor *y dete* čvora x. Na primer, u stablu na slici 6.1 je čvor v_2 roditelj čvora v_7 i v_7 je dete čvora v_2.

Čvor u stablu može imati nula ili više dece, ali svaki čvor različit od korena ima tačno jednog roditelja i koren stabla je jedini čvor bez roditelja. Ova osobina je upravo ona koja karakteriše stablo i koristi se za formalnu definiciju (korenskog) stabla: stablo T je skup čvorova i skup grana između

njih (tj. uređenih parova čvorova) sa sledećim svojstvima:

1. Ako je T neprazno stablo, ono ima jedinstven poseban čvor (koren) koji nema roditeljski čvor.
2. Svaki čvor u u T osim korena ima jedinstven roditeljski čvor v.

Primetimo da na osnovu ove formalne definicije stablo može biti prazno, što znači da takvo stablo nema nijedan čvor (i nijednu granu). To omogućava da se stablo definiše i rekurzivno na način kojim se veća stabla konstruišu od manjih: stablo T je prazno ili se sastoji od čvora r, koji se naziva koren stabla T, i (moguće praznog) skupa stabala $T_1, T_2, \ldots, T_k$ čiji su koreni deca čvora r. Na slici 6.2 je ilustrovana ova rekurzivna definicija stabla.



SLIKA 6.2: Rekurzivna definicija stabla T.

Ostali pojmovi u vezi sa stablima su prilično prirodni, mada mnogobrojni. Deca istog čvora (tj. čvorovi sa zajedničkim roditeljem) nazivaju se *braća (sestre)*. Čvorovi koji imaju bar jedno dete su *unutrašnji*, a čvorovi bez dece su *spoljašnji*. Spoljašnji čvorovi se često nazivaju i *listovi*. Na primer, u stablu na slici 6.1 čvorovi v_2, v_3, v_4 i v_5 su braća, a svi listovi tog stabla su v_3, v_6, v_7, v_8, v_9 i v_{11}.

Pojam puta u stablu je na neki način uopštenje pojma grane. *Put* između dva različita čvora x i y je niz čvorova $x = v_1, v_2, \ldots, v_m = y$ ukoliko postoji grana između svih susednih čvorova (v_i, v_{i+1}) za $1 \leq i < m$. *Dužina* tog puta je broj grana na putu, odnosno $m - 1$. Za svaki čvor se podrazumeva da postoji put dužine 0 od tog čvora do samog sebe. Na primer, put u stablu na slici 6.1 od čvora v_5 do v_{11} sastoji se od čvorova v_5, v_{10} i v_{11}, a dužina tog puta je dva.

Bitna osobina stabla je da za svaki čvor u stablu postoji jedinstven put od korena do tog čvora. Da bismo ovo pokazali, lako je najpre videti da uvek

postoji put od korena do nekog čvora x. Naime, ukoliko uzimamo čvorove od čvora x prateći u svakom koraku jedinstvenog roditelja pojedinih čvorova, dobijamo niz čvorova koji u obrnutom redosledu formira put od korena do čvora x. Jedinstvenost tog puta se pokazuje kontradikcijom: ako bi postojao još jedan različit put, onda bi ta dva puta od korena morali da se kod nekog čvora najpre razdvoje i zatim ponovo spoje u bar jednom zajedničkom čvoru y (inače bi divergirali i jedan od puteva nikad ne bi stigao do x). Ali to bi značilo da čvor y ima dva roditelja, što je nemoguće kod stabla.

Svaki čvor u na jedinstvenom putu od korena do nekog čvora v se naziva *predak* čvora v. Obrnuto, čvor v je *potomak* čvora u ukoliko je u predak čvora v. Podrazumeva se da je svaki čvor u stablu sam sebi predak i potomak. Odnos predak-potomak za čvorove stabla se kraće može definisati rekurzivno: čvor u je predak čvora v ako je $u = v$ ili je u predak roditelja čvora v. Na primer, u stablu na slici 6.1 čvorovi v_1, v_4 i v_8 su preci čvora v_8, a potomci čvora v_5 su čvorovi v_5, v_9, v_{10} i v_{11}.

Podstablo sa čvorom v kao korenom je stablo koje se sastoji od tog čvora v i svih njegovih potomaka, zajedno sa nasleđenim granama. Na primer, sva stabla $T_1, T_2, \ldots, T_k$ na slici 6.2 su podstabla stabla T sa korenima, redom, $r_1, r_2, \ldots, r_k$.

Dubina čvora je dužina jedinstvenog puta od korena do tog čvora. Zato je, rekurzivno, dubina korena jednaka 0, a dubina svakog drugog čvora je za jedan veća od dubine njegovog roditelja. *Visina čvora* se takođe može definisati rekurzivno: visina lista je 0, a visina čvora koji nije list je za jedan veća od maksimalne visine njegove dece. *Visina stabla* je visina njegovog korena. Na primer, visina stabla na slici 6.1 je tri, a visina čvora v_5 je dva i njegova dubina je jedan.

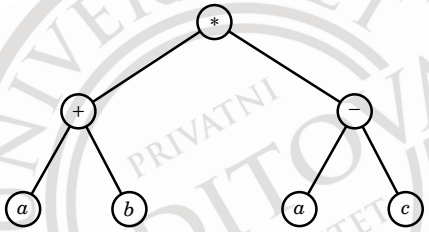
Stablo kao struktura podataka

Slično kao u slučaju osnovnih struktura podataka (nizova, listi, stekova, redova), da bismo stabla koristili u programima za organizaciju kolekcije podataka, moramo imati način za predstavljanje stabala u računaru i implementirati skup korisnih operacija nad stablima.

U radu sa stablima se može pojaviti potreba za vrlo raznolikim operacijama. Na primer, pored osnovnih operacija za dodavanje, uklanjanje ili određivanje deteta datog čvora, možemo imati i operacije koje kao rezultat daju roditelja, sasvim levog deteta ili desnog brata datog čvora. Radi lakšeg programiranja korisno bi bilo realizovati i upitne operacije koje određuju da li je dato stablo prazno, ili da li je dati čvor koren, interni, eksterni i

slično. Najzad, treba imati na umu i pomoćne operacije koje obuhvataju postupke konstruisanja novog stabla, određivanja broja čvorova (veličine) stabla, određivanje visine stabla i tako dalje. Međutim, ukoliko se koristi odgovarajuća reprezentacija stabla, sve ove operacije se mogu relativno lako implementirati i zato nećemo ići u konkretne detalje njihove praktične realizacije. Zato ćemo u ovom odeljku malo detaljnije razmotriti samo načine za predstavljanje korenskih stabala.

Pre svega, stablo predstavlja kolekciju podataka tako što se pojedinačni podaci u kolekciji nalaze u čvorovima stabla. Na primer, stablo na slici 6.3 predstavlja aritmetički izraz $(a+b)*(a-c)$ na prirodan način: koren tog stabla sadrži operator $*$ za množenje, a njegova deca su koreni dva podstabla koja predstavljaju podizraze $a+b$ i $a-c$.

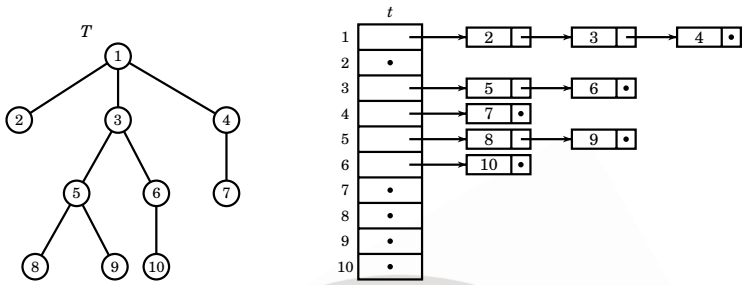


SLIKA 6.3: Stablo izraza $(a+b)*(a-c)$.

Slično kao kod osnovnih struktura podataka, pojedinačni podaci u čvorovima stabla se na opštem nivou poistovećuju sa jednoznačnom vrednošću koja se naziva njihov *ključ* i tako radi jednostavnosti zanemaruju dodatni podaci koje čvorovi u stvarnosti možda sadrže. Najprirodnija reprezentacija stabla, koja najočiglednije odražava njegovu logičku strukturu, bila bi da se svaki čvor stabla predstavlja objektom koji obuhvata polje ključa, više polja u kojima se nalaze pokazivači na svako dete tog čvora i, eventualno, polje za pokazivač na roditelja čvora. Međutim, pošto je ovde reč o opštim stablima, odnosno broj dece po čvoru može biti vrlo različit i nije unapred ograničen, ovaj pristup nije najbolji jer proizvodi loše iskorišćenje memorijskog prostora. Zato se obično koriste dva druga načina za predstavljanje opštih stabala o kojima govorimo u nastavku.

Predstavljanje stabla listama dece čvorova. Jedan način za prevazilaženje problema nepoznatog broja dece nekog čvora u opštem stablu je formiranje povezane liste dece svakog čvora. Ovaj pristup je važan, jer se može lako generalizovati za predstavljanje složenijih struktura kao što su

grafovi. U desnom delu slike 6.4 prikazano je kako se stablo T u levom delu te slike može predstaviti na ovaj način.



SLIKA 6.4: Predstavljanje stabla listama dece čvorova.

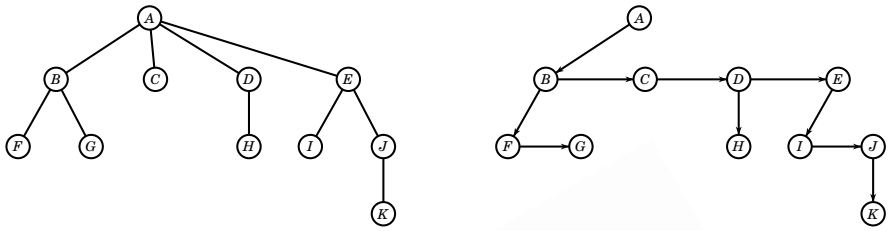
Kao što se vidi sa slike 6.4, u ovoj reprezentaciji stabla imamo jedan niz pokazivača t čiji elementi predstavljaju čvorove i zato je njegova veličina jednaka ukupnom broju čvorova u stablu. Pored toga, svaki element niza t pokazuje na početak povezane liste čvorova stabla. Pri tome, elementi liste na koju pokazuje element niza t_i su čvorovi-deca čvora i tako da poredak elemenata u listi odgovara poretku dece sleva na desno u stablu. Na primer, na slici 6.4 element t_5 pokazuje na listu $8 \rightarrow 9$ pošto su čvorovi 8 i 9, time redom, deca čvora 5.

Jedan nedostatak ovog načina predstavljanja stabla je to što ne možemo lako da konstruišemo veća stabla od manjih. To je posledica toga što svako stablo ima svoj niz pokazivača za svoje čvorove. Prema tome, da bismo napravili novo stablo, recimo, od datog čvora kao korena i dva data podstabla T_1 i T_2, morali bismo da kopiramo T_1 i T_2 u treće stablo i da dodamo listu za novi koren čija su deca koreni za T_1 i T_2.

Predstavljanje stabla pomoću sasvim levog deteta i desnog brata čvorova. Drugo rešenje za problem predstavljanja opštih stabala je *predstavljanje sasvim levog deteta i desnog brata* svakog čvora stabla. U ovoj reprezentaciji svaki čvor pored ključa sadrži samo još dva pokazivača: jedan pokazuje na njegovog sasvim levog deteta i drugi na njegovog desnog brata. Naravno, ovi pokazivači imaju specijalnu vrednost `null` ukoliko čvor nema dece ili nema desnog brata.

Ovaj način predstavljanja opšteg stabla je ilustrovan u desnom delu na slici 6.5 za stablo koje je prikazano u levom delu. Na toj slici je za svaki

čvor stabla strelicom nadole označen pokazivač na sasvim levog deteta, a strelicom nadesno je označen pokazivač na desnog brata.



SLIKA 6.5: Predstavljanje stabla pokazivačima na sasvim levog deteta i desnog brata čvorova.

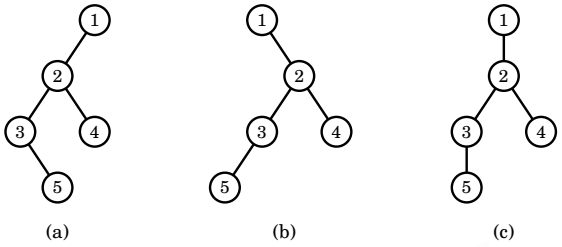
Sve operacije nad stablima, osim određivanja roditelja datog čvora, mogu se efikasno realizovati ako je stablo predstavljeno pomoću sasvim levog deteta i desnog brata svih čvorova. A ako je potrebno i da operacija određivanja roditelja bude efikasna, to se može lako rešiti dodavanjem još jednog pokazivača svakom čvoru koji direktno pokazuje na njegovog roditelja.

6.2 Binarna stabla

Binarno stablo je stablo u kojem nijedan čvor nema više od dvoje dece. Ova deca se zovu *levo dete* i *desno dete*, pošto se na crtežima prikazuju levo i desno od čvora koji je njihov roditelj. Formalno, binarno stablo je stablo u kojem svaki čvor ili nema decu, ili ima samo levo dete, ili ima samo desno dete, ili ima i levo i desno dete.

Obratite pažnju na to da pojam binarnog stabla nije isti kao za obično stablo koje smo definisali u prethodnom odeljku, jer svako dete u binarnom stablu mora biti levo dete ili desno dete. Ovo zapažanje je ilustrovano na slici 6.6, gde imamo dva različita binarna stabla i jedno obično stablo koje izgleda vrlo slično ovim binarnim stablima.

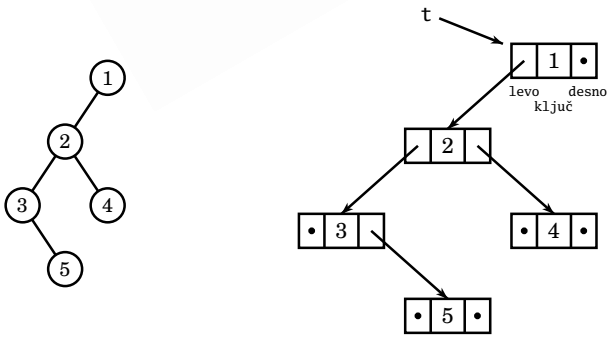
Primetimo da stabla pod (a) i (b) na slici 6.6 nisu ista binarna stabla, niti je neko od njih uporedivo sa običnim stablom pod (c). Na primer, u binarnom stablu pod (a) čvor 2 je levo dete čvora 1, a čvor 1 nema desno dete, dok u binarnom stablu pod (b) čvor 1 nema levo dete ali ima čvor 2 kao desno dete. Sa druge strane, u običnom stablu pod (c) nemamo definisano levo ili desno dete — čvor 2 je samo dete čvora 1, niti levo niti desno.



SLIKA 6.6: Razlike između binarnih i običnih stabala.

Uprkos ovim malim konceptualnim razlikama, osnovna terminologija opštih stabala prenosi se i na binarna stabla. Svi osnovni pojmovi koje smo definisali za obična stabla imaju smisla i za binarna stabla.

Predstavljanje binarnog stabla je umnogome pojednostavljeno zahvaljujući činjenici da svaki čvor binarnog stabla ima najviše dvoje dece. U praksi se za predstavljanje binarnih stabala skoro isključivo koriste pokazivači, jer je takva njihova reprezentacija prirodna i jednostavna. Preciznije, binarno stablo je organizovano u obliku povezane strukture podataka u kojoj je svaki čvor predstavljen jednim objektom. Taj objekat ima polje `ključ`, koje sadrži ključ čvora, kao i dva pokazivačka polja `levo` i `desno`, koja pokazuju na objekte koji predstavljaju levo i desno dete čvora. Ako neko dete nije prisutno, odgovarajuće pokazivačko polje sadrži vrednost `null`. Stablu T se kao celini pristupa preko posebnog spoljašnjeg pokazivača `t` koji nije deo stabla i koji pokazuje na koren stabla. Ovaj način predstavljanja binarnog stabla je ilustrovan u desnom delu na slici 6.7 za stablo koje je prikazano u levom delu.

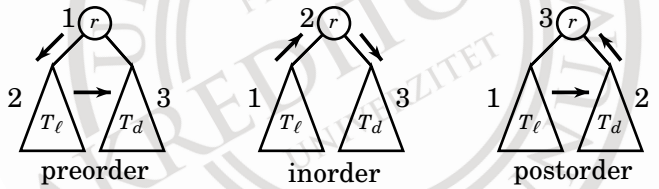


SLIKA 6.7: Predstavljanje binarnog stabla.

Obilazak binarnog stabla

Važna operacija nad nelinearnom strukturom stabla je neka aktivnost koja se izvršava nad podacima (ključevima) svih čvorova binarnog stabla. Ta operacija se neutralno naziva *obilazak* binarnog stabla i sastoji se kretanja duž grana stabla od čvora do čvora radi sistematičnog „posećivanja" svih čvorova stabla kako se neki čvor ne bi ponovio. Konkretna aktivnost koja se obavlja nad podacima u čvoru v prilikom njegovog posećivanja zavisi od primene operacije obilaska stabla i može obuhvatati sve od samog prikazivanja podataka čvora v do neke mnogo složenije obrade podataka čvora v.

Tri najčešća sistematična postupka za obilazak binarnog stabla T od njegovog korena jesu obilazak čvorova u preorder, inorder i postorder redosledu. Oni se razlikuju samo po redosledu po kojem se posećuju naredni čvor i njegovo levo i desno podstablo. Na slici 6.8 su ilustrovana ova tri načina tako što je brojevima 1, 2 i 3 označen redosled po kojem se posećuju naredni čvor r (aktuelni koren) i njegovo levo i desno podstablo T_ℓ i T_d.



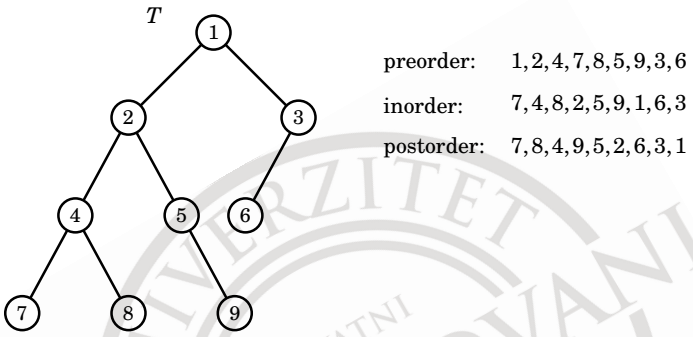
SLIKA 6.8: Tri načina obilaska binarnog stabla.

Formalna rekurzivna definicija preorder, inorder i postorder obilaska binarnog stabla T je:

1. Ako je stablo T prazno, ništa se ne obilazi jer T nema nijedan čvor.
2. Inače, neka je r koren stabla T sa (moguće praznim) levim i desnim podstablima T_ℓ i T_d čiji su koreni deca od r. Onda:
   - *preorder obilazak* čvorova stabla T sastoji se najpre od posete korena r, zatim od rekurzivnog obilaska čvorova stabla T_ℓ u preorder redosledu i na kraju od rekurzivnog obilaska čvorova stabla T_d u preorder redosledu;
   - *inorder obilazak* čvorova stabla T sastoji se najpre od rekurzivnog obilaska čvorova stabla T_ℓ u inorder redosledu, zatim od posete korena r i na kraju od rekurzivnog obilaska čvorova stabla T_d u inorder redosledu;

- *postorder obilazak* čvorova stabla T sastoji se najpre od rekurzivnog obilaska čvorova stabla T_ℓ u postorder redosledu, zatim od rekurzivnog obilaska čvorova stabla T_d u postorder redosledu i na kraju od posete korena r.

Na slici 6.9 su ilustrovana prethodna tri načina obilaska svih čvorova na konkretnom primeru jednog binarnog stabla.



SLIKA 6.9: Lista čvorova binarnog stabla T dobijena na preorder, inorder i postorder način.

Imajući u vidu standardni način predstavljanja binarnih stabala pomoću pokazivača, lako možemo napisati algoritme za preorder, inorder i postorder obilaske čvorova binarnog stabla. U nastavku ćemo pokazati samo algoritam za inorder obilazak binarnog stabla, pri čemu se za svaki čvor prikazuje njegov ključ. Algoritmi za ostala dva načina obilaska binarnog stabla ostavljaju se čitaocima za vežbu.

```
// Ulaz:  čvor t binarnog stabla
// Izlaz: lista ključeva podstabla sa korenom t u inorder redosledu
algorithm bt-inorder(t)

   if (t != null) then
      bt-inorder(t.levo);
      print(t.ključ);
      bt-inorder(t.desno);

   return;
```

Obratite pažnju na to da je rekurzivni algoritam `bt-inorder` neposredna implementacija rekurzivne definicije inorder obilaska binarnog stabla. Pri tome, obrada svakog čvora prilikom njegovog „posećivanja" sastoji se od prostog prikazivanja ključa čvora.

Nije teško pokazati da vreme izvršavanja algoritma bt-inorder za binarno stablo od n čvorova iznosi $O(n)$. (U stvari, to je tačno i za druga dva algoritma ostalih načina obilaska.) To sledi na osnovu činjenice da se algoritam bt-inorder poziva tačno dva puta za svaki čvor stabla — jednom za njegovo levo dete i jednom za njegovo desno dete. Pošto se deo algoritma bt-inorder koji ne obuhvata ove rekurzivne pozive izvršava za konstantno vreme, ukupno vreme izvršavanja biće $2n \cdot O(1) = O(n)$.

Ovo je dobra prilika da se osvrnemo i na naš „stil programiranja". Često ćemo jednostavne algoritme, kao što je to algoritam bt-inorder, izražavati u rekurzivnom obliku. Naravno, svaki rekurzivni algoritam se može napisati iterativno, ako ništa drugo simulirajući rekurziju pomoću steka. U stvari, u praksi je bolje pisati algoritme na iterativni nego na rekurzivni način, jer se tako dobija na efikasnosti izbegavanjem dodatnih implicitnih operacija koje svaka rekurzija nameće. Na primer, iterativnu verziju istog algoritma bt-inorder možemo napisati na sledeći način:

```
// Ulaz:  čvor t binarnog stabla
// Izlaz: lista ključeva podstabla sa korenom t u inorder redosledu
algorithm bt-inorder(t)

   stack-make(s);  // inicijalizacija praznog steka s
   repeat
      while (t != null) do
         push(t,s);
         t = t.levo;
      t = pop(s);
      print(t.ključ);
      t = t.desno;
   until (stack-empty(s) && (t == null));

   return;
```

Nije teško razumeti kako ovaj iterativni algoritam radi — on u suštini simulira rekurziju pomoću steka s. Ipak, iako su rekurzivni algoritmi manje efikasni (samo u apsolutnom smislu, ali ne i asimptotski), nama je bliža rekurzija pošto se dobijaju algoritmi koji su skoro uvek lakši za razumevanje i analiziranje.

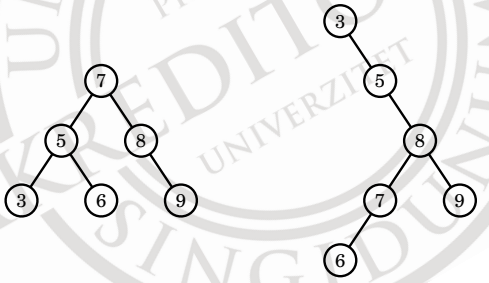
6.3 Binarna stabla pretrage

Binarno stablo pretrage je binarno stablo u kojem svaki čvor v ima sledeće *BSP svojstvo*: svi čvorovi u levom podstablu čvora v imaju ključeve

koji su manji od ključa čvora v, a svi čvorovi u desnom podstablu čvora v imaju ključeve koji su veći od ključa čvora v. BSP svojstvo binarnog stabla T možemo dakle izraziti na sledeći način: za svaki čvor v stabla T, ako je u čvor u levom podstablu čvora v, onda je `v.ključ > u.ključ`; a ako je w čvor u desnom podstablu čvora v, onda je `v.ključ < w.ključ`. (Ovde pretpostavljamo pojednostavljen slučaj da su svi ključevi različiti.)

Obratite pažnju na to da u binarnom stablu pretrage implicitno podrazumevamo da se ključevi čvorova mogu međusobno upoređivati. To znači da ključevi pripadaju nekom totalno uređenom skupu, tj. skupu nad čijim elementima je definisana relacija „manje od". Primeri ovih skupova obuhvataju skupove celih i realnih brojeva sa uobičajenom relacijom poretka brojeva, kao i skup znakovnih stringova sa leksikografskim poretkom.

Na slici 6.10 su prikazana dva binarna stabla pretrage koja sadrže isti skup vrednosti ključeva. Kao što ćemo videti, vreme izvršavanja većine operacija nad binarnim stablom pretrage je proporcionalno visini stabla. U tom smislu, desno stablo na slici 6.10 je manje efikasno.



SLIKA 6.10: Dva binarna stabla pretrage sa istim ključevima.

BSP operacije

Tipične operacije nad binarnim stablom pretrage su traženje, dodavanje i uklanjanje čvora iz takvog stabla.

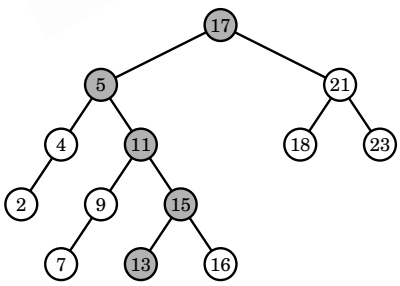
Traženje čvora. Najpre ćemo pažnju posvetiti problemu traženja čvora sa datim ključem x u binarnom stablu pretrage T. Da bismo brzo locirali čvor sa ključem x u stablu T, ili utvrdili da se čvor sa tim ključem ne nalazi u stablu, koristimo BSP svojstvo. Naime, ukoliko uporedimo x sa ključem korena stabla T, razlikuju se tri slučaja.

Prvi, najlakši slučaj je ako je ključ korena stabla T jednak x, jer smo onda našli traženi čvor i možemo završiti pretragu. U drugom slučaju, ako je ključ korena stabla T veći od x, prema BSP svojstvu možemo zaključiti da se čvor sa ključem x može naći samo u levom podstablu korena stabla T, jer su u desnom podstablu svi ključevi veći od ključa korena, pa time i od x. Zbog toga pretragu rekurzivno nastavljamo na isti način od korena levog podstabla korena stabla T. Najzad, u trećem slučaju koji je simetričan drugom, ako je ključ korena stabla T manji od x, onda se čvor sa ključem x može naći samo u desnom podstablu korena stabla T. Zbog toga pretragu rekurzivno nastavljamo na isti način od korena desnog podstabla korena stabla T.

Ovaj neformalni opis operacije traženja čvora u binarnom stablu pretrage preciziran je rekurzivnim algoritmom `bst-search` u nastavku. Ako su dati pokazivač t na neki čvor u binarnom stablu pretrage i ključ x, algoritam `bst-search` kao rezultat vraća pokazivač na čvor sa ključem x, ukoliko se takav čvor nalazi u podstablu čiji je koren t; u suprotnom slučaju, ukoliko takav čvor ne postoji, ovaj algoritam vraća pokazivač `null`.

```
// Ulaz:  čvor t binarnog stabla pretrage, ključ x
// Izlaz: pokazivač na čvor sa ključem x, ili null
algorithm bst-search(t, x)

   if ((t == null) || (t.ključ == x)) then
      return t;
   else if (t.ključ > x) then
      bst-search(t.levo, x);
   else
      bst-search(t.desno, x);
```

Na slici 6.11 je ilustrovan način rada algoritma `bst-search` na primeru jednog binarnog stabla pretrage.



SLIKA 6.11: Traženje čvora 13 u datom binarnom stablu pretrage.

Kao što se vidi sa slike 6.11, u algoritmu `bst-search` se pretraga započinje od korena stabla i prelazi se određen put nadole duž grana i čvorova u stablu. Za svaki čvor na tom putu, u algoritmu se upoređuje ključ tog čvora sa traženom vrednošću. Ako su oni jednaki, pretraga se završava. U suprotnom slučaju, pretraga se nastavlja u levom ili desnom podstablu čvora zavisno od toga da li je ključ čvora veći ili manji od tražene vrednosti. Pošto čvorovi na koje se rekurzivno dolazi tokom pretrage obrazuju put u stablu od korena pa do nekog lista u najgorem slučaju, vreme izvršavanja algoritma `bst-search` je $O(h)$ za stablo visine h.

Dodavanje čvora. Operacija dodavanja (kao i uklanjanja) čvora modifikuje binarno stablo pretrage tako da moramo paziti da ne narušimo BSP svojstvo u rezultujućem stablu. Imajući to u vidu, dodavanje novog čvora u binarno stablo pretrage može se ipak uraditi na relativno jednostavan način.

Da bismo novi čvor p dodali u binarno stablo pretrage T, u suštini najpre pokušavamo da lociramo ključ čvora p u T primenjujući postupak traženja tog ključa. Ako ga nađemo, ne treba ništa drugo da radimo, jer ne dodajemo duplikat čvora u stablo u skladu sa našom pojednostavljenom pretpostavkom da su svi ključevi stabla T različiti. U suprotnom slučaju, prema proceduri traženja ključa dolazimo do čvora koji nema levo ili desno dete što ukazuje da čvor p nije u stablu T. Nije teško videti da je to pravo mesto u stablu T za dodavanje novog čvora. Zbog toga na tom mestu dodajemo p kao novi čvor-list u T i povezujemo p tako da bude odgovarajuće dete čvora u kojem se zaustavila procedura traženja.

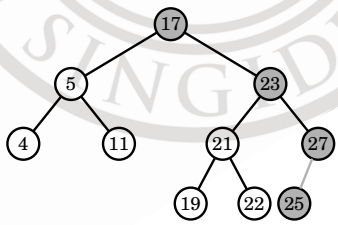
U ispravnost ovog postupka čitaoci se verovatno mogu lakše uveriti ukoliko se operacija dodavanja čvora formuliše u rekurzivnom obliku. Ako je T prazno stablo, ono postaje stablo koje se sastoji samo od novog čvora p. Ako stablo T nije prazno, a njegov koren ima ključ jednak ključu čvora p, tada se p već nalazi u stablu i zato ništa dodatno ne treba uraditi. Sa druge strane, ako stablo T nije prazno i njegov koren nema isti ključ kao čvor p, čvor p dodajemo u levo podstablo korena ako je ključ korena veći od ključa čvora p, ili dodajemo p u desno podstablo korena ako je ključ korena manji od ključa čvora p.

Rekurzivni algoritam `bst-insert` u nastavku predstavlja realizaciju ovog neformalnog opisa operacije dodavanja čvora. Ako su dati pokazivač t na čvor binarnog stabla pretrage i pokazivač na novi čvor p, tim algoritmom se dodaje čvor p u podstablo čiji je koren t. Pri tome se pretpostavlja

da polje p.ključ već sadrži željeni ključ i da polja p.levo i p.desno sadrže vrednost null.

```
// Ulaz:  čvor t binarnog stabla pretrage, novi čvor p
// Izlaz: čvor p dodat u podstablo sa korenom t
algorithm bst-insert(t, p)

   if (t == null) then
      t = p;
   else if (p.ključ < t.ključ) then
      t.levo = bst-insert(t.levo, p);
   else if (p.ključ > t.ključ) then
      t.desno = bst-insert(t.desno, p);
   else
      ; // duplikati se ne dodaju

   return t;
```

Na slici 6.12 je ilustrovan način rada algoritma bst-insert na primeru jednog binarnog stabla pretrage. Sa te slike se vidi da rad ovog algoritma počinje od korena stabla i da se prelazi određen put nadole u stablu baš kao u slučaju traženja čvora. Taj put se završava u čvoru čije levo ili desno dete treba da bude novi čvor. Stoga, baš kao u slučaju operacije traženja čvora, vreme izvršavanja algoritma bst-insert je $O(h)$ za stablo visine h.

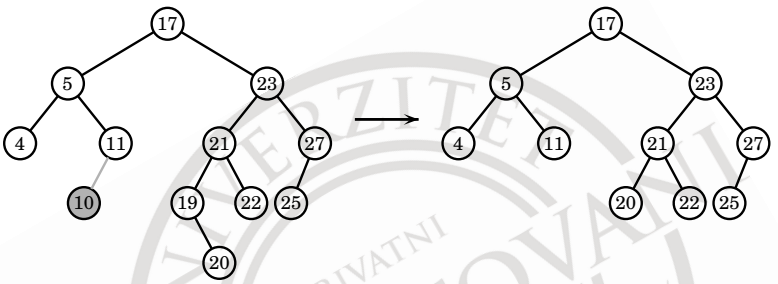


SLIKA 6.12: Dodavanje čvora 25 datom binarnom stablu pretrage.

U algoritmu bst-insert se srećemo sa jednom tehničkom poteškoćom koja se sastoji u tome što kada novi čvor treba zameniti umesto trenutno praznog čvora, potrebno je vratiti se i ažurirati jedno od pokazivačkih polja u čvoru roditelja novog čvora. To je razlog zašto prenosimo povratnu informaciju o rezultujućem podstablu posle dodavanja čvora. Drugim rečima, algoritam bst-insert vraća pokazivač na koren podstabla sa novododatim čvorom. Međutim, to je istovremeno jedan od uzroka relativne neefikasnosti rekurzivne implementacije, budući da samo poslednji čvor treba da se

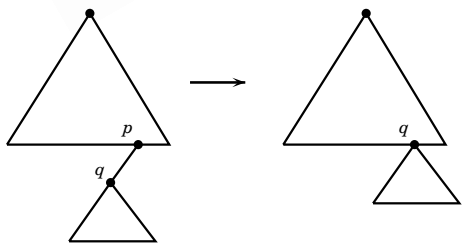
ažurira. Kod svih ostalih čvorova algoritam vraća isti pokazivač koji je primio, tako da kod tih čvorova faktički nema stvarnih promena.

Uklanjanje čvora. Uklanjanje čvora p iz binarnog stabla pretrage je malo komplikovanije od traženja ili dodavanja čvora. Pre svega, ako je p spoljašnji čvor (list) u stablu, možemo jednostavno ukloniti taj čvor iz stabla. Na slici 6.13 je prikazano uklanjanje spoljašnjeg čvora (lista) iz jednog binarnog stabla pretrage.



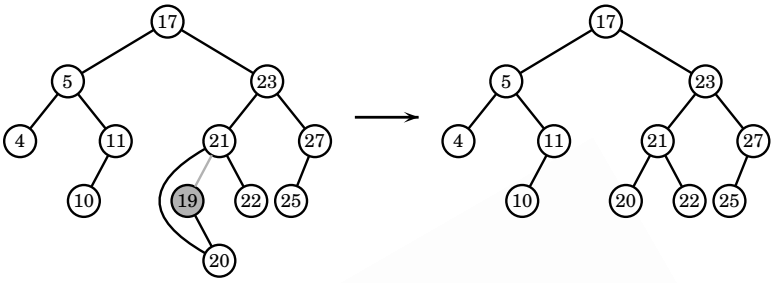
SLIKA 6.13: Uklanjanje spoljašnjeg čvora (lista) 10 iz datog binarnog stabla pretrage.

Međutim, ako je p unutrašnji čvor, ne možemo ga prosto ukloniti jer ćemo time prekinuti strukturu stabla. Zbog toga, stablo se mora restrukturirati tako da čvor p bude uklonjen, ali i da BSP svojstvo za preostale čvorove ne bude narušeno. A da bi se to postiglo, razlikuju se dva opšta slučaja. U prvom slučaju, kada čvor p ima samo jedno dete q, čvor p može se zameniti tim detetom. Preciznije, dete q se povezuje tako da bude novo dete roditelja čvora p umesto starog deteta — čvora p. Ova opšta transformacija binarnog stabla pretrage je ilustrovana na slici 6.14.



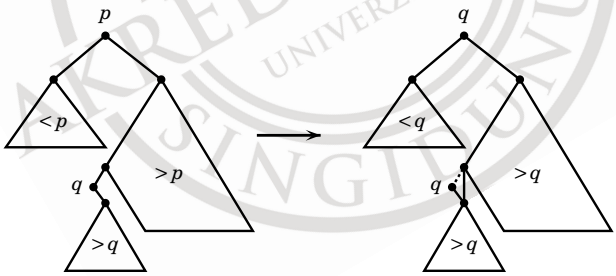
SLIKA 6.14: Opšti slučaj uklanjanja unutrašnjeg čvora p sa jednim detetom iz binarnog stabla pretrage.

Na slici 6.15 je prikazan konkretan primer uklanjanja unutrašnjeg čvora sa jednim detetom iz binarnog stabla pretrage.



SLIKA 6.15: Uklanjanje unutrašnjeg čvora 19 sa jednim detetom iz datog binarnog stabla pretrage.

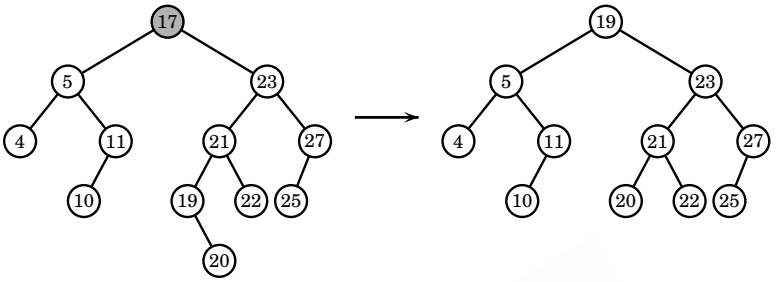
U drugom slučaju, kada unutrašnji čvor p ima dvoje dece, jedan način za njegovo uklanjanje je da se najpre nađe najmanji čvor q u desnom podstablu čvora p. (Druga mogućnost je da se nađe najveći čvor u levom podstablu čvora p.) Zatim se zamenjuje čvor p čvorom q i uklanja se čvor q iz desnog podstabla. Ovaj opšti pristup je ilustrovan na slici 6.16.



SLIKA 6.16: Opšti slučaj uklanjanja unutrašnjeg čvora p sa dvoje dece iz binarnog stabla pretrage.

Primetimo da BSP svojstvo nije narušeno u dobijenom stablu, jer je čvor p veći od svih čvorova u njegovom levom podstablu, a čvor q je veći od čvora p, kao i bilo koji drugi čvor iz desnog podstabla čvora p. Dakle, što se tiče levog podstabla čvora p, čvor q može zameniti čvor p. Ali q je odgovarajući čvor i što se tiče desnog podstabla čvora p, jer je q najmanji čvor u tom desnom podstablu.

Na slici 6.17 je ilustrovan konkretan primer uklanjanja čvora sa dvoje dece iz binarnog stabla pretrage.



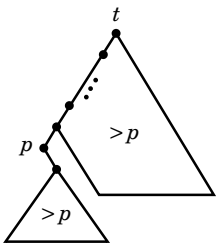
SLIKA 6.17: Uklanjanje unutrašnjeg čvora 17 sa dvoje dece iz datog binarnog stabla pretrage.

Pre predstavljanja kompletnog algoritma za operaciju uklanjanja čvora iz binarnog stabla pretrage, navodimo pomoćni algoritam koji uklanja *najmanji* čvor iz nepraznog podstabla sa korenom t. Najmanji čvor u podstablu sa korenom t je lako ukloniti, pošto taj čvor sigurno nema levo dete (jer bi ono moralo biti manje) i zato uvek imamo jedan od dva prosta slučajeva za uklanjanje čvora koji je bez dece ili ima samo jedno dete.

```
// Ulaz:  čvor t binarnog stabla pretrage
// Izlaz: uklonjen najmanji čvor p iz podstabla sa korenom t
algorithm bst-deletemin(t)

   if (t.levo == null) then
      p = t;
      return t.desno;
   else
      t.levo = bst-deletemin(t.levo);
      return t;
```

U rekurzivnom algoritmu `bst-deletemin` se u podstablu sa korenom t traži najmanji čvor tako što se prate leva deca od čvora t dok se ne naiđe na čvor p koji nema levo deto. To se prepoznaje time što je za taj čvor pokazivač `p.levo` jednak `null`. Slika 6.18 sugeriše zašto je taj čvor najmanji. Naime, prateći put od t do p, koji predstavlja levu ivicu podstabla sa korenom t, znamo da svi preci čvora p na tom putu, kao i čvorovi u njihovim desnim podstablima, jesu veći od p. Veći su i svi čvorovi u desnom podstablu čvora p, čime su iscrpljeni svi čvorovi različiti od p u podstablu sa korenom t.

U algoritmu `bst-deletemin` se kao rezultat vraća i pokazivač na uklonjeni najmanji čvor preko globalne promenljive p. Razlog za to je da bi algoritam u kome se poziva `bst-deletemin` mogao da eventualno iskoristi



SLIKA 6.18: Nalaženje najmanjeg čvora u binarnom stablu pretrage.

taj najmanji čvor za svoje potrebe. Jedan takav algoritam je rekurzivni algoritam bst-delete prikazan u nastavku. On predstavlja kompletnu realizaciju operacije uklanjanja čvora sa datim ključem x iz binarnog stabla pretrage na čiji koren ukazuje pokazivač t. Primetimo da algoritam bst-delete ne radi ništa ukoliko se čvor sa datim ključem ne nalazi u stablu.

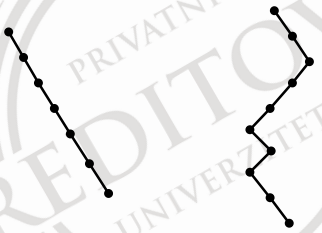
```
// Ulaz:  čvor t binarnog stabla pretrage, ključ x
// Izlaz: uklonjen čvor sa ključem x iz stabla sa korenom t
algorithm bst-delete(t, x)

   if (t != null) then  // neprazno stablo, naći ključ x
      if (t.ključ > x) then
         t.levo = bst-delete(t.levo, x);
      else if (t.ključ < x) then
         t.desno = bst-delete(t.desno, x);
      else              // t.ključ = x, ukloniti čvor t
         p = t;
         if (t.levo == null) then
            return t.desno;
         else if (t.desno == null) then
            return t.levo;
         else           // čvor sa dva deteta
            t.desno = bst-deletemin(t.desno);
            t.ključ = p.ključ;

   return t;
```

U algoritmu bst-delete se takođe počinje od korena binarnog stabla pretrage i prelazi određen put niz stablo da bi se najpre našao čvor koji treba ukloniti. U algoritmu se zatim uklanja taj čvor primenjujući odgovarajući postupak zavisno od broja njegove dece. Najgori slučaj je kada čvor za uklanjanje ima dvoje dece, jer se tada u algoritmu prelazi dodatni put dalje niz stablo da bi se pronašao najmanji čvor u desnom podstablu čvora

koji se uklanja. U svakom slučaju dakle, put koji se prelazi nije duži od visine stabla. Prema tome, kao i za operacije traženja i dodavanja čvora, vreme izvršavanja algoritma `bst-delete` je $O(h)$ za stablo visine h.

Efikasnost BSP operacija. Pokazali smo da se sve tri osnovne operacije nad binarnim stablom pretrage izvršavaju za vreme $O(h)$, pri čemu je h visina stabla. Međutim, visina binarnog stabla pretrage se menja kako se čvorovi dodaju i uklanjaju. Da bismo zato našu analizu učinili preciznijom, moramo oceniti visinu tipičnog binarnog stabla pretrage sa n čvorova.

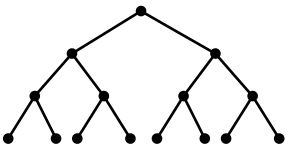
U najgorem slučaju, svi čvorovi stabla se mogu nalaziti na jednom putu. To bismo dobili, na primer, kada bismo n ključeva dodali u rastućem redosledu u početno prazno stablo. Drugi degenerativni slučajevi su putevi koji vijugaju levo ili desno kod nekih unutrašnjih čvorova. Na slici 6.19 su prikazana dva primera ovih najgorih scenarija.



SLIKA 6.19: Najgora binarna stabla pretrage.

Visina svih ovih degenerativnih binarnih stabla pretrage sa n čvorova je očigledno $n-1$. Na osnovu toga možemo zaključiti da se osnovne BSP operacije izvršavaju za linearno vreme $O(n)$ u najgorem slučaju.

U najboljem slučaju, binarno stablo pretrage predstavlja *puno* binarno stablo. U takvom stablu, svi listovi se nalaze samo na poslednjem nivou i svi unutrašnji čvorovi imaju tačno dvoje dece. Na slici 6.20 je prikazano puno binarno stablo sa 15 čvorova.



SLIKA 6.20: Puno binarno stablo.

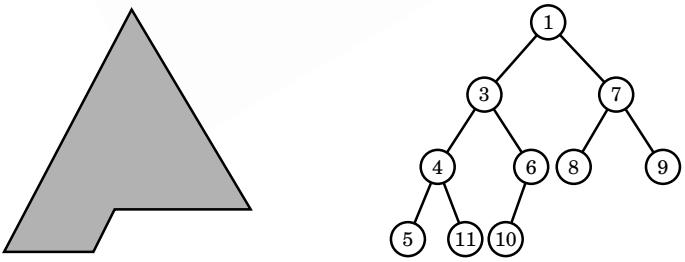
U punom binarnom stablu visine h, jednostavnom indukcijom po visini stabla može se pokazati da je njegov broj čvorova jednak $n = 2^{h+1} - 1$. Dakle, u takvom stablu je $n + 1 = 2^{h+1}$ ili, ekvivalentno, $h + 1 = \log(n + 1)$, odnosno $h = \log(n + 1) - 1$. Drugim rečima, vreme izvršavanja osnovnih BSP operacija je $O(\log n)$ u najboljem slučaju.

U praksi nije čest nijedan od ova dva slučaja, pa se može postaviti pitanje šta je tipičan slučaj između ova dva ekstrema? To jest, da li je tipično vreme bliže linearnom ili logaritamskom vremenu u odnosu na broj čvorova stabla? Odgovor na to pitanje može se dobiti složenijom matematičkom analizom koju ovde izostavljamo: uz relativno razumne pretpostavke verovatniji je bolji slučaj, pa se osnovne BSP operacije tipično izvršavaju za logaritamsko vreme.

6.4 Binarni hipovi

Binarni hipovi su binarna stabla kod kojih svi čvorovi zadovoljavaju specijalno svojstvo, slično kao kod binarnih stabala pretrage. Ali uz to, odgovarajuća binarna stabla kod binarnih hipova moraju dodatno imati specijalan oblik i biti što više razgranata (balansirana).

Preciznije, binarno stablo se naziva *kompletno* ako ima čvorove na svim nivoima osim možda poslednjem. Na poslednjem nivou, na kojem neki čvorovi mogu nedostajati, svi postojeći čvorovi moraju biti levo od onih koji nedostaju. Na slici 6.21 su prikazani opšti oblik i konkretan primer kompletnog binarnog stabla.



SLIKA 6.21: Kompletno binarno stablo.

Binarni hip (ili kraće samo *hip*) je kompletno binarno stablo u kojem je ključ svakog čvora manji od ključeva dece tog čvora. Drugim rečima, za svaki čvor v kompletnog stabla T, ako su u i w deca čvora v, onda je

`v.ključ < u.ključ` i `v.ključ < w.ključ`. Ovo svojstvo se naziva *hip svojstvo*.

Obratite pažnju na to da hip svojstvo implicira da je ključ svakog čvora v u hipu manji od ključeva svih čvorova u podstablu sa korenom v. Pored toga, najmanji čvor (tj. čvor sa minimalnim ključem) mora biti koren hipa, ali manji čvorovi ne moraju biti na manjoj dubini od većih čvorova. Na primer, u binarnom hipu na slici 6.21 čvorovi 4, 5 i 6 se nalaze na većoj dubini od čvora 7. Drugo zapažanje je da se čvorovi na svakom putu od nekog čvora nadole niz stablo nalaze u sortiranom rastućem redosledu.

U stvari, hipovi koji zadovoljavaju prethodno navedeno hip svojstvo nazivaju se preciznije min-hipovi. Mogli bismo isto tako da definišemo max-hipove kod kojih je ključ svakog čvora veći od ključeva njegove dece. Nema ništa posebno što izdvaja min-hipove u odnosu na max-hipove. Oba oblika hipova se ravnopravno koriste u praksi i sva zapažanja uz očigledne izmene važe u oba slučaja.

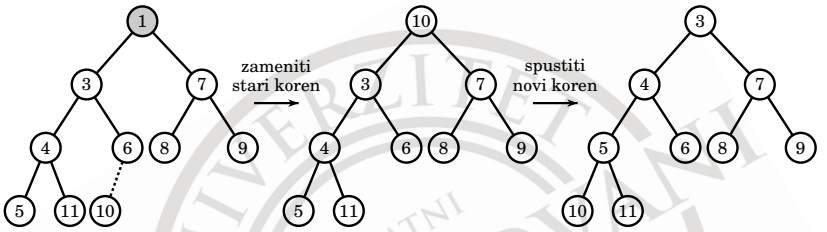
Hip operacije

Tipične operacije nad hipom su uklanjanje najmanjeg čvora (i vraćanje njegovog ključa), dodavanje novog čvora u hip, kao i stvaranje hipa od kompletnog binarnog stabla čiji se čvorovi nalaze u proizvoljnom poretku. U nastavku ovog odeljka govorimo o tome kako se ove operacije mogu realizovati.

Uklanjanje najmanjeg čvora. Najmanji čvor u binarnom hipu je lako pronaći, jer smo zapazili da taj čvor mora biti koren stabla. Međutim, ako prosto uklonimo koren, više nećemo imati stablo. Zbog toga moramo nekako naći novi ključ za koren tako da stablo i dalje bude skoro puno, ali i da hip svojstvo ne bude narušeno. Ovo obezbeđujemo sledećim postupkom koji se sastoji od dve faze.

U prvoj fazi, da bi se stablo sačuvalo da bude skoro puno, najpre se uzima ključ sasvim desnog lista na poslednjem nivou, zatim se uklanja taj list i na kraju se ključ korena privremeno zamenjuje sa uzetim ključem sasvim desnog lista. Time je kompletnost rezultujućeg stabla očevidno sačuvana, ali hip svojstvo novog korena može biti narušeno. Zato u drugoj fazi, da bi se obnovila važnost hip svojstva, ključ novog korena se pomera nadole niz stablo do njegovog pravog mesta. To znači da se taj ključ „gura” nadole niz stablo sve dok se može zameniti sa ključem jednog od deteta aktuelnog čvora koje ima manji ključ.

Preciznije, pretpostavimo da je x aktuelni čvor koji narušava hip svojstvo. To znači da je x veći od jednog ili oba svoja deteta. Ključ čvora x možemo zameniti ključem jednog od njegova dva deteta, ali pri tome moramo biti pažljivi. Ako x zamenimo sa manjim od njegova dva deteta, sigurno nećemo narušiti hip svojstvo između dva ranija deteta, od kojih je jedan sada postao roditelj drugog. Ponavljajući ovaj postupak za početni ključ, taj ključ će na kraju doći ili do nekog lista ili do čvora čija oba deteta imaju veći ključ. Na slici 6.22 je ilustrovan ovaj postupak uklanjanja najmanjeg čvora iz hipa sa slike 6.21.

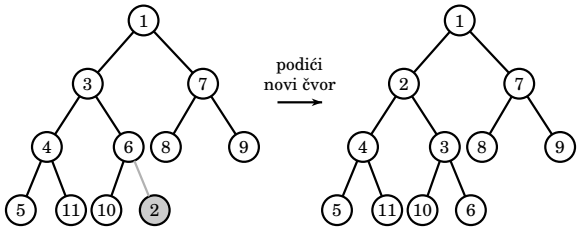


SLIKA 6.22: Uklanjanje najmanjeg čvora iz hipa.

Primetimo da se operacija uklanjanja najmanjeg čvora iz hipa izvršava za vreme $O(\log n)$ ukoliko hip ima n čvorova. Ovo sledi otuda što se postupak potiskivanja privremenog ključa korena ponavlja najviše h puta, gde je h visina stabla. A pošto je $h \leq \log(n+1)$ u kompletnom binarnom stablu od n čvorova i svaki korak spuštanja ključa se izvršava za konstantno vreme po čvoru, ukupno vreme je $O(\log n)$.

Dodavanje čvora. Razmotrimo sada kako se može dodati novi čvor u binarni hip, imajući u vidu da dobijeni hip mora biti skoro pun i da hip svojstvo u njemu mora ostati na snazi. Jedan način je da se novi čvor doda na poslednjem nivou na prvom mestu koje je sasvim levo moguće, počinjući novi nivo ako je poslednji nivo trenutno popunjen.

Time je kompletnost rezultujućeg stabla sačuvana, ali hip svojstvo roditelja novog čvora može biti narušeno. Zato se opet pokušava da se to neutrališe „guranjem" naviše novog ključa uz stablo. To znači da se međusobno zamenjuju novi ključ aktuelnog čvora i ključ roditelja tog čvora, ako roditelj ima manji ključ. Ovo zamenjivanje se ponavlja sve dok novi ključ ne dođe ili do korena ili do čvora čiji roditelj ima manji ključ. Na slici 6.23 je ilustrovan postupak dodavanja čvora 2 hipu sa slike 6.21.



SLIKA 6.23: Dodavanje novog čvora u hip.

Vreme izvršavanja potrebno za operaciju dodavanja čvora u hip je proporcionalno dužini puta koji novi čvor prelazi od početnog mesta na poslednjem nivou do svog pravog mesta. Ta dužina puta je očigledno ograničena visinom stabla hipa, pa je vreme izvršavanja jednako $O(\log n)$ za hip od n čvorova.

Konstruisanje hipa. Pretpostavimo da je dato kompletno binarno stablo čiji su čvorovi proizvoljno razmešteni tako da njihovi ključevi ne zadovoljavaju hip svojstvo. Operacija konstruisanja hipa sastoji se od preuređivanja ključeva tog stabla tako da ono na kraju bude hip.

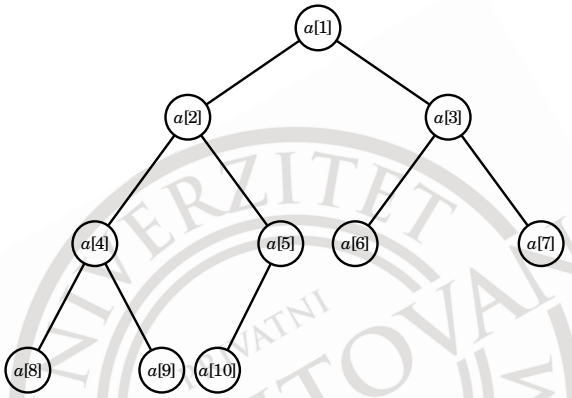
Jedan očigledan način da se ovo postigne je da se počne od praznog hipa i da mu se zatim redom dodaju čvorovi datog kompletnog binarnog stabla jedan po jedan. (Pri tome se, recimo, čvorovi datog stabla uzimaju tokom obilaska tog stabla jednim od tri standardna metoda.) Kako svaka operacija dodavanja čvora u hip proizvodi stablo koje je takođe binarni hip, to će se posle dodavanja svih čvorova na kraju dobiti traženi hip.

Međutim, ovaj postupak nije mnogo efikasan, jer bi za stablo od n čvorova njegovo vreme izvršavanja bilo $O(n \log n)$. To je zato što se njime obavlja n operacija dodavanja čvora, a svaka od tih operacija se izvršava za vreme $O(\log n)$ (što smo pokazali u prethodnom odeljku). Postoji bolje rešenje za ovaj problem čije je vreme izvršavanja samo $O(n)$. O njemu ćemo više govoriti u narednom odeljku koje je posvećeno jednoj konkretnoj realizaciji binarnog hipa.

Predstavljanje hipa pomoću niza

Pošto je hip skoro puno binarno stablo, ono se može efikasno predstaviti na neobičan način pomoću niza. Ideja ove reprezentacije je pozajmljena od načina na koji bi se čvorovi stabla poređali po nivoima. To znači da

elementi jednog niza predstavljaju čvorove binarnog hipa nivo po nivo, počinjući od korena i idući sleva na desno unutar jednog nivoa. Preciznije, prvi element niza sadrži koren stabla, drugi element niza sadrži levo dete korena, treći element niza sadrži desno dete korena, četvrti element niza sadrži levo dete levog deteta korena i tako dalje. Primer ovakve reprezentacije hipa sa 10 čvorova pomoću niza a je prikazan na slici 6.24.



SLIKA 6.24: Predstavljanje hipa sa 10 čvorova pomoću niza.

U opštem slučaju dakle, ako za $i \geq 1$ postoji levo dete čvora `a[i]`, ono je predstavljeno elementom `a[2*i]`, a njegovo desno dete, ako postoji, predstavljeno je elementom `a[2*i+1]`. Ekvivalentna interpretacija je da se roditelj čvora `a[i]` nalazi u elementu `a[i/2]` za $i > 1$.

Radi jednostavnosti, za ključeve čvorova hipa pretpostavljamo da su celi brojevi i zanemarujemo dodatne podatke koje čvorovi mogu imati. Na taj način se binarni hip može jednostavno predstaviti celobrojnim nizom. Naravno, u praksi često ovakva jednostavna reprezentacija nije moguća, ali se hip onda može predstaviti nizom objekata odgovarajućeg tipa. Ili još bolje, pošto se sve operacije nad hipom zasnivaju na mnogobrojnim zamenama podataka pojedinih čvorova, hip bismo mogli predstaviti u obliku niza pokazivača na objekte-čvorove i onda bi se zapravo zamenjivali samo ovi pokazivači.

Prema tome, u daljem izlaganju algoritama kojima se realizuju operacije nad hipom pretpostavljamo da je binarni hip predstavljen pomoću celobrojnog niza a tačno onako kako je to prikazano na slici 6.24. Da ne bismo preopteretili listu parametara algoritama o kojima ćemo govoriti, dalje pretpostavljamo da se aktuelna veličina hipa čuva u globalnoj promenljivoj

n. Na kraju, podrazumevamo i da je veličina niza uvek dovoljno velika za hip od n čvorova, kako u algoritmima ne bismo morali da proveravamo da li je prekoračena gornja granica niza.

Za operaciju dodavanja čvora u hip smo pokazali da se ona zasniva na postupku pomeranja novog čvora naviše uz stablo prema njegovom pravom mestu. Uobičajeno je da se za to kaže da novi čvor *isplivava* na svoje pravo mesto u stablu. Naredni rekurzivni algoritam predstavlja realizaciju tog postupka isplivavanja čvora `a[i]` u hipu koji je predstavljen nizom a.

```
// Ulaz:  čvor a_i hipa predstavljenog nizom a
// Izlaz: čvor a_i pomeren naviše do svog pravog mesta u hipu
algorithm heap-bubbleup(a, i)

   j = i / 2;  // a_j je roditelj čvora a_i
   if ((j > 0) && (a[i] < a[j])) then
      swap(a[i],a[j]);
      heap-bubbleup(a,j);

   return;
```

Algoritam `heap-bubbleup` se koristi u narednom algoritmu kojim se realizuje operacija dodavanja čvora x u hip od n čvorova koji je predstavljen nizom a.

```
// Ulaz:  hip predstavljen nizom a, čvor x
// Izlaz: hip predstavljen nizom a proširen za čvor x
algorithm heap-insert(a, x)

   n = n + 1;
   a[n] = x;
   heap-bubbleup(a,n);

   return;
```

Operacija uklanjanja najmanjeg čvora pomera privremeni koren niz stablo do njegovog pravog mesta. Uobičajeno je da se za to kaže da se privremeni čvor *prosejava* kroz stablo do svog pravog mesta u stablu. Naredni rekurzivni algoritam predstavlja realizaciju tog postupka prosejavanja čvora `a[i]` u hipu od n čvorova koji je predstavljen nizom a.

```
// Ulaz:  čvor a_i hipa predstavljenog nizom a
// Izlaz: čvor a_i pomeren nadole do svog pravog mesta u hipu
algorithm heap-siftdown(a, i)

   j = 2 * i;  // a_j je levo dete čvora a_i
```

```
   if ((j < n) && (a[j+1] < a[j])) then
      j = j + 1;

   // a_j je manje dete čvora a_i
   if ((j <= n) && (a[i] > a[j])) then
      swap(a[i],a[j]);
      heap-siftdown(a,j);

   return;
```

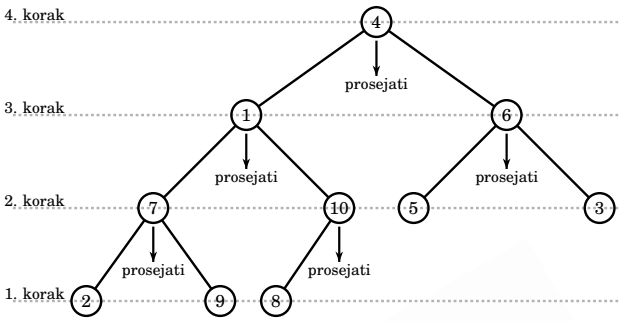
Sada je lako napisati algoritam kojim se iz min-hipa od n čvorova predstavljenog nizom a uklanja najmanji čvor (koren) i vraća njegov ključ.

```
// Ulaz:  hip predstavljen nizom a
// Izlaz: ključ uklonjenog korena hipa predstavljenog nizom a
algorithm heap-deletemin(a)

   x = a[1];
   a[1] = a[n];
   n = n - 1;
   heap-siftdown(a,1);

   return x;
```

Preostalo je da pokažemo kako se može formirati hip od niza čiji se elementi nalaze u proizvoljnom poretku. Pomenuli smo jedan očigledan, mada neefikasan postupak kojim se počinje od praznog hipa i zatim mu se redom dodaju elementi niza jedan po jedan. Ovo rešenje prepuštamo čitaocima za vežbu.

Bolji pristup je da se postupno konstruišu sve veći hipovi počinjući od čvorova na poslednjem nivou. Svi ovi čvorovi na poslednjem nivou su listovi stabla i svaki od njih trivijalno predstavlja hip sam za sebe. Zatim se konstruišu hipovi od podstabala čiji su koreni oni čvorovi koji se nalaze na pretposlednjem nivou. Za ovo je, u stvari, potrebno prosejati samo korene tih podstabala.

U narednom koraku se konstruišu hipovi od podstabala čiji su koreni oni čvorovi koji se nalaze na prvom nivou iznad prethodnog. Za ovaj korak je opet potrebno prosejati samo korene ovih podstabala. To je dovoljno pošto su njihova leva i desna podstabla već formirani hipovi. Nastavljajući ovaj postupak nivo po nivo i idući nagore dok se ne dođe do globalnog korena celog stabla, na kraju će se formirati čitav hip. Na slici 6.25 je ilustrovan ovaj način konstruisanja hipa od stabla neuređenih čvorova.



SLIKA 6.25: Konstruisanja hipa od stabla neuređenih čvorova.

Ova strategija je primenjena u upečatljivo jednostavnom i efikasnom algoritmu u nastavku kojim se konstruiše hip od n elemenata niza a koji se nalaze u proizvoljnom redosledu.

```
// Ulaz:  niz a od n neuređenih elemenata
// Izlaz: preuređen niz a tako da predstavlja binarni hip
algorithm heap-make(a)

   for i = n/2 downto 1 do
      heap-siftdown(a,i);

   return;
```

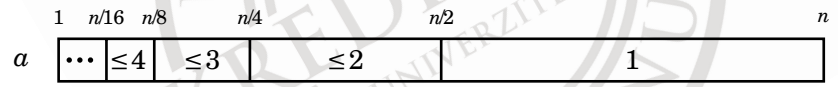
Analiza algoritma `heap-make` nije jednostavna, jer se u njemu $n/2$ puta poziva rekurzivni algoritam `heap-siftdown`.[1] Sa druge strane, vreme izvršavanja algoritma `heap-siftdown` nije lako oceniti na standardan način rešavanjem neke rekurentne jednačine o čemu smo govorili u odeljku 5.2. Zbog toga ćemo postupiti na drugi, ređe primenjivan način i izračunaćemo blisku gornju granicu za broj poziva algoritma `heap-siftdown` tokom rada algoritma `heap-make` za ulazni niz dužine n. Nije se teško uveriti da je taj ukupan broj poziva dominantan za vreme izvršavanja algoritma `heap-make`, jer se deo algoritma `heap-siftdown` van rekurzivnog poziva izvršava za konstantno vreme.

Ukoliko se dakle primeni ovaj pristup, pokazuje se da je vreme izvršavanja algoritma `heap-make` linearno, odnosno $T(n) = O(n)$. Da bismo dokazali ovaj pomalo iznenađujući rezultat, posmatraćemo najgori slučaj kada se radi o hipu koji je dat punim binarnim stablom visine h za neko $h \geq 0$.

[1] Ovde je tačnije reći da se algoritam `heap-siftdown` poziva $\lfloor n/2 \rfloor$ puta, ali smo već napomenuli da radi jednostavnosti zanemarujemo ovu malu nepreciznost.

Ekvivalentno, dužina niza a koji predstavlja to stablo iznosi $n = 2^{h+1} - 1$. Pored toga, mada algoritam `heap-make` početno ne prosejava listove tog stabla, pretpostavićemo da ih on ne preskače u svom radu. Drugim rečima, to je kao da u algoritmu `heap-make` imamo da `for` petlja počinje od n, a ne od $n/2$.

Za ovaj najgori slučaj možemo dakle zaključiti da se najpre algoritam `heap-siftdown` poziva za svaki od 2^h listova. Zatim, za 2^{h-1} čvorova na pretposlednjem nivou, algoritam `heap-siftdown` se poziva najviše dva puta, pošto se u ovoj fazi prosejavaju koreni podstabala visine jedan. Na sledećem gornjem nivou, koji ima 2^{h-2} čvorova, za svaki od njegovih čvorova se algoritam `heap-siftdown` poziva najviše tri puta, pošto se tada prosejavaju koreni podstabala visine dva. Nastavljajući rezonovanje na ovaj način, zaključujemo da se na nultom nivou korena celog stabla algoritam `heap-siftdown` poziva najviše $h+1$ puta, pošto se na kraju prosejava ovaj koren čitavog stabla koje ima visinu h. Na slici 6.26 je ilustrovana ova argumentacija tako što su prikazani delovi niza a i odgovarajući brojevi poziva algoritma `heap-siftdown` za podstabla čiji se koreni nalaze u tim delovima.



SLIKA 6.26: Ukupan broj poziva algoritma `heap-siftdown`.

Prema tome, ukoliko sa $S(h)$ označimo ukupan broj poziva algoritma `heap-siftdown` za puno binarno stablo visine h, onda je:

$$\begin{aligned} S(h) &\le 2^h \cdot 1 + 2^{h-1} \cdot 2 + 2^{h-2} \cdot 3 + \cdots + 2^1 \cdot h + 2^0 \cdot (h+1) \\ &= \sum_{i=1}^{h+1} 2^{h+1-i} \cdot i \\ &= 2^{h+1} \sum_{i=1}^{h+1} \frac{i}{2^i}. \end{aligned}$$

Da bismo ograničili poslednji zbir, ograničićemo njegovu beskonačnu varijantu $\sum_{i=1}^{\infty}(i/2^i)$:

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \cdots$$

$$= \frac{1}{2} + \left(\frac{1}{4} + \frac{1}{4}\right) + \left(\frac{1}{8} + \frac{1}{8} + \frac{1}{8}\right) + \left(\frac{1}{16} + \frac{1}{16} + \frac{1}{16} + \frac{1}{16}\right) + \cdots .$$

A da bismo ograničili ovaj beskonačni zbir, najpre ćemo ga preurediti u „trougaoni" zbir, zatim sabrati redove i, na kraju, sabrati poslednju kolonu iza znaka jednakosti:

$$\begin{aligned}
\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \cdots &= 1 \\
\frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \cdots &= \frac{1}{2} \\
\frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \cdots &= \frac{1}{4} \\
\frac{1}{16} + \frac{1}{32} + \cdots &= \frac{1}{8} \\
&\vdots
\end{aligned}$$

Kako je $1 + 1/2 + 1/4 + 1/8 + \cdots = 2$, to je

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = \sum_{i=0}^{\infty} \frac{1}{2^i} = 2 ,$$

i zato

$$\sum_{i=1}^{h+1} \frac{i}{2^i} \leq \sum_{i=1}^{\infty} \frac{i}{2^i} = 2 .$$

Stoga, vreme izvršavanja algoritma heap–make je:

$$\begin{aligned}
T(n) &= S(h) \cdot O(1) \\
&\leq 2^{h+1} \cdot 2 \cdot O(1) \\
&= 2(n+1) \cdot O(1) \\
&= O(n) ,
\end{aligned}$$

što je trebalo pokazati.

6.5 Primene stabala

Od mnogobrojnih primena stabala u računarstvo u ovom odeljku navodimo samo dva primera koja koristimo u daljem tekstu u poglavlju 8.

Redovi sa prioritetima

Red za čekanje sa prioritetima elemenata predstavlja strukturu podataka u kojoj je svakom elementu ove strukture pridružena određena numerička vrednost koja se naziva njegov *prioritet* (ili *ključ*). Klasičnu primenu ove strukture podataka nalazimo u operativnom sistemu sa deljenjem vremena čiji je osnovni zadatak da raspoređuje različite procese u sistemu radi izvršavanja od strane procesora. U realizaciji tog zadataka takav operativni sistem koristi red čiji su elementi pojedini procesi u računarskom sistemu koji čekaju za izvršavanje od strane procesora.

Međutim, svi trenutno aktivni procesi u sistemu ne moraju imati istu važnost i zato se klasifikuju prema svom prioritetu. Najviši prioritet verovatno imaju sistemski procesi kao što su upravljački programi za tastaturu ili mrežnu karticu. Zatim po važnosti dolaze korisnički procesi u kojima se izvršavaju interaktivne aplikacije. Na kraju dolaze pozadinski procesi kao što su neki program za pravljenje rezervnih kopija datoteka ili program za neko dugačko izračunavanje koji je namenski predviđen da radi sa niskim prioritetom.

Svaki proces u sistemu se zato predstavlja jednim objektom koji pored ostalih polja koja bliže opisuju proces sadrži još dva polja: jedno celobrojno polje za identifikacioni broj procesa i jedno celobrojno polje za prioritet procesa. Operativni sistem sa deljenjem vremena raspoređuje ove procese radi izvršavanja na osnovu njihovog prioriteta. Kada se u sistemu pokrene novi proces koji treba izvršiti, on najpre dobija svoj identifikacioni broj i prioritet. Ovaj proces se ne izvršava odmah, jer možda po svom prioritetu to ne zaslužuje, nego se dodaje u strukturu podataka reda svih procesa sa prioritetima koji čekaju na izvršavanje. Kada procesor postane besposlen, ili kada proces koji se trenutno izvršava iskoristi svoj kvantum vremena, operativni sistem određuje proces najvišeg prioriteta u redu procesa koji čekaju na izvršavanje i taj proces dodeljuje procesoru radi izvršavanja.

Na osnovu ovog primera može se zaključiti da su osnovne operacije nad redom sa prioritetima iste one koje se koriste za rad sa običnim redovima za čekanje ili stekovima. To su operacije dodavanja i uklanjanja elemenata, s tom razlikom što se iz reda sa prioritetima uklanja element koji ima najmanji prioritet. (Podsetimo se da se kod običnih redova uklanja „najstariji" element, dok se kod stekova uklanja „najnoviji" element.)

Red sa prioritetima Q je dakle struktura podataka koja se sastoji od skupa elemenata u kojem svaki element sadrži polje ključa koje označava prioritet elementa reda. Dve osnovne operacije nad redom sa prioritetima

Q su:

1. Operacija dodavanja `pq-insert(Q,x)` kojom se se u red sa prioritetima Q dodaje novi element x.
2. Operacija uklanjanja `pq-deletemin(Q)` kojom se u redu sa prioritetima Q najpre pronalazi element najnižeg prioriteta, a zatim se taj element uklanja iz reda Q i kao dodatni rezultat vraća se prioritet tog „najmanjeg" elementa.

Jedan efikasan način za implementaciju reda sa prioritetima je pomoću binarnog hipa o kome smo govorili u odeljku 6.4. Osnovne operacije dodavanja i uklanjanja elemenata za red sa prioritetima odgovaraju bez izmena algoritmima `heap-insert` i `heap-deletemin`, odnosno operacijama dodavanja i uklanjanja elemenata za binarni hip. Primetimo i da je vreme izvršavanja ovih operacija logaritamsko, odnosno $O(\log n)$ za red sa prioritetima od n elemenata.

U praksi je struktura podataka reda sa prioritetima obično složenija zbog toga što treba obezbediti dodatne operacije za složenije algoritme koji koriste neki red sa prioritetima. Nepotpun spisak ovih dodatnih operacija je:

- `pq-make(Q,a)`: formirati red sa prioritetima Q od datih elemenata u nizu a.
- `pq-changekey(Q,x,p)`: promeniti prioritet datog elementa x u redu sa prioritetima Q tako da bude jednak p.
- `pq-delete(Q,x)`: ukloniti dati proizvoljni element x iz reda sa prioritetima Q.

Ako se podsetimo strukture podataka binarnog hipa iz odeljka 6.4, možemo primetiti da se sve ove dodatne operacije mogu efikasno realizovati ukoliko je red sa prioritetima predstavljen binarnim hipom u obliku niza. Naime, da bismo konstruisali red sa prioritetima od datih elemenata, možemo iskoristiti algoritam `heap-make` koji se izvršava za linearno vreme. Da bismo promenili prioritet datog elementa koji se nalazi negde unutar binarnog hipa, možemo iskoristiti algoritam `heap-siftdown` za njegovo pomeranje nadole ako mu se prioritet povećava, ili algoritam `heap-bubbleup` za njegovo pomeranje naviše ako mu se prioritet smanjuje. Da bismo uklonili proizvoljni element iz binarnog hipa, možemo primeniti postupak sličan onom koji se koristi za uklanjanje minimalnog elementa. Obratite pažnju na to da u svim ovim operacijama dolazi do zamenjivanja elemenata duž jednog puta u hipu, u najgorem slučaju od vrha hipa pa sve do njegovog

dna ili obrnuto. To znači da je vreme izvršavanja ovih operacija takođe logaritamsko.

Disjunktni skupovi

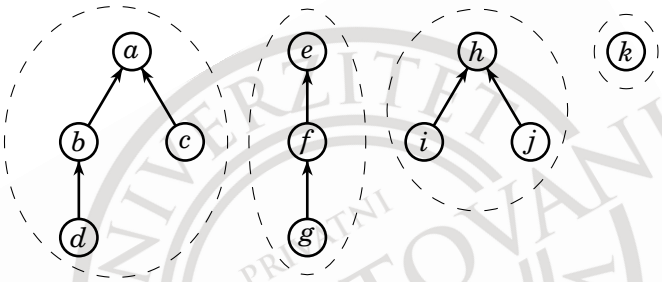
U nekim primenama je potrebno da podaci budu organizovani u više grupa pri čemu su važne dve operacije: jedna kojom se određuje kojoj grupi pripada neki podatak i druga kojom se objedinjuju dve date grupe u jednu. U tim slučajevima dolazi do izražaja struktura podataka disjunktnih skupova.

Apstraktna struktura podataka *disjunktnih skupova* predstavlja n elemenata grupisanih u kolekciju od najviše n disjunktnih skupova. Svaki skup u kolekciji sadrži dakle međusobno različite elemente koji su objekti istog tipa. Efikasne operacije potrebne za rad sa ovom strukturom podataka su:

- *Konstruisanje skupa od jednog elementa.* Operacijom `ds-make(x)` se konstruiše novi skup čiji je jedini član dati element x. Pošto su skupovi u kolekciji disjunktni, podrazumeva se da se x ne nalazi već u nekom drugom skupu.
- *Određivanje matičnog skupa.* Operacijom `ds-find(x)` se dobija identifikacija jedinstvenog skupa u kolekciji koji sadrži dati element x.
- *Objedinjavanje dva skupa.* Za data dva elementa x i y koji pripadaju različitim disjunktnim skupovima S_x i S_y u kolekciji, operacijom `ds-union(x,y)` objedinjuju se skupovi S_x i S_y u jedan novi skup koji predstavlja njihovu uniju. Preciznije, formira se treći skup S_z takav da je $S_z = S_x \cup S_y$, a kako skupovi u kolekciji moraju biti disjunktni, skupovi S_x i S_y se uklanjaju iz kolekcije nakon njihovog objedinjavanja.

Primenom operacije određivanja matičnog skupa datog elementa kao rezultat se dobija *identifikacija* (ili ime) tog skupa. Tačna priroda identifikacije skupova u kolekciji disjunktnih skupova je prilično fleksibilna pod uslovom da je dosledna. To znači da rezultat operacija `ds-find(x)` i `ds-find(y)` mora biti jednak ukoliko elementi x i y pripadaju istom skupu, a različit u suprotnom slučaju. Obično se svaki skup u kolekciji identifikuje nekim svojim članom koji se naziva *predstavnik* tog skupa.

Predstavljanje disjunktnih skupova pomoću stabala. Jedan efikasan način za predstavljanje disjunktnih skupova je pomoću kolekcije (opštih) korenskih stabala, pri čemu svaki čvor stabla predstavlja jednog člana nekog skupa i svako stablo predstavlja jedan skup iz kolekcije. U svakom stablu ove reprezentacije jedan čvor sadrži, pored elementa skupova koga predstavlja, još samo pokazivač na svog roditelja. Skup koga neko stablo predstavlja identifikuje se preko predstavnika koji se nalazi u korenu tog stabla. Na slici 6.27 je ilustrovana ovakva reprezentacija za kolekciju disjunktnih skupova $\{a,b,c,d\}$, $\{e,f,g\}$, $\{h,i,j\}$ i $\{k\}$.



SLIKA 6.27: Disjunktni skupovi predstavljeni kolekcijom stabala.

Tri karakteristične operacije za rad sa disjunktnim skupovima predstavljenim kolekcijom stabala mogu se algoritamski realizovati se na sledeći način:

- U algoritmu `ds-make(x)` se za dati objekat x sa odgovarajućim elementom skupa prosto konstruiše jedan čvor stabla.
- U algoritmu `ds-find(x)` se prate roditeljski pokazivači od čvora x dok se ne dođe do korena stabla. Za identifikaciju (predstavnika) matičnog skupa čvora x vraća se kao rezultat element koji se nalazi u korenu stabla.
- U algoritmu `ds-union(x,y)` se objedinjuju dva disjunktna skupa predstavljena pomoću dva stabla sa korenima x i y tako što se koren stabla manje visine vezuje za koren stabla veće visine. Drugim rečima, koren stabla veće visine postaje roditeljski čvor za koren stabla manje visine. (U slučaju dva stabla iste visine proizvoljno se bira koren jednog stabla da bude roditeljski čvor drugog.) Ovaj način objedinjavanja stabala naziva se *objedinjavanje po visini*.

Ovaj neformalni opis algoritamske realizacije operacija za rad sa disjunktnim skupovima pokazuje da svakom čvoru u stablima kolekcije treba

pridružiti informaciju o njegovoj visini. Zbog toga svaki čvor x sadrži celobrojno polje h za visinu tog čvora. Pored toga, svaki čvor x sadrži polje ključ u kome se nalazi odgovarajući element skupa, kao i polje p u kome se nalazi pokazivač na roditelja čvora x. Naravno, ovo pokazivačko polje korena svakog stabla sadrži vrednost null.

Kada se operacijom ds-make konstruiše stablo od jednog čvora, početna visina tog čvora je 0. Operacijom ds-find se visine čvorova stabla uopšte ne menjaju. Kod primene operacije ds-union mogu se desiti dva slučaja zavisno od visina korena stabala koja se objedinjuju. U prvom slučaju u kojem ovi koreni imaju nejednake visine, koren veće visine postaje roditelj korena manje visine, ali same visine ostaju nepromenjene. U drugom slučaju u kojem ovi koreni imaju jednake visine, proizvoljno se bira jedan od korena da bude roditelj drugog, ali se njegova visina uvećava za jedan.

```
// Ulaz:  objekat x sa odgovarajućim elementom skupa
// Izlaz: čvor x stabla u kolekciji stabala
algorithm ds-make(x)

   x.h = 0;
   x.p = null;

   return x;
```

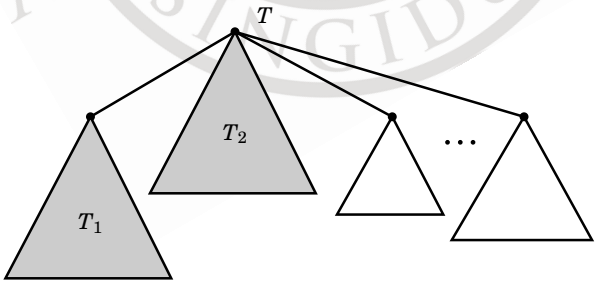
```
// Ulaz:  čvor x u kolekciji stabala
// Izlaz: predstavnik (identifikacija) skupa koji sadrži element x
algorithm ds-find(x)

   while (x.p != null) do
      x = x.p;

   return x;
```

```
// Ulaz:  koreni x i y dva stabla u kolekciji stabala
// Izlaz: stablo koje objedinjuje stabla koja sadrže x i y
algorithm ds-union(x, y)

   if (x.h > y.h) then
      y.p = x;
      return x;   // x je koren rezultujućeg stabla
   else
      x.p = y;
      if (x.h == y.h) then
         y.h = y.h + 1;
      return y;   // y je koren rezultujućeg stabla
```

Algoritmi `ds-make` i `ds-union` očevidno se izvršavaju za konstantno vreme. Pošto vreme izvršavanja algoritma `ds-find` zavisi od visina stabala u kolekciji, pokazaćemo da visina nekog stabla može rasti najviše logaritamski u odnosu na broj čvorova u tom stablu. Zbog toga kada se prelazi put od nekog čvora u stablu do njegovog korena radi nalaženja predstavnika odgovarajućeg skupa, vreme izvršavanja tog postupka je proporcionalno najviše logaritmu broja čvorova u datom stablu. Ova logaritamska granica visine nekog stabla u kolekciji stabala koja predstavljaju disjunktne skupove preciznije se izražava na sledeći način.

LEMA *Neka je T jedno stablo u kolekciji stabala koje je početno konstruisano algoritmom `ds-make` i zatim formirano (višestrukom) primenom algoritma `ds-union`. Ako je visina stabla T jednaka h, onda T ima bar 2^h čvorova.*

Dokaz: U dokazu koristimo indukciju po h. Bazni slučaj $h = 0$ je očigledan, jer stablo mora imati samo jedan čvor i $2^0 = 1$. Pretpostavimo da je tvrđenje leme tačno za neko $h \geq 0$ i posmatrajmo stablo T u kolekciji koje ima visinu $h+1$. U nekom trenutku tokom formiranja stabla T operacijama objedinjavanja po visini, visina tog stabla je prvi put dostigla vrednost $h+1$. Jedini način na koji je moglo da se tada dobije stablo visine $h+1$ je da je jedno stablo T_1 visine h postalo dete korena drugog stabla T_2 jednake visine h. Stablo T se sastoji dakle od stabala T_1 i T_2, i možda nekih drugih stabala koja su dodata kasnije. Izgled stabla T je ilustrovan na slici 6.28.



SLIKA 6.28: Stablo T visine $h+1$ obrazovano operacijama objedinjavanja po visini.

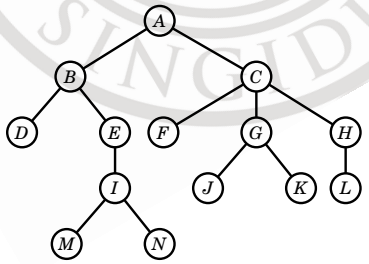
Stabla T_1 i T_2 po indukcijskoj pretpostavci imaju bar po 2^h čvorova. S obzirom na to da T sadrži čvorove stabala T_1 i T_2, i možda dodatne čvorove, T mora imati bar $2^h + 2^h = 2^{h+1}$ čvorova. Ovim je dokazan indukcijski korak, a time i tvrđenje leme. ■

Na osnovu ovog zapažanja sledi da ako neko stablo u kolekciji stabala koja predstavljaju disjunktne skupove ima m čvorova i visinu h, onda mora biti $m \geq 2^h$. Logaritmovanjem obe strane ove nejednakosti dobijamo $h \leq \log m$. Stoga vreme potrebno za prelaženje puta od nekog čvora u tom stablu do njegovog korena je $O(\log m)$. Prema tome, ako kolekcija disjunktnih skupova sadrži ukupno n elemenata, onda je naravno $m \leq n$, pa je vreme izvršavanja algoritma `ds-find` jednako $O(\log n)$.

Zadaci

1. Odogovorite na sledeća pitanja u vezi sa stablom na slici 6.29.

a) Koji čvorovi su listovi?
b) Koji čvor je koren?
c) Koji čvor je roditelj čvora C?
d) Koji čvorovi su deca čvora C?
e) Koji čvorovi su preci čvora E?
f) Koji čvorovi su potomci čvora E?
g) Koja je dubina čvora C?
h) Koja je visina čvora C?



SLIKA 6.29.

2. Navedite listu čvorova stabla na slici 6.29 u preorder, inorder i postoder redosledu.

3. Nacrtajte reprezentaciju izraza $\big((a+b)+c*(d+e)+f\big)*(g+h)$ u obliku stabla.

4. Napišite i analizirajte algoritam za izračunavanje dubine čvora v u stablu T.

5. Pokažite da je visina stabla T jednaka maksimalnoj dubini nekog lista u stablu T.

6. Dokažite sledeća tvrđenja o stablima:

a) Svako stablo sa n čvorova ima $n-1$ grana.

b) *Stepen čvora* u stablu je broj njegove dece. U svakom binarnom stablu je broj listova za jedan veći od broja čvorova stepena dva.

c) Binarno stablo je *pravilno* binarno stablo ako svaki čvor ima nula ili dvoje dece. Ako takvo stablo ima n čvorova koji nisu listovi, tada ono ima $n+1$ list. Dakle, svako pravilno binarno stablo ima neparan broj čvorova.

d) Podsetimo se da puno binarno stablo ima listove samo na poslednjem nivou, a svaki čvor koji nije list ima tačno dvoje dece. Ako je visina punog binarnog stabla jednaka h, tada je broj njegovih čvorova jednak $2^{h+1}-1$.

7. Ako je binarno stablo predstavljeno na standardan način pokazivačima na čvorove, napišite i analizirajte algoritme za preorder i postorder obilaske čvorova binarnog stabla.

8. Za svaku od sledećih sekvenci ključeva odredite binarno stablo pretrage koje se dobija kada se ti ključevi dodaju početno praznom stablu, redom jedan po jedan u datom redosledu.

a) $1,2,3,4,5,6,7$.

b) $4,2,1,3,6,5,7$.

c) $1,6,7,2,4,3,5$.

9. Za svako od binarnih stabla pretrage dobijenih u prethodnom zadatku odredite binarno stablo pretrage koje se dobija kada se ukloni koren.

10. Pretpostavimo da je dat sortiran niz od n ključeva $x_1 \leq x_2 \leq \cdots \leq x_n$ i, radi jednostavnosti, pretpostavimo da je n jednak stepenu broja 2.

a) Koja je minimalna visina binarnog stabla sa n čvorova?
b) Napišite algoritam za konstruisanje binarnog stabla pretrage od datih ključeva koji se redom dodaju u njega tako da visina dobijenog stabla bude minimalna.
c) Koje je vreme izvršavanja tog algoritma?

11. U standardnoj reprezentaciji binarnog stabla pretrage pomoću pokazivača, pretpostavimo da svaki njegov čvor dodatno sadrži pokazivač na svog roditelja. Napišite algoritam `bst-successor(t,v)` koji kao rezultat vraća čvor sa ključem koji je prvi sledeći iza ključa čvora v u sortiranom rastućem redosledu svih ključeva binarnog stabla pretrage T. Koje je vreme izvršavanja tog algoritma? (*Savet:* Sledbenik čvora v je minimalni čvor u desnom podstablu čvora v, ako ono postoji; u suprotnom slučaju, to je najniži (prvi) predak čvora v čije je levo dete takođe predak čvora v.)

12. Pod istim pretpostavkama kao u prethodnom zadatku, napišite efikasan algoritam `bst-predecessor(t,v)` koji kao rezultat vraća čvor sa ključem koji je prvi prethodni ispred ključa čvora v u sortiranom rastućem redosledu svih ključeva binarnog stabla pretrage T. Koje je vreme izvršavanja tog algoritma?

13. Napišite efikasan algoritam koji određuje k-ti najmanji ključ u binarnom stablu pretrage. Na primer, ako je dato binarno stablo pretrage sa n čvorova, za $k = 1$ treba odrediti najmanji ključ, za $k = n$ treba odrediti najveći ključ, a za $k = n/2$ treba odrediti ključ u sredini sortirane liste ključeva u rastućem redosledu.

14. Za svaku od sledećih sekvenci ključeva odredite binarni hip koji se dobija kada se ti ključevi dodaju početno praznom hipu, redom jedan po jedan u datom redosledu.

a) $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.
b) $3, 1, 4, 1, 5, 9, 2, 6, 5, 4$.
c) $2, 7, 1, 8, 2, 8, 1, 8, 2, 8$.

15. Za svako od binarnih hipova dobijenih u prethodnom zadatku odredite binarni hip koji se dobija kada se ukloni njegov minimalni čvor.

16. Pretpostavljajući da je hip predstavljen pomoću niza, napišite algoritam koji nalazi *najveći* ključ u min-hipu. Koje je vreme izvršavanja tog algoritma? (*Savet:* Najpre pokažite da najveći ključ u min-hipu mora biti jedan od listova.)

17. Pretpostavimo da se za sortiranje niza neuređenih brojeva $a_1, a_2, \ldots, a_n$ u rastućem redosledu koristi algoritam u kome se redom izvršavaju sledeće dve faze:

1. Faza dodavanja: Redom dodati čvorove sa ključevima $a_1, a_2, \ldots, a_n$ u početno prazno binarno stablo pretrage.

2. Faza listanja: Izlistati ključeve čvorova završnog stabla.

a) U kojem redosledu treba izlistati čvorove u drugoj fazi ovog algoritma — preorder, inorder ili postorder?

b) Koliko je vreme izvršavanja za fazu listanja?

c) Ako je ulazni niz brojeva već sortiran, koliko je ukupno vreme izvršavanja ovog algoritma?

d) Ako se koristi binarni hip umesto binarnog stabla pretrage, koliko je vreme izvršavanja za fazu dodavanja ovog algoritma?

e) Kako treba modifikovati fazu listanja u slučaju binarnog hipa?

f) Koliko je vreme izvršavanja za ovu modifikovanu fazu listanja?

18. Napišite algoritam `heap-sort(a,n)` kojim se niz a od n neuređenih brojeva sortira u rastućem redosledu korišćenjem strukture podataka binarnog hipa. Koliko je vreme izvršavanja ovog načina sortiranja niza?

19. Implementirajte operacije nad redom sa prioritetima o kojima smo govorili na strani 176. Konkretno, ukoliko je red sa prioritetima Q predstavljen pomoću binarnog min-hipa u obliku niza, na pseudo jeziku napišite algoritme:

- `pq-insert(Q,x)`
- `pq-deletemin(Q)`
- `pq-make(Q,a)`
- `pq-changekey(Q,x,p)`
- `pq-delete(Q,x)`

GLAVA

7

Grafovi

U raznim primenama često treba predstaviti neke objekte i uzajamne odnose koji postoje između tih objekata. Grafovi su prirodan model za takvu reprezentaciju. Jedan primer je auto-karta na kojoj se gradovi predstavljaju tačkama, a međusobni odnos povezanosti tih gradova putevima predstavlja se linijama koje povezuju odgovarajuće tačke. Drugi primer je lokalna mreža računara u kojoj se pojedini računari predstavljaju čvorovima, a direktni fizički linkovi između računara predstavljaju se linijama koje povezuju odgovarajuće čvorove. U stvari, skoro u svim oblastima svakodnevnog života nailazimo na grafove, doduše češće pod imenom mreže: transportne mreže, komunikacione mreže, informacione mreže, socijalne mreže i tako dalje. Nije daleko od istine konstatacija da što se više radi sa grafovima, to se sve više grafovi prepoznaju u svemu.

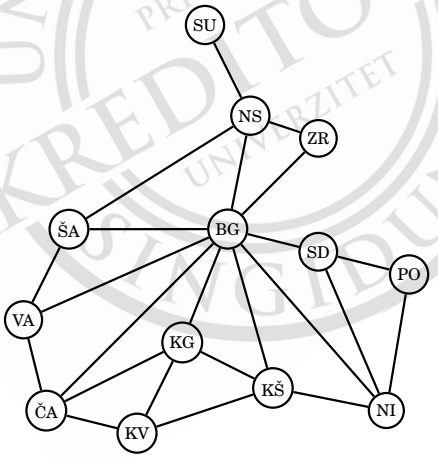
Zbog svoje široke praktične primene, grafovi su sveprisutni model podataka u računarstvu, a algoritmi sa grafovima su osnovni gradivni elementi složenijih algoritama. Iako grafovi matematički predstavljaju obične binarne relacije, oni se skoro uvek vizuelno predstavljaju pomoću skupa tačaka povezanih linijama ili strelicama. U tom smislu je grafovski model podataka uopštenje strukture podataka stabla koju smo izučavali u poglavlju 6.

U ovom poglavlju se najpre definišu osnovni pojmovi koji se odnose na grafove. Zatim se govori o nekoliko elementarnih grafovskih algoritama koji su značajni u primenama grafova. O složenijim grafovskim algoritmima će biti više reči na kraju ovog poglavlja, kao i u celom narednom poglavlju.

7.1 Osnovni pojmovi i definicije

Povezane strukture podataka o kojima smo do sada govorili, kao što su liste i korenska stabla, sastoje se od čvorova koji se povezuju na tačno određen način. Na primer, binarna stabla pretrage imaju oblik koji je diktiran načinom na koji se obavljaju operacije pretrage, dodavanja i uklanjanja čvorova u tim stablima. Međutim, priroda mnogih važnih problema zahteva opštiju strukturu koja se sastoji od kolekcije čvorova povezanih na proizvoljan način. Te strukture se nazivaju grafovi.

Graf se dakle sastoji od skupa čvorova i skupa grana koje povezuju neke od tih čvorova. Grafovi se neformalno često predstavljaju dijagramima u kojima se čvorovi prikazuju kružićima, a grane se prikazuju linijama (ili strelicama) koje spajaju odgovarajuće kružiće. Na slici 7.1 je prikazan graf koji predstavlja veće gradove u Srbiji i njihovu povezanost glavnijim putevima.



SLIKA 7.1: Graf gradova u Srbiji povezanih glavnijim putevima.

Formalno, graf G se sastoji od skupa čvorova V i skupa grana E, pri čemu svaka grana iz E *spaja* dva čvora iz V. Graf se označava zapisom $G = (V, E)$ da bi se ukazalo da su V skup čvorova i E skup grana u grafu G. Svaka grana e u E je dakle dvoelementni podskup čvorova iz V koje spaja, odnosno $e = \{u, v\}$ za neke čvorove u i v iz V. Za čvorove u i v koje spaja grana e kažemo da su *susedni* čvorovi ili da je grana e *incidentna* čvorovima u i v. Na primer, graf G gradova u Srbiji povezanih putevima

na slici 7.1 sastoji se od skupa V sa 13 čvorova:

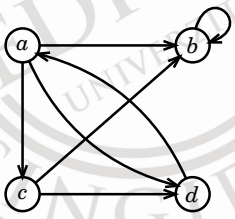
$$V = \{\text{BG}, \text{NS}, \text{NI}, \text{KG}, \text{SD}, \text{PO}, \text{VA}, \text{ČA}, \text{KV}, \text{KŠ}, \text{ŠA}, \text{ZR}, \text{SU}\},$$

kao i skupa E sa 23 grane:

$$E = \big\{\{\text{BG}, \text{NS}\}, \{\text{BG}, \text{VA}\}, \{\text{BG}, \text{NI}\}, \{\text{KG}, \text{ČA}\}, \ldots\big\}.$$

Broj grana nekog čvora v koje ga spajaju sa drugim čvorovima grafa je *stepen* čvora v. Čvorovi na drugom kraju tih incidentnih grana čvora v nazivaju se *susedi* čvora v. Drugačije rečeno, stepen nekog čvora je broj njegovih suseda. Na primer, u grafu na slici 7.1 stepen čvora BG je 9, a stepen čvora SU je 1.

U stvari, prethodne definicije se odnose na *neusmerene* grafove kao što je to graf na slici 7.1. Kod takvih grafova grane označavaju simetričan odnos između čvorova na njihovim krajeva. Sa druge strane, čvorovi često stoje u asimetričnom odnosu i za takve slučajeve se koriste *usmereni* grafovi. Na dijagramima usmerenih grafova se grane koje spajaju dva čvora prikazuju strelicama od jednog do drugog čvora da bi se ukazalo na njihov asimetričan odnos. Na slici 7.2 je prikazan primer jednog usmerenog grafa.



SLIKA 7.2: Usmeren graf.

Formalno, usmeren graf $G = (V, E)$ se sastoji od skupa čvorova V i skupa *usmerenih* grana E. Svaka usmerena grana e u E je *uređen par* dva čvora iz V koje spaja, odnosno $e = (u, v)$ za neke čvorove u i v iz V. Drugim rečima, uloga čvorova u i v nije zamenjiva u usmerenoj grani e, pa se čvor u zove *početak* i čvor v se zove *kraj* grane e. Isto tako, u slučaju grane $e = (u, v)$, čvor v jeste sused čvora u, ali u nije sused čvora v. Na primer, graf G na slici 7.2 sastoji se od skupa V sa 4 čvora:

$$V = \{a, b, c, d\},$$

kao i skupa E sa 7 usmerenih grana:

$$E = \big\{(a, b), (a, c), (a, d), (b, b), (c, b), (c, d)(d, a)\big\}.$$

Obratite pažnju na to da usmerena grana u usmerenom grafu može imati početak i kraj u istom čvoru. Takva jedna grana, koja se naziva *petlja*, u grafu na slici 7.2 je grana kod čvora b. Sa druge strane, petlje u neusmerenim grafovima nisu dozvoljene, jer grana u neusmerenom grafu mora biti podskup od dva različita čvora.

Kod usmerenih grafova se razlikuje *izlazni stepen* i *ulazni stepen* nekog čvora v: izlazni stepen je broj usmerenih grana koje „izlaze" iz čvora v i ulazni stepen je broj usmerenih grana koje „ulaze" u čvor v. *Stepen* čvora u usmerenom grafu je zbir njegovog izlaznog i ulaznog stepena, pri čemu se petlja broji dvaput.

Nije teško primetiti da se svaki neusmeren graf može smatrati ekvivalentnim jednom usmerenom grafu koji ima isti skup čvorova i čiji se skup grana dobija tako što se svaka grana $\{u,v\}$ u neusmerenom grafu zameni dvema usmerenim granama (u,v) i (v,u). Teorijski dakle neusmereni grafovi su specijalan slučaj usmerenih grafova, pa je dovoljno posmatrati samo opštije usmerene grafove. Uprkos tome u ovoj knjizi, radi jednostavnosti, uglavnom govorimo o neusmerenim grafovima i zato ćemo pod kraćim izrazom „graf" podrazumevati neusmeren graf ukoliko nije drugačije naglašeno.

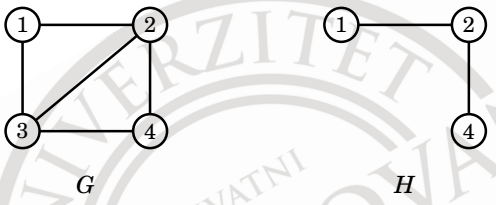
Putevi i povezanost. Jedna od osnovnih operacija u grafu je prolazak kroz niz čvorova povezanih granama. Uzmimo na primer graf kojim se može predstaviti neki veb sajt. U tom grafu čvorovi odgovaraju pojedinačnim veb stranama i postoji grana između čvora u i v ako u ima hiperlink na v. U takvom grafu jedna operacija prolaska kroz čvorove odgovara načinu na koji neki korisnik pregleda veb sajt prateći hiperlinkove.

Prethodna operacija se formalizuje pojmom puta u grafu. *Put* u neusmerenom grafu $G=(V,E)$ je niz P čvorova $v_1, v_2, \ldots, v_{k-1}, v_k$ takvih da je svaki susedan par čvorova v_i, v_{i+1} u tom nizu spojen granom u G. *Dužina* puta P je broj grana na tom putu, odnosno $k-1$. Za put P se često kaže da je to put koji ide *od* čvora v_1 *do* čvora v_k, ili da je to v_1–v_k put. Na slici 7.1, niz čvorova BG, ČA, VA, BG, NS obrazuje put dužine 4 od čvora BG do čvora NS.

Put u grafu je *prost* ako su svi čvorovi na tom putu međusobno različiti. *Ciklus* je put $v_1, v_2, \ldots, v_{k-1}, v_k$ takav da je $k>2$, prvih $k-1$ čvorova su međusobno različiti i $v_1 = v_k$. Drugim rečima, ciklus u grafu je zatvoren prost put ne računajući prvi i poslednji čvor. Graf koji nema nijedan ciklus se naziva *acikličkі* graf.

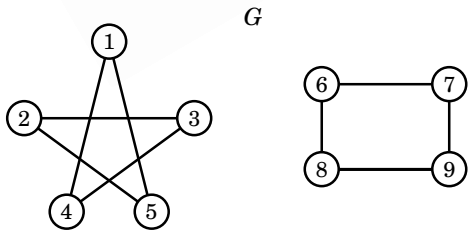
Sve ove definicije se prirodno prenose na usmerene grafove, uz jednu izmenu: u putu ili ciklusu svi parovi susednih čvorova moraju biti povezani usmerenim granama čiji se smer mora poštovati između susednih čvorova u putu ili ciklusu. Na primer, niz čvorova a, d, a, c, b, b na slici 7.2 obrazuje put dužine 5 od čvora a do čvora b.

Za graf $H = (V', E')$ kažemo da je *podgraf* grafa $G = (V, E)$ ukoliko je $V' \subseteq V$ i $E' \subseteq E$. Drugim rečima, graf H je podgraf grafa G ako su čvorovi grafa H podskup čvorova grafa G, a grane grafa H su podskup grana grafa G koje su incidentne čvorovima u H. Na slici 7.3 je prikazan primer grafa i jedan od njegovih podgrafova.



SLIKA 7.3: Graf i jedan od njegovih podgrafova.

Neusmeren graf G je *povezan* ukoliko G ima samo jedan čvor (i nijednu granu) ili za svaki par čvorova u i v u G postoji neki put od u do v. Primetimo da ovo ne znači da u povezanom grafu postoji grana između svaka dva čvora, već samo da postoji put (moguće dugačak) između njih. Primetimo i da ako postoji u-v put u neusmerenom grafu, onda postoji i v-u put u tom grafu koji se sastoji od obrnutog niza čvorova u-v puta. Graf na slici 7.1 je primer jednog povezanog grafa. Sa druge strane, graf na slici 7.4 nije povezan graf, jer ne postoji nijedan put, recimo, između čvorova 1 i 6.



SLIKA 7.4: Nepovezan graf.

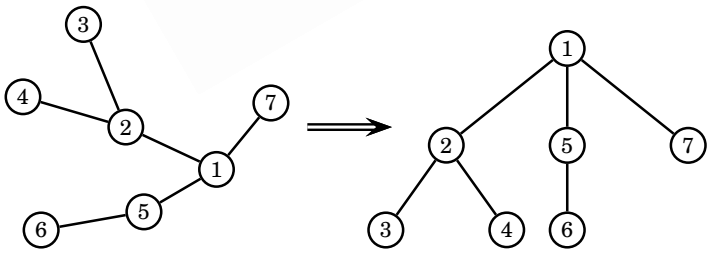
Nepovezan graf na slici 7.4 se očigledno sastoji od dva povezana dela. Ti delovi se nazivaju *povezane komponente* grafa (ili kraće samo *komponente*)

koje predstavljaju maksimalne povezane podgrafove.[1] Primetimo da se u neusmerenom grafu svaki čvor nalazi u tačno jednoj od njegovih povezanih komponenti.

Definicija povezanosti usmerenih grafova nije tako prosta kao za neusmerene grafove, jer je moguće da postoji put od čvora u do čvora v, ali ne i obrnut put od čvora v do čvora u. Zbog toga kažemo da je usmeren graf *jako povezan* ukoliko za svaka dva čvora u i v postoji put od u do v i put od v do u. Na primer, usmeren graf na slici 7.2 nije jako povezan, jer ne postoji nijedan put, recimo, od čvora B do čvora A.

Slobodna stabla. Povezan graf koji nema nijedan ciklus naziva se *slobodno stablo*. Slobodna stabla se razlikuju od korenskih stabala o kojima smo govorili u poglavlju 6 jedino po tome što su kod slobodnih stabala svi čvorovi ravnopravni, odnosno slobodna stabla nemaju jedan poseban čvor koji se zove koren.

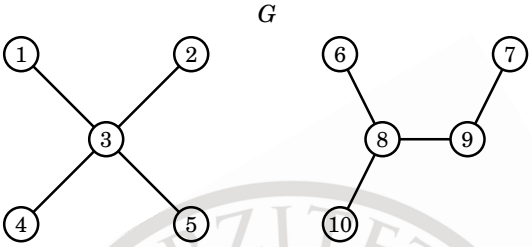
Neko slobodno stablo se lako može transformisati u korensko stablo ukoliko se najpre izabere proizvoljni čvor za koren i zatim se povežu svi čvorovi na drugom kraju svih grana koje idu od tog korena, zatim povežu svi čvorovi svih grana koje idu od prethodnih čvorova (dece korena) i tako dalje. Slikovitiji način za razumevanje ovog postupka je da zamislimo da grane u slobodnom stablu predstavljaju konce kojima su povezani čvorovi i da se to stablo podigne da visi u vazduhu držeći ga za izabrani koren. Tako, na primer, dva crteža na slici 7.5 predstavljaju zapravo isto stablo, tj. isti parovi čvorova su spojeni granama, ali crtež na levoj strani je slobodno stablo i crtež na desnoj strani je to stablo transformisano u korensko stablo sa čvorom 1 kao korenom.



SLIKA 7.5: Transformacija slobodnog stabla u korensko stablo.

[1]Komponenta grafa je maksimalna u smislu da nije pravi podgraf nijednog drugog povezanog podgrafa datog grafa.

U pravom smislu reči stabla su najprostija vrsta povezanih grafova, jer uklanjanjem bilo koje grane iz stabla ono postaje nepovezano. Graf koji se sastoji od više stabala, tačnije graf čije sve povezane komponente predstavljaju slobodna stabla sama za sebe, naziva se *šuma*. Na slici 7.6 je prikazan primer grafa G koji je šuma.



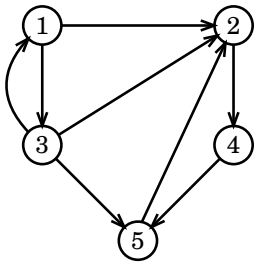
SLIKA 7.6: Šuma.

7.2 Predstavljanje grafova

Da bi se grafovi mogli efikasno koristiti u računaru, moraju se predstaviti u programima na način koji obezbeđuje lako manipulisanje čvorovima i granama grafova. Najprostija struktura podataka koja se koristi za predstavljanje grafova je *matrica susedstva*. Matrica susedstva je običan dvodimenzionalni niz čiji su brojevi redova i kolona jednaki broju čvorova grafa. Obratite pažnju na to da za matricu susedstva grafa nisu bitne oznake čvorova u grafu kojim su oni obeleženi, već je bitan samo njihov ukupan broj.

Matrica susedstva za usmeren graf $G = (V, E)$ od n čvorova koji su numerisani u proizvoljnom redosledu tako da je $V = \{v_1, \ldots, v_n\}$ jeste dakle $n \times n$ matrica a takva da je $a_{i,j} = 1$ ako postoji usmerena grana od čvora v_i do čvora v_j, a u suprotnom slučaju je $a_{i,j} = 0$. Na slici 7.7 je prikazan primer usmerenog grafa i njegova matrica susedstva.

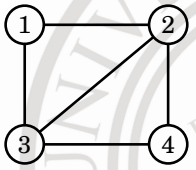
Matrica susedstva za neusmeren graf se obrazuje na sličan način tako što se jedna neusmerena grana između dva čvora predstavlja zapravo dvema usmerenim granama — jednom usmerenom granom od prvog do drugog čvora i drugom usmerenom granom od drugog do prvog čvora. To znači da je matrica susedstva neusmerenog grafa simetrična, odnosno $a_{i,j} = a_{j,i}$ za svako i i j. Pored toga, za matricu susedstva neusmerenog grafa važi da je $a_{i,i} = 0$ za svako i, jer neusmeren graf ne sadrži grane-petlje od nekog



	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	1	0
3	1	1	0	0	1
4	0	0	0	0	1
5	0	1	0	0	0

SLIKA 7.7: Usmeren graf i njegova matrica susedstva.

čvora do istog čvora. Na slici 7.8 je prikazan primer neusmerenog grafa i njegova matrica susedstva.



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

SLIKA 7.8: Neusmeren graf i njegova matrica susedstva.

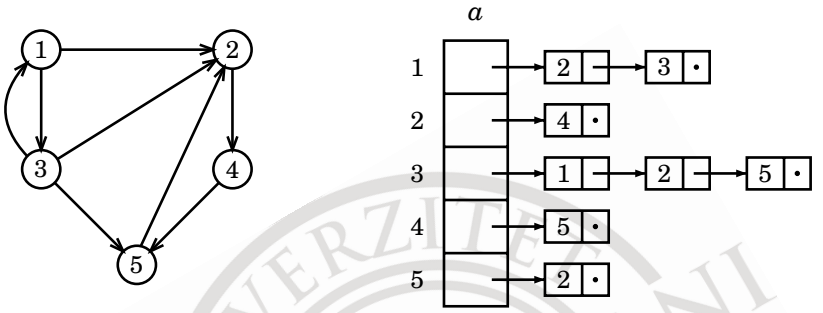
Ako se koristi reprezentacija grafa (usmerenog ili neusmerenog) pomoću matrice susedstva, vreme pristupa bilo kom elementu matrice je konstantno, odnosno to vreme ne zavisi od broja čvorova i grana grafa. Zbog toga je ovakva reprezentacija pogodna u slučajevima kada se u algoritmu često proverava da li neka grana postoji ili ne u grafu.

Glavni nedostatak matrice susedstva za predstavljanje grafova je u tome što matrica susedstva za graf od n čvorova zahteva veličinu memorijskog prostora $\Theta(n^2)$, čak i za grafove sa malim brojem grana. Zbog toga vreme potrebno samo za formiranje cele matrice susedstva grafa iznosi $\Theta(n^2)$, čime se automatski ne mogu dobiti algoritmi brži od $\Theta(n^2)$ za rad sa retkim grafovima sa, recimo, n grana. Graf je *redak* ukoliko je njegov broj grana mali deo od najvećeg mogućeg broja grana. Nije teško proveriti da najveći broj grana za usmeren graf sa n čvorova iznosi n^2, dok je taj broj za neusmeren graf jednak $n(n-1)/2$.

Da bi se rešio ovaj problem za retke grafove, koristi se alternativna reprezentacija grafova pomoću *listi susedstva*. Ova reprezentacija ima nekoliko varijacija, a u najprostijem obliku se graf predstavlja jednim nizom povezanih listi susedstva za svaki čvor grafa. Lista susedstva za čvor v je

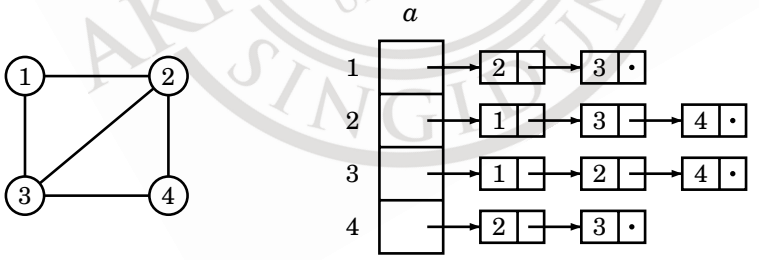
lista svih čvorova koji su susedni čvoru v u grafu, u nekom redosledu.

Usmeren graf $G = (V,E)$ sa skupom čvorova $V = \{v_1,\ldots,v_n\}$ može se tako predstaviti pomoću jednog niza a veličine n, pri čemu element a_i sadrži pokazivač na listu susedstva čvora v_i. Na slici 7.9 je ilustrovana reprezentacija usmerenog grafa nizom listi susedstva.



SLIKA 7.9: Usmeren graf predstavljen listama susedstva.

Neusmeren graf se listama susedstva predstavlja na sličan način tako što se jedna neusmerena grana predstavlja dvema usmerenim granama. Na slici 7.10 je ilustrovana reprezentacija neusmerenog grafa nizom listi susedstva.



SLIKA 7.10: Neusmeren graf predstavljen listama susedstva.

Nije teško proveriti da je za predstavljanje grafa od n čvorova i m grana nizom listi susedstva potreban memorijski prostor veličine $\Theta(n+m)$. Naime, niz kojim se predstavljaju čvorovi grafa očigledno zauzima memorijski prostor veličine $\Theta(n)$. Dužine listi susedstva zavise od čvora do čvora i jednake su broju suseda (stepenu) pojedinih čvorova. Ali primetimo da, ako je n_u broj suseda čvora u, onda je $\sum_{u\in V} n_u = 2m$. Naime, svaka grana $e = \{u,v\}$ se računa tačno dvaput u ovom zbiru: jednom za n_u i jednom za n_v. Pošto je ovaj zbir jednak ukupnom broju puta koliko se

računa svaka grana, taj zbir je $2m$. Drugim rečima, ukupna dužina svih listi susedstva je $2m$, odnosno memorijski prostor za sve liste susedstva je veličine $\Theta(2m) = \Theta(m)$.

Veličina memorijskog prostora $\Theta(n+m)$ za niz listi susedstva je mnogo manja od $\Theta(n^2)$ za matricu susedstva u slučaju retkih grafova. Sa druge strane, treba imati u vidu da potencijalni nedostatak reprezentacije nizom listi susedstva može biti vreme od $\Theta(n)$ koje je potrebno za proveravanje da li postoji određena grana između dva čvora, jer može biti $\Theta(n)$ čvorova u listi susedstva za bilo koji čvor.

7.3 Obilazak grafa

U poglavlju 6 smo upoznali tri načina za obilazak binarnog stabla pomoću kojih se svi čvorovi nekog stabla mogu sistematično posetiti kretanjem duž grana stabla od čvora do čvora. Kod grafova se sličan problem tradicionalno naziva *pretraga* grafova i rešava se primenom dva metoda koji se nazivaju pretraga u širinu i pretraga u dubinu. Značaj ovih postupaka pretrage grafova je u tome što se na njima zasnivaju mnogi algoritmi za rešavanje praktičnih problema koji se prirodno predstavljaju grafovima.

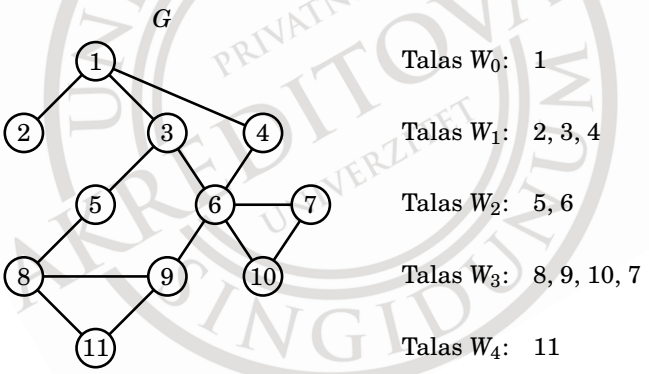
Jedan od takvih osnovnih algoritamskih problema za graf je pitanje povezanosti dva čvora. Za graf G i dva njegova konkretna čvora s i t, problem *s-t povezanosti* sastoji se u nalaženju efikasnog algoritma koji daje odgovor na sledeće pitanje: da li postoji put u grafu G od čvora s do čvora t? Ovo prosto pitanje o tome da li postoji put između dva čvora može se proširiti pitanjem da li postoji *kratak* put između njih. Naime, u mnogim primenama grafova potrebno je pronaći put između dva čvora koji ima najmanje „skokova". Kod transportnih ili komunikacionih mreža, na primer, vrlo je važno znati najkraći put između dve tačke.

Zbog toga se uvodi pojam *rastojanja* između dva čvora u grafu koje se definiše da bude dužina najkraćeg puta između njih. (Ako ne postoji nijedan put između dva čvora, za njihovo rastojanje se obično uzima da je beskonačno.) Primetimo da pojam rastojanja u grafovima ne mora odgovarati geografskom značenju tog pojma — dva čvora mogu biti na kratkom rastojanju u grafu iako predstavljaju tačke koje geografski mogu biti vrlo udaljene.

Pretraga u širinu

Možda najprostiji algoritam za određivanje *s-t* povezanosti u grafu je *pretraga u širinu* grafa (engl. *breadth-first search, BFS*). Taj postupak je dobio svoje ime po tome što se, polazeći od čvora s, ide što šire u grafu posećivanjem čvorova u svim mogućim pravcima u širinu.

Preciznije, za dati graf $G = (V, E)$ i njegov čvor s, polazi se od čvora s i u svakoj fazi pretrage u širinu čvorovi se otkrivaju u „talasima". Prvi talas se sastoji od svih čvorova koji su spojeni granom sa čvorom s. Drugi talas se sastoji od svih novih čvorova koji su spojeni granom sa nekim čvorom iz prvog talasa. Treći talas se sastoji od svih novih čvorova koji su spojeni granom sa nekim čvorom iz drugog talasa. Ovaj postupak se nastavlja na očigledan način sve dok se ne može otkriti nijedan novi čvor. Na primer, na slici 7.11 su prikazani talasi čvorova koji se otkrivaju pretragom u širinu datog grafa ukoliko je početni čvor 1.



SLIKA 7.11: Talasi pretrage u širinu grafa G od početnog čvora 1.

Talasi čvorova $W_0, W_1, W_2, \ldots$ koji se redom dobijaju pretragom u širinu mogu se preciznije definisati na sledeći način:

- Talas W_0 se sastoji od početnog čvora s.
- Talas W_{i+1} za $i \geq 0$ sastoji se od svih čvorava koji ne pripadaju nekom prethodnom talasu i koji su povezani granom sa nekim čvorom u talasu W_i.

Kako je rastojanje između dva čvora najmanji broj grana na nekom putu između njih, primetimo da je talas W_1 skup svih čvorova na rastojanju 1 od čvora s, da je talas W_2 skup svih čvorova na rastojanju 2 od čvora s i

opštije, da je talas W_i skup svih čvorova na rastojanju tačno i od čvora s. Pored toga, neki čvor se ne nalazi ni u jednom talasu pretrage u širinu ako i samo ako ne postoji put do njega od čvora s. Stoga se pretragom u širinu grafa otkrivaju ne samo svi čvorovi koji su dostupni od čvora s u grafu, nego se određuju i najkraći putevi do njih. Stoga, pretragom u širinu se lako rešava problem s-t povezanosti u grafu — dovoljno je tokom pretrage od čvora s proveriti da li je neki otkriven čvor jednak t.

Na osnovu ovih činjenica sledi i da pretraga u širinu neusmerenog grafa G otkriva sve čvorove komponente grafa G koja sadrži početni čvor s. Preciznije, unija svih talasa čvorova konstruisanih tokom pretrage u širinu daje tačno čvorove komponente grafa G koja sadrži čvor s.

Implementacija pretrage u širinu. Metod pretrage u širinu zasniva se na ispitivanju suseda čvorova radi konstruisanja talasa otkrivenih čvorova. Zbog toga je za efikasnu algoritamsku realizaciju pretrage u širinu idealna reprezentacija grafa nizom listi susedstva, jer ove liste upravo sadrže susede pojedinih čvorova. Za algoritam pretrage u širinu zato pretpostavljamo da je ulazni graf G sa n čvorova predstavljen globalnim nizom listi susedstva.

Kada se konstruiše naredni talas, pored ispitivanja suseda čvorova iz prvog prethodnog talasa, potrebno je voditi računa i o tome da neki od tih suseda ne pripada ranijem talasu, odnosno da ti susedi nisu eventualno već otkriveni dosadašnjom pretragom u širinu. Da bi se ovaj uslov mogao lako proveriti u algoritmu, koristi se globalni logički niz `p` dužine n. Svi elementi ovog niza su početno inicijalizovani vrednošću netačno, a element `p[v]` dobija vrednost tačno čim se pretragom otkrije (poseti) čvor v.

U algoritmu za pretragu u širinu datog grafa G posećuju se svi njegovi čvorovi tako što se konstruišu talasi otkrivenih čvorova $W_0, W_1, W_2, \ldots$ na način koji je prethodno opisan. Ovi skupovi čvorova se predstavljaju globalnim nizom listi `W[0]`, `W[1]`, `W[2]` i tako dalje.[2] Radi lakšeg razumevanja algoritma, operacije za rad sa ovim listama u algoritmu nisu pisane onako detaljno kako je to navedeno u odeljku 4.3. Nadamo se da to neće izazvati zabunu i da pažljivi čitaoci mogu lako prepoznati standardne operacije nad listom o kojima je reč.

```
// Ulaz:  čvor s u grafu G predstavljen listama susedstva
// Izlaz: pretraga u širinu grafa G od čvora s
algorithm bfs(s)
```

[2] Primetimo da je, radi jednostavnosti, niz `W` numerisan od 0, a ne od 1 kako smo navikli.

```
W[0] = s;       // talas W_0 početno sadrži čvor s
p[s] = true;    // polazni čvor s je otkriven
i = 0;          // brojač talasa čvorova

while (W[i] != ∅) do   // prethodni talas W_i nije prazan
   W[i+1] = ∅;         // naredni talas W_{i+1} je početno prazan
   for (svaki čvor u u talasu W_i) do
      // Ispitati susede čvora u
      for (svaki čvor v u listi susedstva G za čvor u) do
         if (p[v] == false) then
            p[v] = true;           // čvor v je novootkriven (posećen)
            W[i+1] = W[i+1] + v;   // dodati v talasu W_{i+1}
   i = i + 1;

return;
```

Analiza vremenske složenosti algoritma bfs obuhvaćena je sledećom činjenicom:

LEMA *Za graf G sa n čvorova i m grana koji je predstavljen nizom listi susedstva, vreme izvršavanja algoritma* bfs *je $O(n+m)$. (To vreme je dakle linearno u odnosu na veličinu ulaza algoritma.)*

Dokaz: Očevidno je da u vremenu izvršavanja algoritma bfs dominira vreme za koje se izvršava glavna while petlja. Međutim, tačan broj iteracija ove while petlje nije lako odrediti, pa ćemo postupiti na drugačiji način i analizirati iteracije prve (spoljašnje) for petlje tokom izvršavanja svih iteracija while petlje u algoritmu.

Pre toga, pošto se u svakoj iteraciji while petlje na početku konstruiše prazna lista za aktuelni talas i na kraju uvećava brojač talasa, potrebno je odrediti ukupno vreme potrebno za ta dva koraka. To je lako oceniti ukoliko se primeti da su talasi disjunktni skupovi čvorova tako da ih najviše može biti n. To znači da je u algoritmu potrebno konstruisati najviše n praznih listi i izvršiti najviše n operacija inkrementa brojača, pa je ukupno vreme potrebno za to jednako $O(n)$.

Da bismo sada ocenili vreme izvršavanja spoljašnje for petlje, primetimo da se svaki čvor može naći najviše jedanput u nekom talasu. Stoga se tokom izvršavanja svih iteracija while petlje u algoritmu, telo spoljašnje for petlje izvršava najviše jedanput za svaki čvor u grafa. Pored toga, ako n_u označava broj suseda (stepen) čvora u, onda je broj iteracija unutrašnje for petlje jednak n_u. Kako se dalje jedna iteracija te unutrašnje for petlje očevidno izvršava za konstantno vreme, to je vreme izvršavanja unutrašnje

`for` petlje za svaki čvor u jednako $O(n_u)$. Ukoliko ovo vreme saberemo po svim čvorovima grafa, dobijamo dakle ukupno vreme izvršavanja spoljašnje `for` petlje tokom izvršavanja svih iteracija `while` petlje:

$$\sum_{u \in V} O(n_u) = O\left(\sum_{u \in V} n_u\right) = O(2m) = O(m).$$

Najzad, ako ovom vremenu dodamo prethodno izračunato vreme za konstruisanje listi i uvećavanje brojača tokom izvršavanja svih iteracija `while` petlje, dobijamo traženo vreme izvršavanja algoritma `bfs`: $O(n) + O(m) = O(n+m)$. ■

Stablo pretrage u širinu. Jedno važno svojstvo pretrage u širinu grafa je da se na vrlo prirodan način može konstruisati stablo T sa korenom koji je početni čvor s i sa onim čvorovima iz grafa koji su dostupni od čvora s. Stablo T se zove *stablo pretrage u širinu* i postupno se konstruiše kako se otkrivaju čvorovi grafa tokom pretrage u širinu na sledeći način:

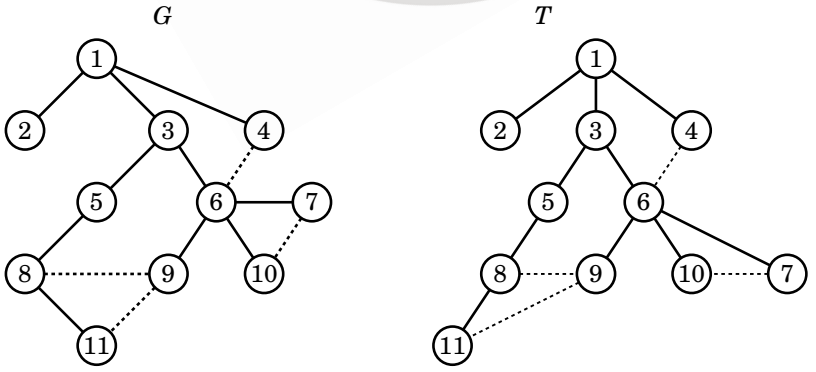
- Početni čvor s je koren stabla T.
- Pretpostavimo da smo konstruisali delimično stablo T koje pokriva čvorove u talasima $W_0, W_1, \ldots, W_i$ za $i \geq 0$ i da su svi čvorovi iz W_i listovi tog stabla. Za svaki otkriven čvor v u talasu W_{i+1}, posmatrajmo trenutak njegovog otkrivanja tokom pretrage. To se događa kada se ispituje neki čvor u iz prethodnog talasa W_i i utvrdi se da je v njegov sused koji nije prethodno otkriven. U tom trenutku se čvor v i grana koja spaja u i v dodaju stablu T, odnosno čvor u postaje roditelj čvora v u stablu T.

Primetimo da je ovaj postupak konstruisanja stabla pretrage u širinu dobro definisan, jer se u svakom koraku dodaju samo novi čvorovi iz narednog talasa, a uz to se dodaju i samo pojedine grane koje ih spajaju sa čvorovima iz prethodnog talasa. Odatle sledi da su novododati čvorovi listovi uvećanog stabla i da novododate grane samo kompletiraju put do njih ne formirajući neki ciklus u stablu.

Prethodni algoritam `bfs` za pretragu u širinu grafa G može se lako proširiti kako bi se, usput, konstruisalo stablo pretrage u širinu T. Naredni prilagođeni algoritam `bfs` sadrži neophodne izmene napisane bez navođenja detalja za operacije dodavanja čvorova i grana strukturi podataka stabla.

```
// Ulaz:  čvor s u grafu G predstavljen listama susedstva
// Izlaz: pretraga u širinu G od s i stablo pretrage u širinu T
algorithm bfs(s)

   W[0] = s;       // talas W_0 početno sadrži čvor s
   p[s] = true;    // polazni čvor s je otkriven
   i = 0;          // brojač talasa čvorova
   T = s;          // koren stabla T je čvor s

   while (W[i] != ∅) do   // prethodni talas W_i nije prazan
      W[i+1] = ∅;         // naredni talas W_{i+1} je početno prazan
      for (svaki čvor u u talasu W_i) do
         // Ispitati susede čvora u
         for (svaki čvor v u listi susedstva G za čvor u) do
            if (p[v] == false) then
               p[v] = true;           // čvor v je novootkriven (posećen)
               W[i+1] = W[i+1] + v;   // dodati v talasu W_{i+1}
               T = T + v + {u,v};     // dodati v i granu {u,v} stablu T
      i = i + 1;

   return;
```

Na slici 7.12 je prikazano stablo pretrage u širinu T koje se dobija ukoliko se pretragom u širinu datog grafa G otkrivaju čvorovi u talasima istim redom kao na slici 7.11. Isprekidanim linijama u grafu G na slici 7.12 istaknute su grane koje ne dovode do otkrivanja novih čvorova tokom pretrage. U stablu T su grane stabla prikazane punim linijama, dok su isprekidanim linijama naznačene grane grafa G koje ne pripadaju stablu T.



SLIKA 7.12: Stablo T koje se dobija pretragom u širinu datog grafa G od čvora 1.

Na slici 7.12 možemo primetiti da sve grane grafa koje ne pripadaju stablu pretrage u širinu (isprekidane linije) povezuju čvorove koji se nalaze ili u istom talasu ili u susednim talasima. Ovo nije slučajno i predstavlja opšte svojstvo stabla pretrage u širinu nekog grafa.

LEMA *Neka je T stablo pretrage u širinu grafa G i neka su u i v čvorovi u T koji pripadaju talasima W_i i W_j, tim redom. Ako neka grana grafa G spaja čvorove u i v, onda se i i j razlikuju najviše za jedan.*

Dokaz: Dokaz izvodimo svođenjem na kontradikciju i zato pretpostavimo da se i i j razlikuju za više od jedan. Radi određenosti, neka je W_i talas ranije otkrivenih čvorova od W_j i $i < j - 1$. U proširenom algoritmu bfs posmatrajmo trenutak kada se ispituju susedi čvora x koji pripada talasu W_i. Pošto je čvor y jedan od suseda čvora x u grafu G, y će biti dodat talasu W_{i+1} ukoliko taj čvor nije ranije otkriven ili y pripada nekom prethodnom talasu ukoliko je taj čvor već otkriven. To znači da mora biti $j = i + 1$ ili $j < i$, što u oba slučaja protivreči pretpostavci da je $i < j - 1$. ■

Pretraga u dubinu

Drugi sistematičan način za obilazak svih čvorova grafa je *pretraga u dubinu* grafa (engl. *depth-first search, DFS*) koja je uopštenje postupka obilaska čvorova stabla u preorder redosledu. Ime ovog metoda potiče od toga što se ide sve dublje u grafu od početnog čvora, duž nekog puta sve dok se taj put više ne može nastaviti, bilo zato što poslednji čvor nema grana koje vode od njega ili zato što sve grane od njega vode do već posećenih čvorova.

Pretraga u dubinu grafa podseća na postupak koji se prirodno primenjuje kada se traži izlaz iz nekog lavirinta. Ukoliko čvorove i grane grafa zamislimo kao lavirint međusobno povezanih prolaza kroz koje treba pronaći izlaz od početnog čvora s, onda je prirodno probati prvu granu koja vodi od s. Ako nas to vodi do nekog čvora v, onda bismo dalje sledili prvu granu koja vodi od v. Ovaj postupak bismo zatim slično nastavili od čvora u koji smo došli od čvora v, sve dok možemo i ne dođemo u „čorsokak", odnosno do čvora čije smo sve susede već probali. Onda bismo se vratili putem kojim smo došli do prvog čvora čije sve susede još nismo probali i zatim bismo odatle ponovili isti postupak.

Primetimo da se od svakog čvora ponavlja ista aktivnost: slediti prvu granu koja vodi do nekog susednog čvora koji nije već posećen. Zato se pretraga u dubinu može najlakše opisati u rekurzivnom obliku u kojem

ova aktivnost odgovara rekurzivnom pozivu, ali se evidencija o tome koji su čvorovi bili posećeni mora voditi na globalnom nivou.

Implementacija rekurzivnog algoritma za pretragu u dubinu grafa zasniva se na istim strukturama podataka koje se koriste i u algoritmu za pretragu u širinu. Pretpostavljamo dakle da je ulazni graf G sa n čvorova globalno predstavljen nizom listi susedstva. Pored toga, globalni logički niz p dužine n, čiji su svi elementi početno inicijalizovani vrednošću netačno, služi za vođenje računa o tome koji čvorovi su već posećeni tokom pretrage.

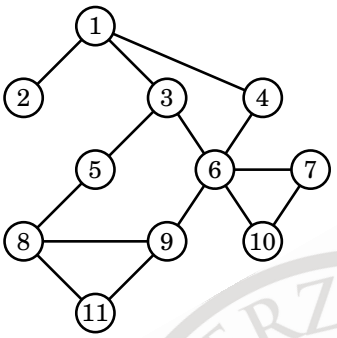
```
// Ulaz:  čvor u u grafu G predstavljen listama susedstva
// Izlaz: pretraga u dubinu grafa G od čvora u
algorithm dfs(u)

   p[u] = true;   // čvor u je posećen

   // Ispitati susede čvora u
   for (svaki čvor v u listi susedstva G za čvor u) do
      if (p[v] == false) then
         dfs(v);   // rekurzivna pretraga u dubinu od čvora v

   return;
```

Primetimo da za algoritam `dfs`, zbog njegove rekurzivne prirode, nije bitan početni čvor, već ulazni čvor može biti bilo koji čvor grafa. Na primer, problem *s-t* povezanosti može se jednostavno rešiti pozivom `dfs(s)`. Pored toga, algoritam `dfs` ne proizvodi neki „opipljiv" rezultat, nego samo propisuje određen način obilaska svih čvorova grafa. (Ovo zapažanje se odnosi i na algoritam `bfs` iz prethodnog odeljka — konstruisani talasi otkrivenih čvorova su samo sredstvo za sistematičan obilazak svih čvorova grafa.) Naravno, pravi rezultat zavisi od konkretnog problema čije se rešenje zasniva na opštem algoritmu pretrage grafa.

Svaki poziv algoritma `dfs` odvija se samo za neki neposećen čvor. Tako, za neki neposećen ulazni čvor u, u algoritmu se najpre registruje da je u posećen, a zatim se ispituju svi susedi čvora u proveravanjem svake grane koja ide od u. Ako se pronađe neki još uvek neposećen čvor v susedan čvoru u, u algoritmu se odmah posećuje čvor v rekurzivnim pozivom istog algoritma za čvor v. Pošto se potpuno provere sve grane koje idu od čvora u, poseta čvoru u se smatra završenom i algoritam završava rad.

Na slici 7.13 je ilustrovano izvršavanje algoritma `dfs` na jednom primeru grafa tako što je uvlačenjem naznačena struktura rekurzivnih poziva. Obratite pažnju na to da rezultat pretrage u dubinu nije jednoznačan

i da zavisi od redosleda kojim se posećuju susedi nekog čvora u `for` petlji algoritma `dfs`.




```
dfs(1)
  dfs(2)
  dfs(3)
    dfs(6)
      dfs(10)
        dfs(7)
      dfs(4)
      dfs(9)
        dfs(8)
          dfs(5)
          dfs(11)
```

SLIKA 7.13: Izvršavanje algoritma `dfs` od čvora 1 datog grafa.

Pretraga u dubinu i pretraga u širinu grafa imaju neke sličnosti i neke razlike. Jedna sličnost ogleda se u činjenici da se na oba načina na kraju obiđe tačno ona povezana komponenta grafa koja sadrži početni čvor. Pored toga, kao što ćemo u nastavku pokazati, pretraga u dubinu ima slične performanse kao pretraga u širinu i pretragom u dubinu se na prirodan način može konstruisati stablo posećenih čvorova slično kao kod pretrage u širinu.

Sa druge strane, iako se primenom oba načina naposletku obiđe isti skup čvorova one komponente grafa koja sadrži početni čvor, pretragom u dubinu se dobija potpuno drugačiji redosled obilaska čvorova. Naime, pretragom u dubinu se obično probijaju vrlo dugački putevi od početnog čvora, odnosno poziv algoritma `dfs(u)` generiše lanac rekurzivnih poziva, potencijalno vrlo dugačak, pre nego što se uradi bilo šta mnogo toga drugog. Ovaj rekurzivni lanac poziva se prekida tek kad se od ulaznog čvora u dođe do čvora v odakle se lanac više ne može nastaviti, bilo zato što čvor v nema grana koje vode od njega ili zato što sve grane od njega vode do već posećenih čvorova. U tom trenutku se rekurzija „odmotava" i vraća se na prvi čvor od tačke prekida odakle se mogu ispitati alternativni putevi.

Analiza pretrage u dubinu. Da bismo analizirali algoritam `dfs`, pretpostavimo da ulazni graf $G = (V, E)$ ima n čvorova i m grana. Iako je `dfs` rekurzivni algoritam, njegovo vreme izvršavanja nećemo odrediti na uobičajen način koji smo primenjivali kod analize rekurzivnih algoritama. Nije

naime jasno kako to vreme možemo izraziti u obliku rekurentne jednačine čijim rešavanjem možemo dobiti funkciju vremena izvršavanja. Umesto toga, postupićemo tako da saberemo vremena izvršavanja svih poziva algoritma `dfs`, ne računajući tu vreme za rekurzivne pozive pojedinih poziva.

Da bismo ocenili vreme izvršavanja jednog poziva `dfs(u)`, izuzimajući vreme za njegove rekurzivne pozive, primetimo da ono zavisi od broja čvorova koji su susedi čvora u. Ako je broj suseda (stepen) čvora u jednak n_u, onda se telo `for` petlje u algoritmu `dfs` izvršava n_u puta. Pošto se u telu te petlje ne računa vreme za izvršavanje rekurzivnog poziva `dfs(v)`, može se zaključiti da se to telo izvršava za konstantno vreme. Prema tome, ukupno vreme za izvršavanje poziva `dfs(u)`, ne računajući njegove rekurzivne pozive, jednako je $O(1+n_u)$. Ovde je jedinica dodata broju suseda n_u kako bi se u obzir uzeo slučaj kada je $n_u = 0$, jer se onda poziv `dfs(u)` ipak izvršava za neko konstantno vreme $O(1)$.

Da bismo dalje ocenili ukupan broj poziva algoritma `dfs`, primetimo da se svaki poziv tog algoritma izvršava najviše jedanput za svaki čvor u grafu. Naime, čim se pozove `dfs(u)` za neki čvor u, odmah se taj čvor obeležava da je posećen, a algoritam `dfs` se nikad ponovo ne poziva za već posećen čvor. Prema tome, ukupno vreme izvršavanja za sve pozive algoritma `dfs` od nekog početnog čvora grafa G je ograničeno zbirom vremena pojedinačnih poziva za sve čvorove grafa G:

$$\sum_{u\in V} O(1+n_u) = O\left(\sum_{u\in V}(1+n_u)\right) = O\left(\sum_{u\in V} 1 + \sum_{u\in V} n_u\right) = O(n+2m) = O(n+m).$$

Drugima rečima, algoritam za pretragu u dubinu grafa G izvršava se za vreme $O(n+m)$, odnosno za linearno vreme u odnosu na zbir brojeva čvorova i grana grafa.

Stablo pretrage u dubinu. Svaka pretraga u dubinu grafa na prirodan način određuje jedno povezujuće stablo komponente grafa koja sadrži početni čvor pretrage s. Naime, pretragom u dubinu se svaki čvor te komponente posećuje samo jednom, pa ti čvorovi i grane grafa kroz koje se prolazi tokom pretrage u dubinu formiraju stablo T. Korensko stablo T se naziva *stablo pretrage u dubinu* i postupno se konstruiše kako se posećuju čvorovi grafa tokom pretrage u dubinu na sledeći način:

- Početni čvor s je koren stabla T.
- Pretpostavimo da se tokom pretrage u dubinu izvršava poziv `dfs(u)` za neki čvor u. Ukoliko se za vreme tog izvršavanja *direktno* rekurzivno poziva `dfs(v)` za neki čvor v, onda se čvor v i grana koja spaja

u i v dodaju stablu T. Drugim rečima, čvor u postaje roditelj čvora v u stablu T, jer je čvor u „odgovoran" za posećivanje čvora v. Tako se u stablu T čvorovi koji su deca čvora u nalaze, sleva na desno, u redosledu po kojem se algoritam `dfs` direktno poziva za te čvorove tokom izvršavanja poziva `dfs(u)`.

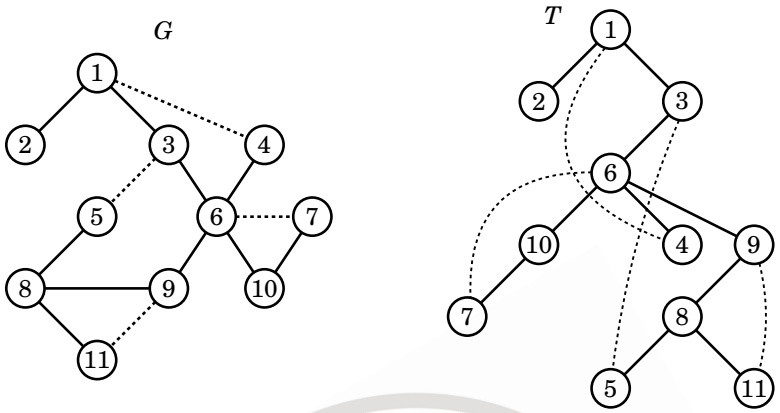
Slično proširenom algoritmu za pretragu u širinu grafa, osnovni algoritam `dfs` za pretragu u dubinu grafa G može se lako proširiti kako bi se usput konstruisalo stabla pretrage u dubinu. Ukoliko je to stablo početno prazno, ono se u narednom prilagođenom algoritmu `dfs` postupno proširuje dodavanjem čvorova i grana.

```
// Ulaz:  čvor u u grafu G predstavljen listama susedstva
// Izlaz: pretraga u dubinu G od u i stablo pretrage u dubinu T
algorithm dfs(u)

   p[u] = true;   // čvor u je posećen
   T = T + u;     // dodati čvor u stablu T

   // Ispitati susede čvora u
   for (svaki čvor v u listi susedstva G za čvor u) do
      if (p[v] == false) then
         dfs(v);   // rekurzivna pretraga u dubinu od čvora v
         T = T + {u,v};   // dodati granu {u,v} stablu T

   return;
```

Na slici 7.14 je prikazano stablo pretrage u dubinu koje se konstruiše pretragom u dubinu koja počinje od čvora 1 grafa G na slici 7.13. Pređene grane grafa su predstavljene punim linijama, dok su isprekidanim linijama predstavljene grane grafa koje nisu korišćene tokom pretrage.

Grane grafa G se mogu klasifikovati prema tome u kom odnosu stoje sa granama stabla pretrage u dubinu T koje se konstruiše tokom pretrage u dubinu. Grana $e = \{u, v\}$ u grafu G koja prouzrokuje to da se u pozivu `dfs(u)` direktno rekurzivno poziva `dfs(v)` naziva se *grana stabla*. Pune linije na slici 7.14 predstavljaju grane stabla.

Primetimo da isprekidane grane na slici 7.14 spajaju dva čvora koja stoje u odnosu predak-potomak u stablu pretrage u dubinu. Isprekidana grana je grana $e = \{u, v\}$ u grafu G takva da ni `dfs(u)` ni `dfs(v)` ne pozivaju jedno drugo direktno, već indirektno. Pošto je isprekidana grana „prečica" za put nagore od potomka do pretka u stablu T, ona se naziva *povratna grana*. Jedno korisno zapažanje u vezi sa ovim jeste da je ovo potpuna klasifikacija grana neusmerenog grafa.



SLIKA 7.14: Stablo T dobijeno pretragom u dubinu datog grafa G.

LEMA *Neka je T stablo pretrage u dubinu povezanog grafa G i neka su u i v čvorovi u T. Ako je $e = \{u, v\}$ grana grafa G koja nije grana stabla T, onda je e povratna grana.*

Dokaz: Primetimo najpre da za dati rekurzivni poziv `dfs(u)`, svi čvorovi koji su posećeni između početka i završetka ovog poziva predstavljaju potomke čvora u u stablu T.

Pretpostavimo sada da je $e = \{u, v\}$ grana grafa G koja nije grana stabla T. Radi određenosti, pretpostavimo još da se pretragom u dubinu dolazi do čvora u pre čvora v, odnosno da je čvor v je u trenutku poziva `dfs(u)` bio neposećen. Kada se čvor v ispituje u pozivu `dfs(u)`, grana $e = \{u, v\}$ se neće dodati stablu T, jer e po pretpostavci nije grana stabla. Ali jedini razlog zašto grana e ne bi bila dodata stablu T je taj što je njen drugi kraj v ranije već posećen. Kako čvor v nije bio posećen pre početka poziva `dfs(u)`, to znači da je čvor v posećen između početka i kraja poziva `dfs(u)`. Na osnovu zapažanja na početku dokaza onda sledi da je čvor v potomak čvora u, odnosno da je e povratna grana. ■

Skup povezanih komponenti grafa

Do sada smo govorili o povezanoj komponenti grafa koja sadrži određen čvor s. Kako se u neusmerenom grafu svaki čvor nalazi u tačno jednoj od njegovih povezanih komponenti, postavlja se pitanje u kom odnosu stoje ove komponente? Taj odnos je izražen na sledeći način:

LEMA *Povezane komponente bilo koja dva čvora s i t u grafu su ili identične ili disjunktne.*

Dokaz: Tvrđenje leme je inuitivno očigledno kada se graf nacrta, jer se onda odmah prepoznaju povezani delovi od kojih se graf sastoji. To nije teško i formalno pokazati, jer treba samo iskoristiti činjenicu da su komponente grafa njegovi povezani delovi, odnosno da postoji put između bilo koja dva čvora u jednoj komponenti.

Pre svega, ukoliko postoji put između čvorova s i t u grafu, onda su povezane komponente koje sadrže s i t isti skup. Naime, svaki čvor v u komponenti koja sadrži s dostupan je i od čvora t — put između t i v sastoji se od dve deonice: deonice između t i s i deonice između s i v. Pošto ova argumentacija važi i kada čvorovi s i t zamene uloge, neki čvor se nalazi u komponenti koja sadrži s ako i samo ako se taj čvor nalazi u komponenti koja sadrži t.

Sa druge strane, ukoliko ne postoji put između čvorova s i t u grafu, onda ne postoji nijedan čvor v koji se nalazi u povezanim komponentama koje sadrže s i t. To je zato što ukoliko bi takav čvor v postojao, onda bi putevi od s do v i od v do t obrazovali i put između s i t. Prema tome, ukoliko ne postoji put između čvorova s i t, onda su njihove povezane komponente disjunktne. ■

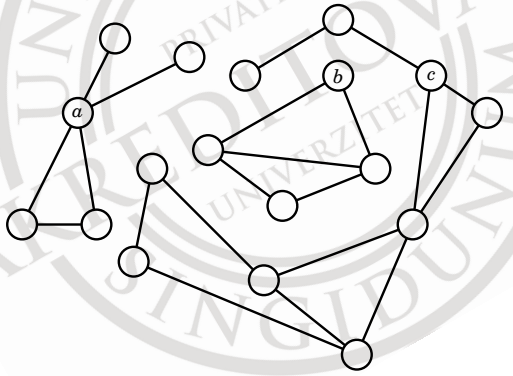
Ovaj vrlo strukturirani međusobni odnos povezanih komponenti grafa može se iskoristiti za njihovo efikasno nalaženje. Prirodni algoritam je da se najpre izabere proizvoljni čvor s i da se zatim pretragom u širinu ili dubinu grafa od čvora s proizvede povezana komponenta koja sadrži taj početni čvor. Isti postupak se zatim ponavlja od proizvoljnog čvora t koji nije bio posećen prethodnom pretragom od čvora s, ukoliko se takav čvor t uopšte može naći. Opravdanje za ovaj pristup leži u prethodnoj lemi: povezane komponente čvorova s i t su međusobno disjunktne. Ako posle završetka druge pretrage od t još uvek postoji neki čvor v koji nije bio posećen prethodnim pretragama, isti postupak se ponavlja i od čvora v, i tako dalje sve dok se ne posete svi čvorovi grafa.

Nije teško proveriti da je vreme izvršavanja ovog algoritma za nalaženje povezanih komponenti grafa od n čvorova i m grana jednako $O(n+m)$. U ovom obliku smo ranije izrazili vremena izvršavanja algoritama za pretrage u širinu i dubinu grafa, ali to vreme je zapravo manje jer se u tim algoritmima koriste samo čvorovi i grane povezane komponente koja sadrži početni čvor pretrage. Mada se pretraga grafa u prethodnom algoritmu za nalaženje svih povezanih komponenti grafa može ponavljati više puta, za

svaku granu ili čvor obavlja se aktivnost konstantnog vremena ipak samo u iteraciji u kojoj se konstruiše povezana komponete kojoj oni pripadaju. Zbog toga je vreme izvršavanja prethodnog algoritma za nalaženje svih povezanih komponenti grafa ukupno jednako $O(n+m)$.

Primer: jedan problem velike TV mreže

Mnogi praktični problemi koji se predstavljaju grafovima mogu se efikasno rešiti prethodnim algoritmom za nalaženje svih povezanih komponenti grafa. Jedan primer takvog problema dobijamo ukoliko zamislimo da je grafom opisana velika TV mreža neke kompanije. U tom grafu svaki čvor predstavlja TV stanicu i svaka grana predstavlja dvosmerni komunikacioni kanal između dve stanice kojim su one direktno zemaljski povezane i mogu da međusobno primaju signal jedna od druge. Na slici 7.15 je prikazan primer jedne TV mreže u obliku neusmerenog grafa.



SLIKA 7.15: TV mreža predstavljena grafom TV stanica.

Ukoliko TV mreža uživo prenosi neki veliki svetski događaj (na primer, svetsko fudbalsko prvenstvo), direktni prenosi se obavljaju preko satalita jer je TV mreža geografski vrlo rasprostranjena. Satelit može da prenosi signal do bilo koje TV stanice, a kada neka TV stanica primi signal od satelita, ona ga može dalje zemaljski emitovati do susednih TV stanica. Na primer, ako stanice a, b i c na slici 7.15 istovremeno dobijaju signal od satelita i difuzno ga reemituju, sve ostale stanice u mreži će dobiti signal direktnog prenosa.

Zemaljsko emitovanje signala je mnogo jeftinije od satelitskog prenosa. Zato je potrebno odrediti *minimalan* broj TV stanica koje moraju da dobiju

signal od satelita kako bi i sve ostale TV stanice u mreži mogle primiti isti signal. Jasno je da se odgovor na pitanje koliki je ovaj minimalni broj svodi na nalaženje broja povezanih komponenti grafa G koji predstavlja TV mrežu. Preciznije rečeno, grafovski problem čije se efikasno rešenje traži je da se svakom čvoru grafa G pridruži broj takav da svi čvorovi u jednoj povezanoj komponenti grafa G dobiju isti redni broj te komponente.

Rešenje ovog problema je dato algoritmom `connected-components` u nastavku i po strukturi je vrlo slično osnovnom algoritmu za nalaženje svih povezanih komponenti grafa G sa n čvorova i m grana. U algoritmu `connected-components` se za brojeve pridružene čvorovima koristi globalni niz c veličine n. Ovi brojevi komponenti se čvorovima dodeljuju na osnovu globalnog brojača k kojim se prebrojavaju povezane komponente grafa G. Rezultat algoritma su dakle elementi niza c takvi da za svaki čvor v u grafu vrednost c_v predstavlja redni broj komponente čvora v. Za otkrivanje jedne komponente grafa se, radi određenosti, koristi pretraga u dubinu grafa. Ona je realizovana algoritmom `dfs-cc` koji je prilagođen tako da se svakom posećenom čvoru pridružuje redni broj njegove komponente.

```
// Ulaz:  graf G predstavljen listama susedstva
// Izlaz: niz c takav da je c_v broj komponente čvora v
algorithm connected-components(G)

   // Svi čvorovi su početno neposećeni
   for (svaki čvor v u nizu listi susedstva G) do
      p[v] = false;
      c[v] = 0;

   k = 0;   // brojač povezanih komponenti grafa G

   // Ispitati sve čvorove grafa G
   for (svaki čvor v u nizu listi susedstva G) do
      if (p[v] == false) then
         k = k + 1;   // čvor v je u novoj komponenti grafa G
         dfs-cc(v);   // nova pretraga u dubinu od čvora v

   return c;
```

```
// Ulaz:  čvor u u grafu G predstavljen listama susedstva
// Izlaz: broj c_u dodeljen svim čvorovima komponente čvora u
algorithm dfs-cc(u)

   p[u] = true;   // čvor u je posećen
   c[u] = k;      // čvor u pripada aktuelnoj komponenti
```

```
// Ispitati susede čvora u
for (svaki čvor v u listi susedstva G za čvor u) do
   if (p[v] == false) then
      dfs-cc(v);   // rekurzivna pretraga od čvora v

return;
```

Vreme izvršavanja algoritma connected-components asimptotski je očigledno jednako vremenu izvršavanja osnovnog algoritma za nalaženje broja povezanih komponenti grafa, odnosno to vreme je $O(n+m)$.

7.4 Obilazak usmerenog grafa

Podsetimo se da su grane u usmerenim grafovima usmerene od jednog do drugog čvora: grana (u,v) ima smer od čvora u do čvora v. Ovaj asimetrični odnos između čvorova proizvodi kvalitativno različit efekat na strukturu usmerenih grafova. Posmatrajmo na primer Internet iz grafovske perspektive kao jedan veliki, komplikovan graf čiji čvorovi predstavljaju pojedine veb strane, a grane predstavljaju međusobne hiperlinkove. Pregledanje Interneta se onda svodi na praćenje niza grana u ovom usmerenom grafu, pri čemu je smer grana od suštinske važnosti jer obično nije moguće „unazad" pratiti hiperlinkove koji vode to date strane.

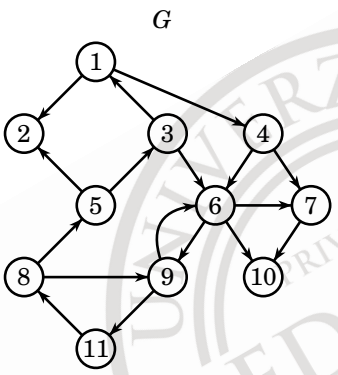
Važnost smera grana odražava se dakle na bitno drugačiji pojam povezanosti u usmerenim grafovima nego u neusmerenim grafovima. U usmerenom grafu je moguće da od čvora s postoji put do čvora t, ali ne i put od t do s. To se onda prenosi na složeniji pojam povezanih komponenti usmerenog grafa i zato se posmatra svojstvo jake povezanosti usmerenih grafova. Jaka povezanost usmerenih grafova otprilike odgovara običnoj povezanosti neusmerenih grafova: usmeren graf je *jako povezan* ako za svaka dva njegova čvora postoji put od jednog do drugog čvora, ali i obrnut put od drugog do prvog čvora.

Skoro svi drugi osnovni pojmovi i algoritmi za neusmerene grafove ipak se vrlo prirodno mogu prilagoditi za usmerene grafove. Ovo specifično važi i za algoritme za pretragu u dubinu i širinu, koji se uz neznatne izmene mogu primeniti za usmerene grafove.

Pretraga usmerenih grafova se konceptualno ne razlikuje od pretrage neusmerenih grafova, jer se jedino menja interpretacija pojma „susedni čvor". Kod usmerenog grafa, čvor v je susedan čvoru u ako postoji usmerena grana (u,v). Ali čvor u nije automatski susedan čvoru v, osim ako ne

postoji i grana (v,u). Ako se uzme u obzir ova promena interpretacije relacije susedstva, osnovni algoritmi `bfs` i `dfs` mogu se bez izmene primeniti i za usmerene grafove. Pored toga, na identičan način se može pokazati da je njihovo vreme izvršavanja za usmeren graf sa n čvorova i m grana takođe jednako $O(n+m)$.

U nastavku ovog poglavlja se zato usredsređujemo samo na pretragu u dubinu usmerenog grafa — pretraga u širinu se može slično analizirati. Na primer, na slici 7.16 ilustrovano je izvršavanje osnovnog algoritma `dfs` za jedan usmeren graf.

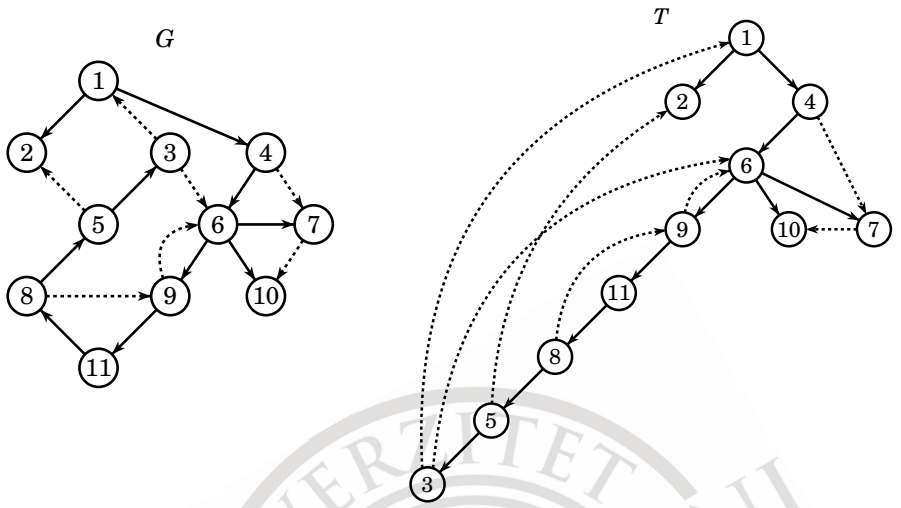



```
dfs(1)
 dfs(2)
 dfs(4)
  dfs(6)
   dfs(9)
    dfs(11)
     dfs(8)
      dfs(5)
       dfs(3)
   dfs(10)
   dfs(7)
```

SLIKA 7.16: Izvršavanje algoritma `dfs` od čvora 1 datog usmerenog grafa.

Na potpuno isti način kao kod neusmerenih grafova može se konstruisati stablo pretrage u dubinu za usmerene grafove: ako se tokom izvršavanja `dfs(u)` direktno poziva `dfs(v)`, stablu se dodaje čvor v i grana stabla koja v povezuje sa u tako da čvor v bude dete čvora u u stablu. Drugim rečima, stablo pretrage u dubinu T koje se indukuje pretragom u dubinu grafa G zapravo je stablo rekurzivnih poziva algoritma `dfs` koje se dobija od početnog poziva tog algoritma za neki čvor grafa G. Na slici 7.17 je prikazano stablo dobijeno pretragom u dubinu koja odgovara primeru na slici 7.16.

Mada su algoritmi pretrage u dubinu usmerenih i neusmerenih grafova konceptualno vrlo slični, njihovo izvršavanje proizvodi prilično različit efekat. Prvo, pretraga u dubinu može proizvesti šumu stabala čak i kada je usmeren graf povezan. Na primer, da smo pretragu u dubinu na slici 7.16 počeli od čvora 7, dobili bismo šumu od dva stabla ili više njih. Drugo, u slučaju neusmerenog grafa pokazali smo da grane grafa koje nisu grane stabla pretrage u dubinu obavezno povezuju neki čvor sa jednim od njego-



SLIKA 7.17: Stablo T dobijeno pretragom u dubinu datog usmerenog grafa G na slici 7.16.

vih predaka u stablu. Drugim rečima, grane neusmerenog grafa koje nisu grane stabla uvek su povratne grane. Sa druge strane, grane usmerenog grafa G u odnosu na njegovo stablo pretrage u dubinu T mogu biti jednog od sledeća tri tipa:

1. *Direktna grana*. To je usmerena grana (u,v) u G koja nije grana stabla T takva da je čvor v pravi potomak čvora u, ali nije dete čvora u u T. Na primer, na slici 7.17 grana $(4,7)$ je direktna grana.

2. *Povratna grana*. To je usmerena grana (u,v) u G koja nije grana stabla T takva da je čvor v predak čvora u u T ($u = v$ je dozvoljeno). Na primer, na slici 7.17 grana $(3,6)$ je povratna grana. Obratite pažnju na to da se petlje, koje su dozvoljene u usmerenih grafovima, smatraju povratnim granama.

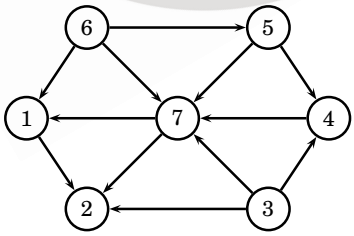
3. *Poprečna grana*. To je usmerena grana (u,v) u G koja nije grana stabla T takva da čvor v nije ni predak ni potomak čvora u u T. Drugačije rečeno, poprečna grana je preostali tip usmerenih grana u G koje nisu grane stabla T. Na primer, na slici 7.17 grana $(7,10)$ je poprečna grana. Primetimo da poprečne grane mogu povezivati čvorove različitih stabala u šumi. Zato u opštem slučaju za određenu poprečnu granu (u,v) ne mora biti tačno da čvorovi u i v imaju zajedničkog pretka.

Pored toga, na slici 7.17 se uočava da poprečna grana $(7,10)$ ima smer zdesna na levo. To nije slučajnost i važi u opštem slučaju: svaka poprečna grana ima smer zdesna na levo, čak i u šumi stabala pretrage u dubinu.

Da bismo to pokazali, neka je (u,v) poprečna grana i posmatrajmo poziv `dfs(u)`. Očevidno je čvor v potomak čvora u u stablu pretrage u dubinu ako i samo ako je čvor v dodat u stablo između vremena početka izvršavanja `dfs(u)` i vremena završetka izvršavanja `dfs(u)`. U tom vremenskom intervalu, do trenutka kada se tokom izvršavanja `dfs(u)` proverava grana (u,v), čvor v je morao biti već posećen, jer (u,v) nije grana stabla. Prema tome, čvor v je morao biti posećen ranije u toku izvršavanja `dfs(u)` ili pre poziva `dfs(u)`. Ali prvi slučaj je nemoguć, jer bi u suprotnom slučaju čvor v bio potomak čvora u. Dakle, v je bio dodat u stablo pretrage u dubinu pre čvora u, a pošto se čvorovi dodaju u redosledu sleva na desno, v mora biti levo od u.

Usmereni aciklički grafovi

Ako neusmeren graf nema nijedan ciklus, onda ima vrlo prostu strukturu šume: sve njegove povezane komponente su stabla. Ali za usmeren graf je moguće da nema nijedan (usmeren) ciklus i da ima složenu strukturu. Na primer, usmeren graf bez ciklusa sa n čvorova može imati veliki broj grana reda $\Theta(n^2)$, dok stablo (neusmeren graf bez ciklusa) sa n čvorova ima „samo" $n-1$ granu. Usmeren graf bez ciklusa se standardno naziva *usmeren aciklički graf* i jedan primer takvog grafa je prikazan na slici 7.18.

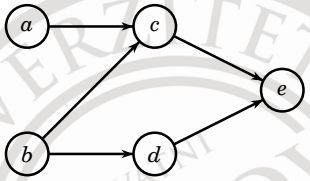


SLIKA 7.18: Usmeren aciklički graf.

Usmeren aciklički graf se u računarstvu često koristi za predstavljanje uslovnog redosleda nekih zadataka po kojem se oni moraju završavati. Na primer, prilikom zidanja neke zgrade, prvi sprat se mora izgraditi pre drugog sprata, ali električni kablovi se mogu postaviti u isto vreme dok se

instaliraju prozori na jednom spratu. Drugi primer je program studiranja na nekom fakultetu, gde su neki predmeti preduslovi za polaganje drugih predmeta. Još jedan primer je kolekcija paralelnih procesa od kojih se sastoji neki računarski program tako da izlaz nekog procesa predstavlja ulaz za neki drugi proces i zato se prvi proces mora završiti pre početka drugog.

Prirodni model ovakvih struktura zavisnosti su usmereni aciklički grafovi: usmerena grana od nekog zadatka a do drugog zadatka b predstavlja uslov da se a mora završiti pre nego što se može početi b. Na slici 7.19 je ilustrovana struktura zavisnosti pet zadataka. U tom primeru, recimo, zadatak c se može obaviti samo kada su zadaci a i b završeni.



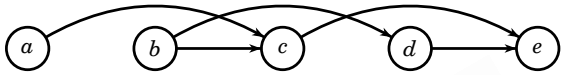
SLIKA 7.19: Usmeren aciklički graf zavisnosti pet zadataka.

U opštem slučaju, pretpostavimo da imamo skup zadataka $\{1,2,\ldots,n\}$ koje treba obaviti i da postoje zavisnosti među njima takve da za izvesne parove zadataka i i j imamo uslov da se i mora završiti pre početka j. Ovakav skup međusobno zavisnih zadataka može se predstaviti usmerenim grafom G sa čvorovima koji predstavljaju pojedine zadatake i granama (i,j) koje predstavljaju uslov da se i mora završiti pre početka j.

Očigledno da bi ova relacija međusobne zavisnosti zadataka uopšte imala smisla, dobijeni usmeren graf G mora biti aciklički, jer ukoliko G sadrži neki ciklus C, onda ne postoji način da se uradi nijedan zadatak u C. Naime, kako svaki zadatak u ciklusu C ima neki drugi zadatak koji se prethodno mora završiti, nijedan zadatak u C se ne može obaviti jer se nijedan zadatak ne može započeti.

Ukoliko usmeren graf zavisnosti zadataka nema cikluse, prirodno je onda tražiti redosled tih zadataka po kojem se oni mogu obaviti tako da sve međusobne zavisnosti zadataka budu poštovane. Takav ispravan redosled zadataka se naziva topološki redosled. Preciznije, *topološki redosled* usmerenog acikličkog grafa G sa n čvorova je neki linerani poredak njegovih čvorova u nizu $v_1, v_2, \ldots, v_n$ tako da za svaku granu (v_i, v_j) važi $i < j$. Slikovito rečeno, sve grane grafa G su usmerene sleva na desno ukoliko se njegovi čvorovi nacrtaju duž horizontalno linije u topološkom redosledu.

Na primer, topološki redosled grafa na slici 7.19 je a, b, c, d, e, što je odmah uočljivo ukoliko se isti graf nacrta kao na slici 7.20. Obratite pažnju na to da taj linearni poredak čvorova nije jedinstven — drugi način za topološki redosled istog grafa je b, d, a, c, e.



SLIKA 7.20: Topološki redosled grafa na slici 7.19.

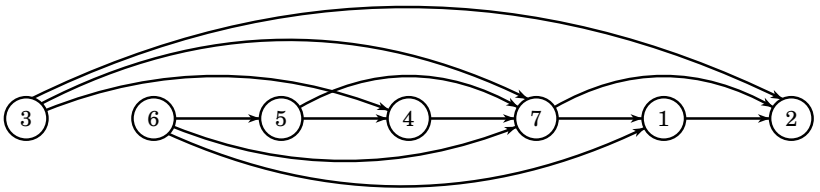
Topološki redosled usmerenog grafa zavisnosti zadataka obezbeđuje dakle redosled po kojem se oni mogu bezbedno uraditi — kada treba početi neki zadatak, svi zadaci koji su njegovi preduslovi sigurno su već završeni. U stvari, posmatrano iz apstraktnijeg ugla, topološki redosled usmerenog grafa predstavlja dokaz da je graf aciklički.

LEMA *Ako usmeren graf G ima topološki redosled, onda G nema nijedan ciklus.*

Dokaz: Pretpostavimo, radi kontradikcije, da G ima jedan topološki redosled svojih čvorova $v_1, v_2, \ldots, v_n$ i da u G postoji ciklus C. Neka je v_j čvor u C sa najmanjim indeksom i neka je v_i čvor u C koji neposredno prethodi čvoru v_j. To znači da je (v_i, v_j) jedna od grana u G. Ali to dovodi do protivrečnosti, jer onda mora biti tačno da je $i < j$ zato što svaka grana za topološki redosled čvorova ide od čvora sa manjim indeksom prema čvoru sa većim indeksom, kao i da je $j < i$ zato što su v_j i v_i čvorovi u C i v_j je čvor u C sa najmanjim indeksom. ■

Prema tome, postojanje topološkog redosleda usmerenog grafa je dovoljan uslov da taj graf nema cikluse. Taj kriterijum može ponekad biti vrlo koristan, jer se vizuelno može lako pokazati da je graf aciklički. Na primer, nije odmah jasno to da graf na slici 7.18 nema cikluse, ali ako se isti graf prikaže u topološkom redosledu kao na slici 7.21, onda je to očigledno pošto sve grane idu sleva na desno.

Nalaženje topološkog redosleda. Teže pitanje za usmeren aciklički graf je obrnuto od onog na koje odgovor daje poslednja lema: da li svaki usmeren aciklički graf ima topološki redosled? Ako je to slučaj, dodatno bi bilo vrlo korisno imati efikasan algoritam za nalaženje jednog topološkog redosleda datog usmerenog acikličkog grafa, jer onda bi se za svaku relaciju zavisnosti nad skupom zadataka bez ciklusa mogao efikasno naći



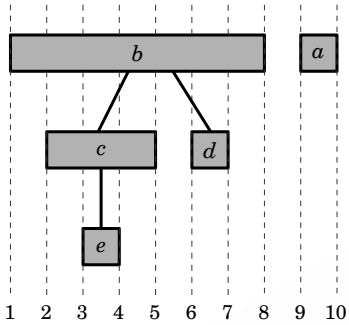
SLIKA 7.21: Topološki redosled grafa na slici 7.18.

redosled po kojem se ti zadaci mogu bezbedno obaviti. Odgovor na prethodno pitanje je potvrdan i, u stvari, to ćemo pokazati na konstruktivan način. Drugim rečima, pokazaćemo jedan efikasan algoritam kojim se za dati usmeren aciklički graf pronalazi njegov topološki redosled.

Algoritam za nalaženje topološkog redosleda usmerenog acikličkog grafa G sličan je algoritmu za nalaženje povezanih komponenti neusmerenog grafa, osim što nije svejedno koja pretraga grafa se koristi nego se mora koristiti pretraga u dubinu. Opšti pristup je dakle da se najpre izabere proizvoljni čvor s u G i da se zatim pretragom u dubinu od čvora s posete svi čvorovi u G koji su dostupni od tog početnog čvora. Isti postupak se zatim ponavlja od proizvoljnog čvora t koji nije bio posećen prethodnom pretragom od čvora s, ukoliko se takav čvor t uopšte može naći. Ako posle završetka druge pretrage od t još uvek postoji neki čvor v koji nije bio posećen prethodnim pretragama, isti postupak se ponavlja i od čvora v, i tako dalje sve dok se ne posete svi čvorovi grafa. Na ovaj način se indukuje šuma stabala pretrage u dubinu sa korenima s, t i tako dalje.

Za razumevanje ovog algoritma je najvažnija činjenica to da jedno stablo pretrage u dubinu u suštini predstavlja stablo rekurzivnih poziva algoritma `dfs` od početnog poziva tog algoritma za neki čvor. Kako se za svaki čvor v početak i završetak poziva `dfs(v)` odigravaju u diskretnim vremenskim trenucima, to se na vremenskoj osi može pratiti izvršavanje celokupnog prethodno opisanog postupka za nalaženje topološkog redosleda. Na primer, na slici 7.22 su na vremenskoj osi prikazani intervali trajanja svih poziva algoritma `dfs` tokom višestruke pretrage u dubinu grafa na slici 7.19 od čvora b i od čvora a. Pravougaoni oblik čvorova na toj slici ukazuje na interval između početka i završetka poziva algoritma `dfs` za odgovarajući čvor.

Podsetimo se da se tokom (višestruke) pretrage u dubinu grafa algoritam `dfs` poziva tačno jedanput za svaki čvor grafa. Zbog toga je vreme završetka poziva `dfs(v)` za svaki čvor v nedvosmisleno određeno. Pretpo-



SLIKA 7.22: Intervali trajanja poziva algoritma `dfs` na vremenskoj osi tokom višestruke pretrage u dubinu grafa na slici 7.19.

stavimo da je tokom (višestruke) pretrage u dubinu grafa G dobijen niz čvorova $v_1, v_2, \ldots, v_n$ u redosledu koji je određen po vremenu završetka poziva algoritma `dfs` za sve čvorove. To znači da se tokom pretrage u dubinu grafa G prvi završio poziv `dfs`(v_1), zatim se završio poziv `dfs`(v_2) i tako dalje, poslednji se završio poziv `dfs`(v_n). Recimo, u primeru na slici 7.22 dobija se redosled čvorova e, c, d, b, a. Ključno zapažanje je da je ovaj redosled zapravo *obrnut* topološki redosled usmerenog acikličkog grafa.

LEMA *Neka je tokom (višestruke) pretrage u dubinu usmerenog acikličkog grafa G dobijen niz čvorova $v_1, v_2, \ldots, v_n$ u redosledu koji je određen po vremenu završetka poziva algoritma* `dfs` *za sve čvorove. Ako je (v_i, v_j) grana grafa G, onda je $i > j$.*

Dokaz: Neka je (v_i, v_j) grana usmerenog acikličkog grafa G. Budući da (višestruka) pretraga u dubinu grafa G indukuje šumu stabala pretrage u dubinu, grana (v_i, v_j) u G se može klasifikovati prema svom tipu kao grana nekog stabla, direktna grana, povratna grana ili poprečna grana. U stvari, pošto je G aciklički graf, (v_i, v_j) ne može biti povratna grana, jer bi se njome očevidno zatvorio jedan put u grafu G koji bi formirao usmeren ciklus.

U slučaju kada je (v_i, v_j) grana stabla ili direktna grana, nije se teško uveriti da je $i > j$. Naime, u tom slučaju je čvor v_j potomak čvora v_i, pa se poziv `dfs`(v_i) svakako završava posle poziva `dfs`(v_j).

Najzad, ako je (v_i, v_j) poprečna grana, onda se mogu razlikovati dva slučaja zavisno od toga da li čvorovi v_i i v_j pripadaju istom stablu ili različitim stablima u šumi pretrage u dubinu. Ipak, u oba slučaja smer grane

(v_i, v_j) zdesna na levo ukazuje da je poziv `dfs`(v_i) započet posle završetka poziva `dfs`(v_j), pa se svakako poziv `dfs`(v_i) i završava posle završetka poziva `dfs`(v_j). A to upravo znači da je $i > j$, što je trebalo pokazati. ■

Da bi se odredio topološki redosled usmerenog acikličkog grafa G, dovoljno je dakle modifikovati osnovni algoritam `dfs` tako da se neporedno pre njegovog završetka čvor koji je njegov parametar dodaje na kraj jedne povezane liste. Na taj način se obrazuje lista čvorova $v_1, v_2, \ldots, v_n$ u obrnutom topološkom redosledu grafa G. Naravno, na kraju postupka nalaženja topološkog redosleda treba još samo obrnuti dobijenu listu. Drugi ekvivalentni pristup, koji smo zapravo primenili u modifikovanom algoritmu `dfs`, sastoji se u tome da se neporedno pre završetka tog algoritma odgovarajući čvor dodaje na početak povezane liste.

```
// Ulaz:  usmeren aciklički graf G predstavljen listama susedstva
// Izlaz: lista čvorova l u topološkom redosledu grafa G
algorithm topological-ordering(G)

   // Svi čvorovi su početno neposećeni
   for (svaki čvor v u nizu listi susedstva G) do
      p[v] = false;

   list-make(l);   // l je početno prazna lista

   // Ispitati sve čvorove grafa G
   for (svaki čvor v u nizu listi susedstva G) do
      if (p[v] == false) then
         dfs-to(v);   // nova pretraga u dubinu od čvora v

   return l;
```

```
// Ulaz:  čvor u u grafu G predstavljen listama susedstva
// Izlaz: čvor u dodat na početak liste l na kraju pretrage u dubinu
algorithm dfs-to(u)

   p[u] = true;   // čvor u je posećen

   // Ispitati susede čvora u
   for (svaki čvor v u listi susedstva G za čvor u) do
      if (p[v] == false) then
         dfs-to(v);      // rekurzivna pretraga od čvora v

   list-insert(u,null,l);  // dodati čvor u na početak liste l

   return;
```

Vreme izvršavanja algoritma `topological-ordering` za topološki redosled usmerenog acikličkog grafa G sa n čvorova i m grana asimptotski je očigledno jednako $O(n+m)$.

Zadaci

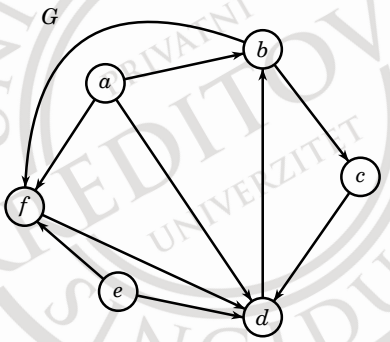
1. Za usmeren graf G na slici 7.23 odgovorite na sledeća pitanja.

a) Koliko čvorova i grana ima graf G?

b) Navedite sve proste puteve od a do b u G.

c) Navedite sve susede čvora b u G.

d) Navedite sve cikluse od čvora b u G.



SLIKA 7.23.

2. Odgovorite na ista pitanja iz prethodnog zadatka za neusmerenu verziju grafa na slici 7.23. Neusmerena verzija se dobija zamenom svih usmerenih grana odgovarajućim neusmerenim granama tako što se strelice samo zamene linijama.

3. Dokažite da ako se granom spoje bilo koja dva čvora u slobodnom stablu T između kojih ne postoji grana, onda se formira tačno jedan ciklus.

4. Dokažite da povezan neusmeren graf G sa n čvorova predstavlja slobodno stablo ako i samo ako G ima $n-1$ granu.

5. Predstavite graf G na slici 7.23 pomoću matrice susedstva i niza listi susedstva.

6. Predstavite neusmerenu verziju grafa G na slici 7.23 pomoću matrice susedstva i niza listi susedstva.

7. Za usmeren graf G predstavljen nizom listi susedstva svojih čvorova potrebno je ponekad za svaki čvor imati i povezanu listu njegovih prethodnika, a ne samo njegovih sledbenika. Preciznije, za svaki čvor v u G treba obrazovati povezanu listu svih čvorova u takvih da je (u,v) usmerena grana u G. Napišite i analizirajte efikasan algoritam kojim se obavlja ovaj zadatak konvertovanja reprezentacije grafa G.

8. Čvor s usmerenog grafa G sa n čvorova naziva se *ponor* ukoliko je ulazni stepen čvora s jednak $n-1$ i njegov izlazni stepen jednak 0. Drugim rečima, za svaki čvor v u G različit od s postoji grana (v,s) i ne postoji grana (s,v). Napišite algoritam vremenske složenosti $O(n)$ koji određuje da li usmeren graf predstavljen matricom susedstva ima jedan čvor koji je ponor.

9. Za usmeren graf G na slici 7.24 odgovorite na sledeća pitanja.

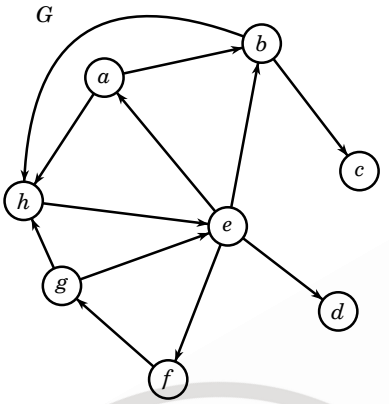
a) Odredite talase čvorova $W_0, W_1, W_2, \ldots$ koji se dobijaju pretragom u širinu grafa G od početnog čvora e.

b) Konstruišite odgovarajuće stablo pretrage u širinu.

10. Odgovorite na ista pitanja iz prethodnog zadatka za neusmerenu verziju grafa G na slici 7.24.

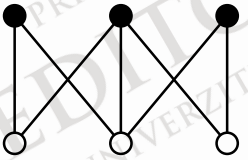
11. Napišite i analizirajte algoritam `bfs` za pretragu u širinu grafa koji koristi jedan red za čekanje umesto više posebnih listi `W[i]` za svaki talas W_i. (*Savet:* U algoritmu se ispituju svi susedi onog čvora koji je prvi u redu za čekanje, a čim se neki čvor otkrije prilikom ispitivanja, dodaje se na kraj reda za čekanje.)

12. Neusmeren graf $G = (V,E)$ je *bipartitan* ukoliko se njegovi čvorovi mogu podeliti u dva disjunktna skupa takva da je svaka grana u G incidentna



SLIKA 7.24.

čvorovima iz ovih skupova. Formalno, $V = X \cup Y$, $X \cap Y = \emptyset$ i $\{u,v\} \in E \Rightarrow u \in X$ i $v \in Y$. Na slici 7.25 je prikazan primer bipartitnog grafa.



SLIKA 7.25.

a) Pokažite da je svako slobodno stablo jedan bipartitan graf.

b) Napišite efikasan algoritam kojim se proverava da li je dati neusmeren graf bipartitan. (*Savet:* Koristite pretragu u širinu.)

13. Za usmeren graf G na slici 7.24, kao i za njegovu neusmerenu verziju, odgovorite na sledeća pitanja.

a) Navedite sve rekurzivne pozive algoritma `dfs` za graf G koji se dobijaju za početni poziv `dfs(a)` i uvlačenjem naznačite njihov odnos. Pretpostavite da ako treba izabrati neki čvor između više njih, onda se od mogućih bira prvi čvor po abecednom redosledu.

b) Konstruišite odgovarajuće stablo (šumu) pretrage u dubinu.

c) Klasifikujte sve grane grafa G u odnosu na konstruisano stablo prema tome da li predstavljaju grane stabla, direktne grane, povratne grane ili poprečne grane.

14. Napišite i analizirajte iterativnu (nerekurzivnu) verziju algoritma `dfs` za pretragu u dubinu grafa od početnog čvora s. (*Savet:* Rekurzivnu strukturu pretrage u dubinu simulirajte stekom na koji se čvorovi smeštaju kako se posećuju, a sa vrha steka se uklanja čvor kada treba ispitati sve njegove susede.)

15. Dokažite da u povezanom neusmerenom grafu G postoji čvor od koga su njegova stabla pretrage u dubinu i širinu jednaka ako i samo ako je G slobodno stablo. (*Savet:* Ova stabla su jednaka ako imaju jednake skupove grana, odnosno redosled u kojem se grane koriste tokom pretrage u dubinu i širinu nije važan.)

16. Napišite algoritam `connected-components` na strani 208 tako da se za otkrivanje jedne komponente grafa koristi pretraga u širinu umesto pretrage u dubinu grafa.

17. Navedite primer usmerenog grafa bez ciklusa sa n čvorova koji ima $\Theta(n^2)$ grana. (*Savet:* Uočite graf $G = (V, E)$ kod koga je $V = \{1, 2, \ldots, n\}$ i $E = \{(i, j) : i < j \text{ za } i, j \in V\}$.)

18. Ako $d(u, v)$ označava dužina najkraćeg puta (rastojanje) između čvorova u i v u grafu $G = (V, E)$, *ekscentričnost* čvora u je:

$$\max_{v \in V} d(u, v).$$

Ekscentričnost čvora u je dakle dužina najdužeg najkraćeg puta koji polazi od u. Napišite efikasan algoritam kojim se izračunava ekcentričnost svih čvorova usmerenog grafa. (*Savet:* Koristite pretragu u širinu.)

19. Neka je G usmeren graf i T jedno njegovo stablo u dubinu.

a) Dokažite da G ima usmeren ciklus ako i samo ako G ima povratnu granu u odnosu na T.

b) Napišite efikasan algoritam kojim se proverava da li je G aciklički graf.

20. Neka je G usmeren aciklički graf.

a) Dokažite da G sadrži čvor bez ulaznih grana.

b) Dokažite indukcijom po broju čvorova grafa da G ima topološki redosled. (*Savet:* U dokazu indukcijskog koraka uočite graf $G - \{v\}$ dobijen uklanjanjem čvora v bez ulaznih grana.)

c) Koristeći indukcijski dokaz napišite drugi algoritam za nalaženje topološkog redosleda grafa G.

21. Prost put u grafu je *Hamiltonov* ako prolazi kroz sve čvorove grafa. Napišite efikasan algoritam kojim se određuje da li dati usmeren aciklički graf sadrži Hamiltonov put. (*Savet:* Posmatrajte topološki redosled datog grafa.)

GLAVA

8

Težinski grafovi

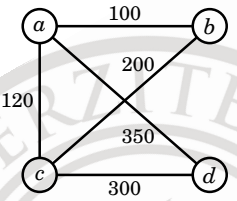
U prethodnom poglavlju smo pokazali da se za nalaženje najkraćeg puta (rastojanja) između dva čvora u povezanom grafu može koristiti pretraga u širinu. Naime, podsetimo se da i-ti talas otkrivenih čvorova tokom pretrage u širinu sadrži čvorove grafa koji se nalaze tačno na rastojanju i od početnog čvora. Pojam rastojanja u ovom slučaju podrazumeva da su sve grane ravnopravne, odnosno da sve imaju istu „težinu" u izračunavanju dužine puteva u grafu. Međutim, u mnogim primenama ovaj pristup nije odgovarajući.

Graf može predstavljati, na primer, neku računarsku mrežu (kao što je Internet) sa računarima kao čvorovima tog grafa i komunikacionim linkovima između pojedinih računara kao granama tog grafa. Ako je cilj naći najbrži način za usmeravanje paketa podataka između neka dva računara u mreži, onda se to svodi na nalaženja najkraćeg puta između dva čvora koji označavaju date računare u grafu. Ali u ovom slučaju nije prirodno da sve grane grafa budu ravnopravne pošto su neke veze u mreži računara obično mnogo brže od drugih — neke grane mogu predstavljati spore telefonske linije, dok druge predstavljaju brze optičke veze.

Da bismo slične probleme modelirali grafovima, u ovom poglavlju proširujemo pojam grafa i posmatramo (usmerene ili neusmerene) grafove čije grane imaju pridruženu numeričku vrednost. Ove vrednosti dodeljene granama nazivaju se *težine* grana, a odgovarajući grafovi *težinski grafovi*.

8.1 Osnovni pojmovi

Težinski graf je običan (usmeren ili neusmeren) graf čije sve grane imaju težinu u obliku numeričke vrednosti. Primetimo da su težine grana neutralni naziv za brojeve koji su pridruženi granama, ali u praktičnim primenama ti brojevi mogu predstavljati razne druge fizičke veličine kao što su dužina, vreme, cena, brzina ili neka druga mera odnosa dva čvora. Primer jednog težinskog grafa je prikazan na slici 8.1.



SLIKA 8.1: Težinski graf.

Koncepti u vazi sa težinskim grafovima se bez izmena odnose kako na neusmerene tako i na usmerene grafove. Zbog toga u ovom poglavlju govorimo samo o neusmerenim težinskim grafovima, a zainteresovanim čitaocima ostavljamo da izložene probleme i rešenja prilagode usmerenim težinskim grafovima.

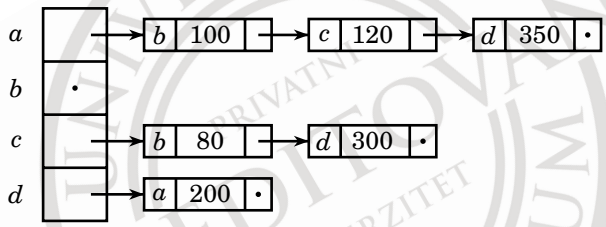
Težinski graf je formalno dakle graf $G = (V, E)$ takav da svaka grana e u E ima dodeljen broj $w(e)$ koji određuje njenu težinu. S obzirom na činjenicu da običan graf možemo smatrati težinskim grafom čije su sve težine grana jednake (na primer, možemo uzeti da su sve težine jednake 1), jasno je da svi osnovni pojmovi za obične grafove imaju smisla i za težinske grafove.

Čak se i dve reprezentacije običnih grafova u računarima, matrica susedstva i niz listi susedstva, mogu lako prilagoditi za predstavljanje težinskih grafova. Podsetimo se da elementi matrice susedstva za običan graf mogu biti brojevi 1 ili 0 čime se određuje da li grana između neka dva čvora postoji ili ne. U slučaju težinskih grafova, postojanje neke grane e i njena težina $w(e)$ zajedno se predstavljaju odgovarajućim elementom matrice susedstva koji je jednak $w(e)$. Ali zato ako neka grana ne postoji u grafu, odgovarajući element matrice susedstva mora biti neka vrednost koja nije moguća za težinu grane. Na slici 8.2 je prikazana matrica susedstva u kojoj se za predstavljanje nepostojeće grane u težinskom grafu na slici 8.1 koristi specijalni simbol ∞.

	a	b	c	d
a	∞	100	120	350
b	∞	∞	∞	∞
c	∞	80	∞	300
d	200	∞	∞	∞

SLIKA 8.2: Matrica susedstva za težinski graf na slici 8.1.

Niz listi susedstva za predstavljajnje običnog grafa može se lako prilagoditi za predstavljajnje težinskog grafa ukoliko se težina neke grane e priključi elementu povezane liste koji određuje drugi kraj grane e. Na slici 8.3 je prikazana reprezentacija težinskog grafa na slici 8.1 pomoću niza listi susedstva.

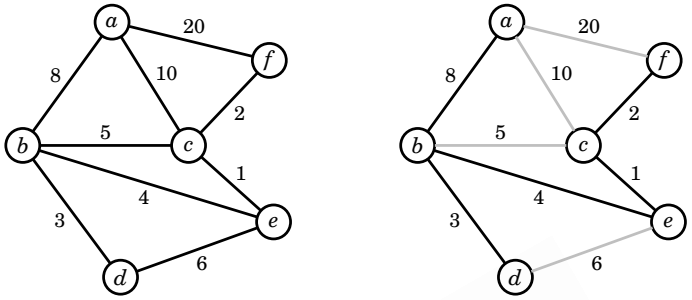


SLIKA 8.3: Niz listi susedstva za težinski graf na slici 8.1.

8.2 Minimalno povezujuće stablo

Da bismo uveli pojam minimalnog povezujućeg stabla nekog težinskog grafa, razmotrimo najpre jedan praktičan primer. Pretpostavimo da neka kompanije ima više predstavništva u različitim gradovima koje treba da poveže u jedinstvenu mrežu. Da bi povezala sva svoja predstavništva u integrisanu mrežu, za kompaniju je važno da odredi najmanju cenu iznajmljivanja telekomunikacionih linija između svojih predstavništava.

Ovaj problem se može predstaviti težinskim grafom sa čvorovima koji označavaju predstavništva kompanije i granama sa numeričkim vrednostima koje odgovaraju cenama zakupljenih prenosnih veza između pojedinih predstavništava. Na slici 8.4(a) je prikazan hipotetički graf koji predstavlja sve moguće zakupljene linije između odgovarajućih predstavništava i njihove cene.



SLIKA 8.4: Težinski graf predstavništava hipotetičke kompanije.

Nije teško doći do zaključka da najekonomičnije rešenje za ovaj problem treba da obuhvati podskup grana T datog grafa koji ispunjava ove uslove:

1. Grane u T moraju povezivati sve čvorove, jer je neophodno da bilo koja dva predstavništva kompanije mogu da međusobno komuniciraju, makar indirektno preko drugih predstavništava.
2. Zbir cena grana u T mora biti najmanji, odnosno ne sme biti striktno veći od zbira cena bilo kog drugog podskupa grana datog grafa koje ispunjavaju prvi uslov.

Primetimo odmah da najekonomičnije rešenje T ne sadrži ciklus, jer bi u suprotnom mogla da se smanji njegova cena uklanjanjem bilo koje grane u ciklusu. Pri tome povezanost između bilo koje dva čvora datog grafa ne bi bila ugrožena. Jedno optimalno rešenje T za graf na slici 8.4(a) prikazano je pod (b) na istoj slici tako što su grane u T istaknute u grafu.

U opštem slučaju, pretpostavimo da je dati težinski graf $G = (V, E)$ povezan i da su sve težine grana pozitivne. Problem o kome je reč u primeru kompanije sa više predstavništava je naći podskup grana $T \subseteq E$ takav da podgraf (V, T) bude povezan i da ukupna težina $\sum_{e \in T} w(e)$ grana u T bude minimalna. Osnovno zapažanje u vezi sa ovim problemom je da svako njegovo rešenje (V, T) mora biti slobodno stablo, odnosno povezan graf bez ciklusa.

LEMA *Ako je* (V, T) *povezan podgraf težinskog grafa* $G = (V, E)$ *čija je ukupna težina grana najmanja, onda je* (V, T) *slobodno stablo.*

Dokaz: Pošto je (V, T) po definiciji povezan graf, treba se još samo uveriti u to da graf (V, T) ne sadrži nijedan ciklus. Ali ako bi graf (V, T) sadržao neki ciklus C, radi kontradikcije uočimo bilo koju granu e tog ciklusa C.

Onda bi graf $(V, T-e)$ i dalje bio povezan, jer svaki put u (V,T) između dva čvora koji koristi e može ići zaobilazno u $(V,T-e)$ preko čvorova u ostatku ciklusa C. Stoga je i graf $(V,T-e)$ jedno rešenje za prethodni problem. Uz to je rešenje $(V,T-e)$ „jeftinije" od rešenja (V,T), jer je ukupna težina grana u grafu $(V,T-e)$ striktno manja za težinu grane e. Ovo pritivreči pretpostavci da je (V,T) „najjeftinije" rešenje navedenog problema, što pokazuje da graf (V,T) ne može imati nijedan ciklus. ■

Podgraf (V,T) običnog (netežinskog) grafa $G=(V,E)$ naziva se *povezujuće stablo* grafa G ukoliko je (V,T) slobodno stablo. Često se radi jednostavnosti sam podskup grana $T \subseteq E$ naziva povezujuće stablo, jer se podrazumeva da ono sadrži skup svih čvorova V datog grafa G. Nije teško proveriti da povezujuće stablo grafa G postoji ako i samo ako je G povezan graf.

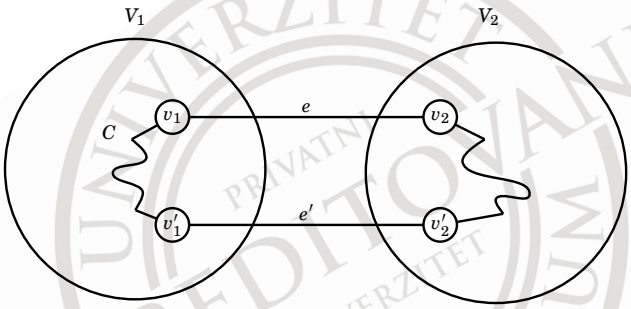
U slučaju povezanog težinskog grafa G, zbir težina grana u nekom povezujućem stablu T tog grafa nazivamo *cenom* stabla T. Problem nalaženja povezujućeg stabla grafa G koje ima najmanju cenu, prema prethodnoj lemi, zapravo je drugim rečima formulisan opšti problem koji smo ilustrovali primerom kompanije sa više predstavništava: za dati težinski graf $G=(V,E)$ naći podskup grana $T \subseteq E$ tako da podgraf (V,T) bude povezan i da ukupna težina grana u T bude minimalna. Zbog toga se rešenje T ovog problema naziva *minimalno povezujuće stablo* (engl. *minimum spanning tree*) grafa G. Dakle, minimalno povezujuće stablo je *stablo* jer je aciklički graf; ono je *povezujuće* jer povezuje sve čvorove datog grafa; i ono je *minimalno* jer ima najmanju cenu među svim povezujućim stablima.

Sem ukoliko G nije vrlo prost graf, G može imati ogroman broj povezujućih stabala čija je struktura potpuno različita. Zbog toga na prvi pogled nije uopšte lako naći povezujuće stablo sa najmanjom cenom među svim mogućim kandidatima. Ipak, otkriveni su mnogi efikasni algoritmi za nalaženje minimalnog povezujućeg stabla datog težinskog grafa. Interesantno je da se svi oni zasnivaju na relativno jednostavnoj činjenici koju zovemo *svojstvo podele grafa*.

LEMA (SVOJSTVO PODELE GRAFA) *Neka je $G=(V,E)$ povezan težinski graf i neka su V_1 i V_2 dva neprazna disjunktna podskupa njegovog skupa čvorova V takva da je $V = V_1 \cup V_2$. Ako je e* jedinstvena *grana najmanje težine u E takva da joj je jedan kraj u V_1 i drugi kraj u V_2, onda svako minimalno povezujuće stablo grafa G sadrži granu e.*

Dokaz: Pretpostavimo da je $e=\{v_1,v_2\}$ *jedinstvena* grana najmanje težine od onih kod kojih je jedan kraj u V_1 i drugi kraj u V_2. Neka je T bilo koje

minimalno povezujuće stablo grafa G. (Primetimo da pošto je G povezan graf, G ima povezujuće stablo, pa zato ima i bar jedno minimalno povezujuće stablo.). Radi kontradikcije pretpostavimo da T ne sadrži granu e. Ako granu e ipak dodamo stablu T, formiraće se tačno jedan ciklus C u stablu T. (Podsetimo se da je ovo jedna od osobina slobodnog stabla). U ovom ciklusu C koji sadrži granu e kao „most" između skupa čvorova V_1 i V_2 mora postojati još jedna grana $e' = \{v_1', v_2'\}$ u T takva da $v_1' \in V_1$ i $v_2' \in V_2$. U suprotnom, ciklus C se ne može zatvoriti u T, odnosno ne postoji način da idući duž ciklusa C dođemo, recimo, od čvora v_1 nazad u čvor v_1 bez ponovnog prelaženja preko grane e. Ovaj argument je ilustrovan na slici 8.5.



SLIKA 8.5: Formiranje ciklusa u minimalnom povezujućem stablu dodavanjem grane e.

Ako se sada ukloni grana e' iz stabla T, prekida se jedini ciklus u T i dobija se novo stablo $T' = T - e'$ koje povezuje sve čvorove grafa G. Naime, svaki put u T koji je koristio granu e' može se zameniti drugim putem u T' koji ide „zaobilazno" između čvorova v_1' i v_2' duž ciklusa C.

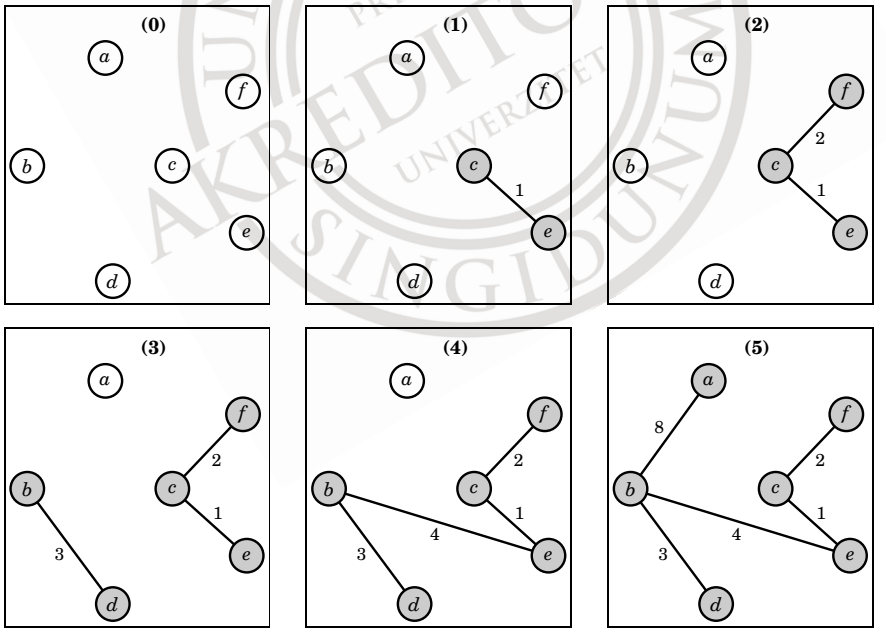
Kako je grana e' jedna od onih koje imaju jedan kraj u V_1 i drugi kraj u V_2, to je po pretpostavci težina grane e striktno manja od težine grane e'. Stoga je i cena stabla T' striktno manja od cene stabla T. Ali to je nemoguće, jer je T minimalno povezujuće stablo grafa G. ■

Praktično svi algoritmi za konstruisanje minimalnog povezujućeg stabla težinskog grafa koriste na neki način ovo svojstvo podele grafa radi postupnog izbora njegovih grana koje pripadaju minimalnom povezujućem stablu. Kruskalov i Primov algoritam su klasični predstavnici ovog pristupa.

Kruskalov algoritam

Radi konstruisanja minimalnog povezujućeg stabla težinskog grafa $G = (V, E)$, u Kruskalovom algoritmu se polazi od šume $(V, \emptyset)$ koja se sastoji od svih čvorova grafa G i nema nijednu granu. Drugim rečima, početna šuma se sastoji od stabala koja čine pojedinačni čvorovi sami za sebe.

U algoritmu se zatim redom ispituju grane grafa u rastućem redosledu njihovih težina. U svakom koraku kako se dolazi do neke grane e u ovom redosledu, proverava se da li e proizvodi ciklus ukoliko se doda granama koje su do tada izabrane za minimalno povezujuće stablo. Tačnije, ako grana e povezuje dva čvora iz dva različita stabla do tada konstruisane šume, onda se e dodaje skupu izabranih grana i ta dva stabla se objedinjuju u jedno stablo. U suprotnom, ako e povezuje dva čvora iz istog stabla do tada konstruisane šume, onda se e odbacuje jer bi proizvela ciklus. Na slici 8.6 je prikazano postupno izvršavanje Kruskalovog algoritma za graf na slici 8.4(a).



SLIKA 8.6: Izvršavanje Kruskalovog algoritma za graf na slici 8.4(a).

Ispravnost Kruskalovog algoritma je neposredna posledica svojstva podele grafa.

LEMA *Kruskalov algoritam konstruiše minimalno povezujuće stablo povezanog težinskog grafa G.*

Dokaz: Pretpostavimo privremeno da su sve težine grana u grafu $G = (V, E)$ međusobno različite, a na kraju ćemo pokazati kako možemo eliminisati ovu pretpostavku.

Razmotrimo bilo koju granu $e = \{u, v\}$ koja je u Kruskalovom algoritmu dodata šumi (stablu) T u k-tom koraku. Neka je A skup svih čvorova povezanih putem sa čvorom u u T i B skup svih čvorova povezanih putem sa čvorom v u T u trenutku neposredno pre dodavanja grane e. Očigledno su A i B disjunktni skupovi, jer bi se u suprotnom slučaju dodavanjem grane e obrazovao ciklus u T.

Neka je $V_1 = A$ i $V_2 = V - A$, pa je $u \in V_1$ i $v \in V_2$. Drugim rečima, V_1 i V_2 su neprazni disjunktni podskupovi čvorova takvi da je $V = V_1 \cup V_2$. Pored toga, nijedna druga grana pre k-tog koraka nije imala jedan kraj u V_1 i drugi kraj u V_2, jer bi u suprotnom takva grana bila već dodata pošto ne obrazuje ciklus u T. Zato je e grana najmanje težine između V_1 i V_2 u grafu G, a uz to je i jedinstvena s tom osobinom jer su sve težine grana različite. Prema svojstvu podele grafa dakle, grana e pripada svakom povezujućem stablu grafa G.

Stoga ukoliko još pokažemo da je završna šuma T koja je rezultat Kruskalovog algoritma zapravo povezujuće stablo grafa G, onda će to automatski značiti da je T ujedno i minimalno povezujuće stablo. Očigledno je da T ne sadrži nijedan ciklus, jer se u Kruskalovom algoritmu eksplicitno odbacuju grane koje obrazuju neki ciklus. Isto tako, završna šuma T ne može se sastojati od dva nepovezana disjunktna stabla T' i T'' (ili više njih), jer je graf G povezan i postoji bar jedna grana između T' i T'', pa će se u algoritmu od tih grana dodati u T ona prva na koju se naiđe u rastućem redosledu težina.

Ostalo je još da pokažemo optimalnost Kruskalovog algoritma u slučaju kada neke grane grafa G imaju jednake težine. U tom slučaju zamislimo da smo od grafa G obrazovali novi težinksi graf G^* tako što smo težinama svih grana sa jednakim težinama u G dodali različite ekstremno male vrednosti kako bi njihove nove težine bile međusobno različite. Takve vrednosti uvek možemo izabrati tako da se ne promeni relativni redosled cena originalnih povezujućih stabala grafa G.

Nije se teško uveriti da je svako minimalno povezujuće stablo T novog grafa G^* ujedno i minimalno povezujuće stablo originalnog grafa G. (Obrnuto, naravno, ne mora biti tačno.) Naime, pošto G i G^* imaju istu

strukturu, ti grafovi imaju ista povezujuća stabla. Ali ako bi stara cena stabla T bila veća od stare cene nekog drugog povezujućeg stabla grafa G, onda za dovoljno male promene težina grana ne bi cena tog drugog stabla mogla da se uveća toliko da stablo T bude bolje sa novim težinama u G^*. To pokazuje da je T minimalno povezujuće stablo i za graf G. Kako su težine grana novog grafa G^* međusobno različite, Kruskalov algoritam će prema prethodnom delu dokaza konstruisati njegovo minimalno povezujuće stablo T koje je optimalno i za originalni graf G. ■

Implementacija Kruskalovog algoritma. U Kruskalovom algoritmu se polazi od šume stabala koja se sastoje samo od pojedinačnih čvorova grafa bez ijedne grane. Ova početna trivijalna stabla u šumi se zatim proširuju dodavanjem grana koje nikad ne proizvode ciklus.

U realizaciji Kruskalovog algoritma potrebno je zato voditi priciznu evideniciju o tome koji disjunktni podskupovi čvorova grafa pripadaju kojim stablima u šumi koja se postupno transformiše u minimalno povezujuće stablo. Pored toga, kako se neka grana $e = \{u, v\}$ ispituje u svakom koraku, potrebno je efikasno odrediti kojim stablima u šumi pripadaju njeni krajevi u i v. Ako su ova stabla različita, onda se grana e dodaje jer ne proizvodi ciklus i dva odgovarajuća skupa čvorova se objedinjuju u jedan skup koji odgovara novom stablu u šumi. Ako krajevi u i v grane e pripadaju istom stablu, onda se grana e preskače jer bi proizvela ciklus u tom stablu.

Za efikasnu realizaciju Kruskalovog algoritma potrebno je dakle tačno ono što obezbeđuje struktura podataka disjunktnih skupova o kojoj smo govorili u odeljku 6.5 na strani 177. Čitaoci se mogu podsetiti da se tom strukturom podataka predstavljaju disjunktni skupovi nekih elemenata. U slučaju Kruskalovog algoritma elementi skupova su čvorovi grafa. Za dati čvor u, operacija `ds-find(u)` kao rezultat daje ime skupa čvorova koji sadrži u. Ova operacija se onda može iskoristiti za utvrđivanje da li čvorovi u i v pripadaju istom skupu tako što se prosto proveri da li je `ds-find(u)` jednako `ds-find(v)`. Isto tako, struktura podataka disjunktnih skupova sadrži operaciju `ds-union(u,v)` kojom se disjunktni skupovi koji sadrže u i v objedinjuju u jedan skup.

Koristeći ova zapažanja, Kruskalov algoritam na pseudo jeziku za nalaženje minimalnog povezujućeg stabla povezanog težinskog grafa G sa n čvorova i m grana je:

```
// Ulaz:  povezan težinski graf G
// Izlaz: minimalno povezujuće stabla T grafa G
algorithm mst-kruskal(G)
```

```
Obrazovati običan niz od grana grafa G sa pridruženim težinama;

Preurediti taj niz u rastućem redosledu težina grana.
Neka je e1,e2,...,em dobijeni niz sortiranih grana;

T = ∅   // šuma T je početno bez grana
// ... ali svi čvorovi su njena stabla
for (svaki čvor v u grafu G) do
   ds-make(v);

for i = 1 to m do
   Neka je ei = {u,v};
   a = ds-find(u);    // ime skupa kome pripada u
   b = ds-find(v);    // ime skupa kome pripada v
   if (a != b) then   // skupovi koji sadrže u i v nisu isti
      T = T + ei;     // dodati granu ei šumi T
      ds-union(a,b);  // objediniti skupove koji sadrže u i v

return T;
```

Vreme izvršavanja algoritma `mst-kruskal` može se odrediti ukoliko se saberu vremena izvršavanja njegovih glavnijih delova:

- Ako je graf G predstavljen matricom susedstva ili nizom listi susedstva, nije se teško uveriti da se običan niz njegovih grana sa pridruženim težinama može lako obrazovati za vreme $O(m)$.
- Za sortiranje tog niza u rastućem redosledu po težinama grana može se iskoristiti neki efikasan algoritam za sortiranje tako da vreme potrebno za preuređivanje niza grana iznosi $O(m \log m)$.
- Vreme izvršavanja prve `for` petlje u kojoj se inicijalizuje n disjunktnih skupova iznosi $O(n)$, jer za inicijalizaciju jednog skupa treba konstantno vreme.
- Telo druge `for` petlje se izvršava m puta, a svaka iteracija se izvršava za vreme $O(\log n)$ ukoliko su operacije sa disjunktnim skupovima realizovane onako kako smo to opisali na strani 177. Prema tome, ukupno vreme izvršavanja druge `for` petlje je $O(m \log n)$.

Kako za povezan graf važi da je $m \geq n-1$, to je $n = O(m)$. Takođe, pošto uvek važi $m \leq n^2$, to je $\log m = O(\log n)$. Uzimajući sve u obzir dakle, ukupno vreme izvršavanja algoritma `mst-kruskal` je:

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

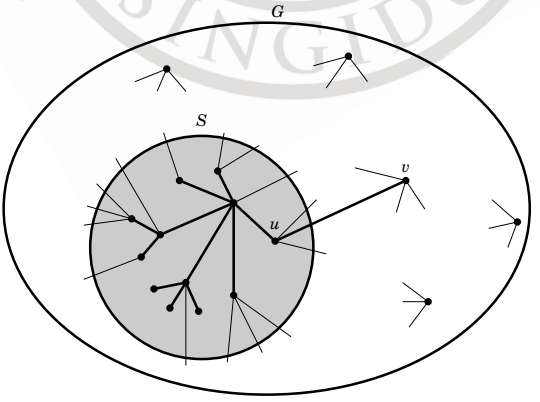
Primov algoritam

U Kruskalovom algoritmu se „pohlepno” biraju grane najmanje težine bez mnogo obzira na to u kakvom odnosu stoje sa prethodno izabranim granama, osim što se pazi da se ne formira ciklus. Na ovaj način se dobija šuma stabala koja se šire prilično nasumično dok se ne stope u jedno završno stablo.

Nasuprot tome, u Primovom algoritmu se dobija minimalno povezujuće stablo tako što se početno najprostije stablo postupno proširuje. To početno stablo je neki proizvoljno izabran čvor s, a u svakom koraku se delimično konstruisanom stablu dodaje novi čvor na drugom kraju grane najmanje težine koja izlazi iz tog stabla. Preciznije, u svakom koraku se obrazuje podskup čvorova $S \subseteq V$ nad kojim je delimično konstruisano povezujuće stablo. Početno je $S = \{s\}$ i zatim se S postupno uvećava za jedan čvor v kojim se minimizuje „cena uvećanja” skupa S:

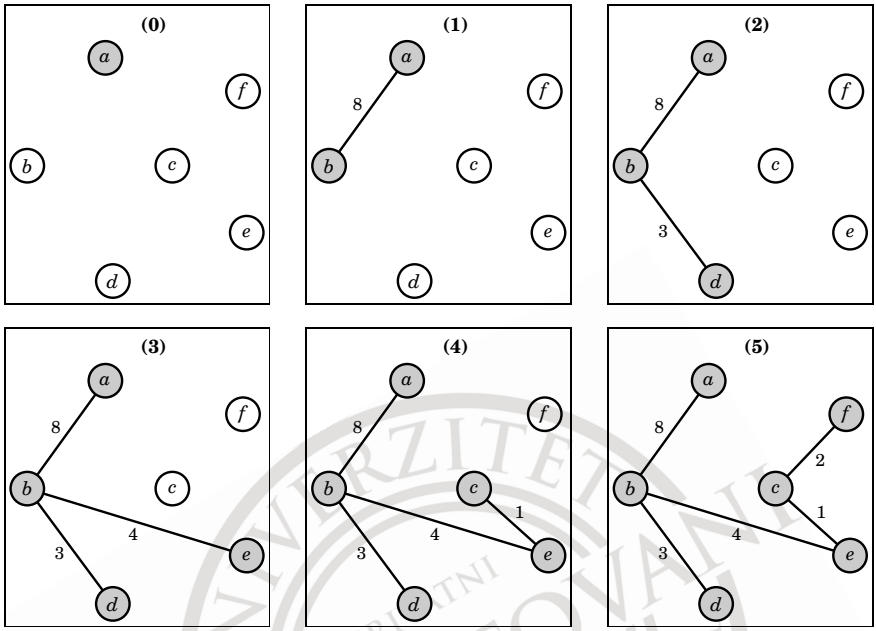
$$\min_{\substack{e=\{u,v\} \\ u\in S, v\notin S}} w(e).$$

U svakom iteraciji Primovog algoritma se dakle grana $e = \{u, v\}$ za koju se postiže ovaj minimum i čvor v dodaju povezujućem stablu grafa G koje se konstruiše. Na slici 8.7 je ilustrovana jedna iteracija Primovog algoritma u kojoj je grana $e = \{u, v\}$ izabrana kao grana minimalne težine od onih kod kojih je jedan kraj u S i drugi kraj je izvan S.



SLIKA 8.7: Jedna iteracija Primovog algoritma.

Na slici 8.8 je prikazano postupno izvršavanje Primovog algoritma za graf na slici 8.4(a) ukoliko je početni čvor izabran da bude čvor a.



SLIKA 8.8: Izvršavanje Primovog algoritma za graf na slici 8.4(a) od početnog čvora a.

Ispravnost Primovog algoritma je takođe neposredna posledica svojstva podele grafa.

LEMA *Primov algoritam konstruiše minimalno povezujuće stablo povezanog težinskog grafa G.*

Dokaz: Ispravnost Primovog algoritma se dokazuje vrlo slično Kruskalovom algoritmu. Zbog toga ćemo pokazati samo da se i u Primovom algoritmu u svakom koraku dodaje grana koja pripada svakom minimalnom povezujućem stablu grafa G.

Ali to je u Primovom algoritmu još očiglednije, jer u svakom koraku postoji podskup čvorova S nad kojim je delimično konstruisano povezujuće stablo. Pri tome se bira grana minimalne težine e sa jednim krajem u S i drugim krajem v u $V - S$, pa se grana e i čvor v dodaju stablu koje se konstruiše. Stoga na osnovu svojstva podele grafa neposredno sledi da grana e pripada svakom minimalnom povezujućem stablu grafa G. ■

Implementacija Primovog algoritma. Da bi se efikasno realizovao Primov algoritam, mora se pažljivo voditi evidencija o skupu čvorova S i skupu grana T koji pripadaju delimično konstruisanom povezujućem stablu grafa G. Pored toga, grane čiji je samo jedan kraj u skupu S moraju se organizovati na način koji omogućava da se može brzo izabrati takva grana minimalne težine. Zbog toga se u ralizaciji Primovog algoritma koriste ove strukture podataka:

- Za skup grana T delimičnog stabla koristi se reprezentacija u obliku povezane liste kako bi dodavanje grane u T moglo da se izvrši za konstantno vreme.
- Za skup čvorova S delimičnog stabla potrebna je reprezentacija koja omogućava da operacije provere članstva i dodavanja novog elementa skupu mogu da se izvrše za konstantno vreme. Obična reprezentacija pomoću *karakterističnog* niza skupa S zadovoljava ovaj uslov. Dužina ovog niza jednaka je broju čvorova grafa G i elementi niza su indeksirani tim čvorovima. Pored toga, svaki element niza ima vrednost tačno ili netačno prema tome da li odgovarajući čvor pripada skupu S.
- Da bi se brzo odredila sledeća grana koju treba dodati delimičnom stablu, koristi se red sa prioritetima čiji su elementi grane grafa G. Preciznije, svaki element tog reda ima dva polja: jedno polje sadrži granu $e = \{u, v\}$ i drugo polje sadrži njenu težinu $w(e)$. Za ključ (prioritet) odgovarajućeg elementa u redu uzima se težina grane u drugom polju.

Imajući ove napomene u vidu i pretpostavljajući reprezentaciju grafa G pomoću niza listi susedstva, Primov algoritam za nalaženje minimalnog povezujućeg stabla grafa G može se preciznije izraziti na pseudo jeziku na sledeći način:

```
// Ulaz:  povezan težinski graf G
// Izlaz: minimalno povezujuće stabla T grafa G
algorithm mst-prim(G)

   Proizvoljno izabrati čvor s u G;
   S = {s}; T = ∅   // inicijalizovati delimično stablo

   Q = ∅;   // konstruisati prazan red sa prioritetima
   for (svaka grana e={s,x} u grafu G) do
      pq-insert(Q,(w(e),e));

   while (Q != ∅) do
```

```
    e = pq-deletemin(Q);
    Neka je e = {u,v};
    T = T + e;  // granu e dodati granama delimičnog stabla
    S = S + v;  // drugi kraj v grane e dodati čvorovima
                // delimičnog stabla
    for (svaka grana e = {v,x} za koju x ∉ S) do
       pq-insert(Q,(w(e),e));

return T;
```

U vremenu izvršavanja algoritma `mst-prim` dominiraju dve petlje: u prvoj `for` petlji se inicijalizuje red sa prioritetima Q, dok se u drugoj `while` petlji konstruiše minimalno povezujuće stablo. Za graf G sa n čvorova i m grana vreme izvršavanja `for` petlje je $O(n \log n)$, pošto se u njoj izvršava najviše n operacija dodavanja elementa u red sa prioritetima veličine najviše n. Slično, u `while` petlji se izvršava najviše m operacija dodavanja i uklanjanja za red elemenata sa prioritetima dužine najviše m. Pošto se dodatne operacije u svakoj iteraciji te petlje izvršavaju za konstantno vreme, vreme izvršavanja `while` petlje iznosi $O(m \log m)$. Ukupno vreme izvršavanja algoritma `mst-prim` je dakle:

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

Kao zaključak se prema tome može izvesti činjenica da Primov i Kruskalov algoritam za nalaženje minimalnog povezujućeg stabla grafa imaju asimptotski jednako vreme izvršavanja.

8.3 Najkraći putevi od jednog čvora

Grafovi se često koriste za modeliranje mreža u kojima se prelazi neki put od jedne tačke do druge — na primer, od jednog grada do drugog preko puteva kojima su gradovi povezani ili od jednog računara do drugog preko komunikacionih linkova kojima su računari povezani. Zbog toga je jedan od osnovnih algoritamskih problema za grafove nalaženje najkraćih puteva između čvorova u grafu.

Neka je dat povezan težinski graf $G = (V, E)$. U vezi sa problemom nalaženja najkraćih puteva u grafu G prirodnije je podrazumevati da numeričke vrednosti pridružene granama grafa predstavljaju njihovu dužinu, a ne težinu. (Naravno, dužina ili težina mogu stvarno predstavljati nešto sasvim treće, recimo, vreme ili cenu.) Tome ćemo se prilagoditi u izlaganju

do kraja ovog poglavlja i govoriti da svaka grana $e = \{u, v\}$ u težinskom grafu G ima pridruženu dužinu $\ell(e) \geq 0$ koja predstavlja meru prelaska u jednom koraku od čvora u do čvora v.

Za neki put P u grafu G, dužina tog puta $\ell(P)$ je prosto zbir dužina svih grana na putu P. Formalno, ako se put P sastoji od čvorova $v_1, v_2, \ldots, v_k$ i grana $e_i = \{v_i, v_{i+1}\}$ između njih, onda je:

$$\ell(P) = \sum_{i=1}^{k-1} \ell(e_i).$$

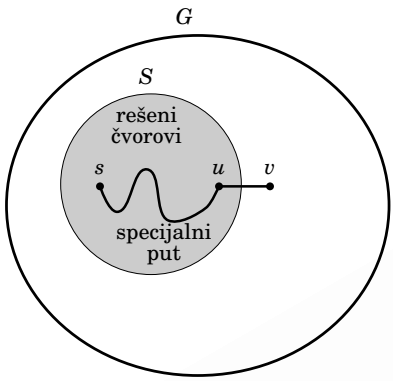
Problem najkraćih puteva od jednog čvora sastoji se u tome da se za dati čvor s u težinskom grafu G odrede najkraći putavi od tog čvora s do svih ostalih čvorova u grafu G. (Interesantno je da naizgled jednostavniji problem nalaženja najkraćeg puta od jednog datog čvora do drugog datog čvora ima istu složenost u opštem slučaju kao problem najkraćih puteva od jednog datog čvora do svih ostalih čvorova.) Jedno efikasno rešenje za ovaj problem prvi je predložio Edsger Dijkstra i po njemu se ono naziva *Dijkstrin algoritam*.

Dijkstrin algoritam

Podsetimo se da se problem nalaženja najkraćih puteva od jednog čvora u specijalnom slučaju običnog grafa, kod koga su sve grane jednake dužine, može rešiti pretragom u širinu. U Dijkstrinom algoritmu se ovaj pristup generalizuje za težinski graf tako što se na određen način primenjuje „ponderisana" pretraga u širinu od datog čvora. Radi jednostavnijeg izlaganja Dijkstrinog algoritma, najpre ćemo pojednostaviti problem i pretpostaviti da se traže samo *dužine* najkraćih puteva od čvora s do svih ostalih čvorova u grafu — same najkraće puteve je u algoritmu lako rekonstruisati.

Suštinska odlika Dijkstrinog algoritma je da se najkraći putevi od datog čvora s do ostalih čvorova otkrivaju u rastućem redosledu dužina najkraćih puteva. U algoritmu se postupno konstruiše skup čvorova S takav da je za svaki čvor u u S određena dužina najkraćeg puta $d(u)$ od čvora s. To se skup *rešenih* čvorova grafa G. Na početku je $S = \{s\}$ i $d(s) = 0$.

Da bi se u narednoj iteraciji skup rešenih čvorova S proširio za jedan čvor, za svaki nerešen čvor v u $V - S$ se određuje dužina najkraćeg *specijalnog* puta od s do v. Specijalni put do čvora v je neki put koji počinje od čvora s, zatim prolazi samo kroz rešene čvorove i, na samom kraju, dolazi do nekog čvora $u \in S$ i završava se jednom granom u čvoru v. Jedan specijalni put do čvora v je ilustrovan na slici 8.9.



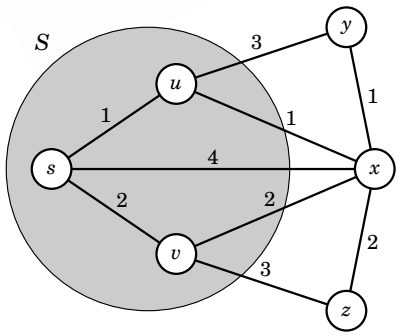
SLIKA 8.9: Specijalni put od s do v u Dijkstrinom algoritmu.

Drugim rečima, za svaki nerešen čvor v izvan skupa S određuje se veličina

$$\delta(v) = \min_{\substack{e=\{u,v\} \\ u \in S}} d(u) + \ell(e).$$

Nakon toga, radi proširenja skupa S, bira se čvor x za koji se ova veličina minimizuje, odnosno bira se čvor x takav da je $\delta(x) = \min_{v \in V-S} \delta(v)$. Taj čvor se dodaje skupu S i za njegovu dužinu najkraćeg puta $d(x)$ uzima se $\delta(x)$. Skup S se proširuje na ovaj način za jedan čvor sve dok se ne obuhvate svi čvorovi iz skupa V grafa G.

Verovatno nije odmah jasno zašto je ovaj postupak ispravan. Zato da bismo ga bolje razumeli, posmatrajmo jedno moguće stanje izvršavanja Dijkstrinog algoritma koje je prikazano na slici 8.10.

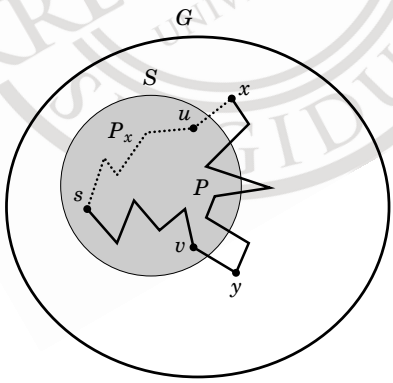


SLIKA 8.10: Primer izvršavanja Dijkstrinog algoritma.

U trenutku izvršavanja Dijkstrinog algoritma na slici 8.10 završene su dve iteracije: skup S je u prvoj iteraciji bio proširen za čvor u i u drugoj za čvor v. U trećoj iteraciji bi se dodao čvor x, jer se za taj čvor postiže najmanja vrednost $\delta(x)$. Naime, $\delta(x) = \min\{1+1, 0+4, 2+2\} = 2$, $\delta(y) = 1+3 = 4$ i $\delta(z) = 2+3 = 5$. Obratite pažnju na to da bi dodavanje čvora y ili z skupu S u ovom trenutku bilo pogrešno; oni će se ipak ispravno dodati u narednim iteracijama zbog grana koje ih povezuju sa čvorom x.

Primetimo da se u svakoj iteraciji Dijkstrinog algoritma traži najkraći specijalni put od čvora s i da se skupu S dodaje čvor x koji se nalazi na kraju tog puta. Takođe, taj najkraći specijalni put P_x od s do x proglašava se za *globalno* najkraći put od s do x, odnosno vrednost $d(x)$ izjednačava se sa $\delta(x)$. Zašto je ovaj pristup ispravan? Ključni razlog je to što su dužine svih grana u grafu nenegativne — u stvari, ukoliko je dužina neke grane negativna, rezultat Dijkstrinog algoritma može biti pogrešan.

Da bismo pokazali da ne postoji kraći put od s do x u grafu G, posmatrajmo bilo koji drugi put P od s do x. Ovaj put mora negde izaći iz skupa S, jer je čvor x izvan S. Neka je y prvi čvor na putu P koji nije u S i neka je P_y deonica puta P od s do y. Kao što je prikazano na slici 8.11, put P iza čvora y može krivudati unutar i izvan skupa S nekoliko puta dok se najzad ne završi u čvoru x.



SLIKA 8.11: Najkraći specijalni put P_x i proizvoljni put P od čvora s do čvora x čiji je prvi deo do čvora y specijalan.

Kako je P_y jedan od specijalnih puteva od čvora s i P_x je najkraći takav specijalni put, to je $\ell(P_y) \geq \ell(P_x)$. Sa druge strane, $\ell(P) \geq \ell(P_y)$ jer su dužine grana nenegativne. Ove dve nejednakosti zajedno daju $\ell(P) \geq \ell(P_x)$, što pokazuje da dužina svakog puta P od s do x nije kraća od dužine

puta P_x. Preciznije tvrđenje o ispravnosti Dijkstrinog algoritma može se formulisati na sledeći način:

LEMA *Skup S u toku izvršavanja Dijkstrinog algoritma sadrži čvorove x takve da je $d(x)$ dužina najkraćeg puta P_x od čvora s do čvora x.*

Dokaz: Dokaz tvrđenja izvodimo indukcijom po broju čvorova u S. U baznom slučaju za $S = \{s\}$ tvrđenje je očigledno, jer je $d(s) = 0$. Pretpostavimo da je tvrđenje tačno kada S ima $k \geq 1$ čvorova i pokažimo da ono važi i za skup S uvećan za jedan čvor x u Dijkstrinom algoritmu.

Neka je grana $\{u, x\}$ poslednja grana na specijalnom putu P_x. Kako je u neki čvor u S, po indukcijskoj pretpostavci je $d(u)$ dužina najkraćeg puta P_u od čvora s do čvora u. Neka je P bilo koji drugi put u grafu od s do x; naš cilj je da pokažemo da njegova dužina nije manja od dužine puta P_x. Put P mora negde izaći iz skupa S, jer je čvor x izvan S. Neka je y prvi čvor na putu P koji nije u S i neka je čvor v u S odmah ispred čvora y na putu P (slika 8.11).

Neka je P' deonica puta P od s do v. Kako je čvor v u S, po indukcijskoj pretpostavci dužina puta P' nije manja od dužine najkraćeg puta od s do v, odnosno $\ell(P') \geq d(v)$. Stoga deonica puta P od s do y ima dužinu $\ell(P') + \ell(v, y) \geq d(v) + \ell(v, y) \geq \delta(y)$. Pošto je u Dijkstrinom algoritmu čvor x izabran da bude dodat skupu S, to je $\delta(y) \geq \delta(x) = d(x) = \ell(P_x)$. Uzimajući sve ovo u obzir, kao i činjenicu da dužina deonice puta P od s do y nije manja od dužine celog puta P jer su grane nenegativne, najzad dobijamo traženu nejednakost $\ell(P) \geq \ell(P') + \ell(v, y) \geq \delta(y) \geq \delta(x) = d(x) = \ell(P_x)$. ■

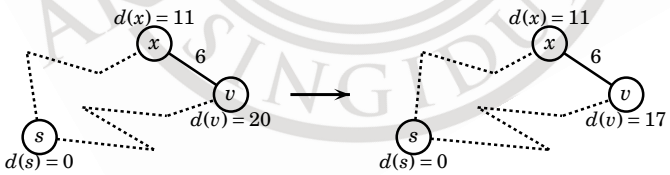
Implementacija Dijkstrinog algoritma. Za težinski graf G sa n čvorova i m grana, u Dijkstrinom algoritmu se izvršava $n - 1$ iteracija dodavanja novog čvora x skupu S. Na prvi pogled, u svakoj iteraciji se zahteva ispitivanje svakog čvora v u $V - S$ i svih grana između čvorova u S i čvora v radi izračunavanja najpre minimuma $\delta(v)$ i zatim izbora čvora x za koji se postiže najmanji od ovih minumuma. U najgorem slučaju dakle vreme izvršavanja svake iteracije može biti $O(m)$, pa bi ukupno vreme izvršavanja Dijkstrinog algoritma bilo $O(mn)$.

Da bi se efikasnost Dijkstrinog algoritma znatno unapredila, moraju se koristiti prave strukture podataka. Pre svega, da se ne bi nepotrebno u svakoj iteraciji ponovo izračunavale dužine najkraćih specijalnih puteva do čvorova u $V - S$, one se čuvaju u nizu d dužine n koji je indeksiran čvorovima grafa G. Tačnije, za svaki rešen čvor v u S, element $d(v)$ jednak je dužini globalno najkraćeg puta od s do v, dok za svaki nerešen čvor v u

$V - S$, element $d(v)$ jednak je dužini najkraćeg specijalnog puta od s do v. Početno je dakle $d(s) = 0$ i $d(v) = \ell(s,v)$ ukoliko u grafu G postoji grana od s do v ili po konvenciji $d(v) = \infty$ ukoliko takva grana ne postoji.

Dodatno poboljšanje može se obezbediti u svakoj iteraciji kada se bira nerešen čvor x koji ima najmanju vrednost $d(x)$ od svih nerešenih čvorova. Radi efikasne realizacije tog postupka može se postupiti na sličan način kao kod Primovog algoritma i koristiti struktura podataka reda sa prioritetima za sve nerešene čvorove u $V - S$. Ako jedan element takvog reda predstavlja čvor v u $V - S$ sa prioritetom jednakim elementu $d(v)$ u nizu d, onda se može efikasno odrediti element tog reda sa najmanjim prioritetom, odnosno nerešen čvor x koji ima najkraću dužinu $d(x)$ specijalnog puta od čvora s.

Nakon dodavanja izabranog čvora x skupu S, mora se uzeti u obzir da je čvor x rešen i da možda postoje kraći specijalni putevi koji vode do njegovih suseda. Zbog toga se mora proveriti vrednost $d(v)$ za svaki nerešen čvor v koji je susedan čvoru x. Naime, za svaki takav čvor v, novi specijalni put do v prati put od s do x u novom S i završava se granom $\{x,v\}$. Taj novi specijalni put do čvora v ima dužinu $d(x) + \ell(x,v)$, pa ako je taj put kraći od postojećeg najkraćeg specijalnog puta do v dužine $d(v)$, ta stara vrednost $d(v)$ mora se ažurirati vrednošću $d(x) + \ell(x,v)$. Jedan primer ovog slučaja prikazan je na slici 8.12.



SLIKA 8.12: Podešavanje nerešenog čvora v.

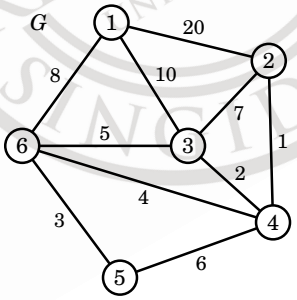
U slučaju kada vrednost $d(x) + \ell(x,v)$ nije manja od stare vrednosti $d(v)$ nerešenog čvora v, novi specijalni put do v nije kraći od postojećeg, pa stara vrednost $d(v)$ treba da ostane nepromenjena. Oba slučaja podešavanja vrednosti $d(v)$ nerešenog čvora v koji je susedan novododatom čvoru x mogu se kraće zapisati u obliku $d(v) = \min\{d(v), d(x) + \ell(x,v)\}$. Ova operacija podešavanja dužine najkraćeg specijalnog puta do nerešenog čvora naziva se *popuštanje*. Metafora za ovaj termin je stanje opruge koja se najpre napinje i zatim opušta do svog normalnog oblika u stanju mirovanja.

Precizniji zapis Dijkstrinog algoritma na pseudo jeziku kojim su obu-

hvaćene prethodne napomene je:

```
// Ulaz:  težinski graf G, čvor s u G
// Izlaz: niz d dužina najkaćih puteva od s do svih čvorova
algorithm dijkstra(G,s)

   S = {s};
   d[s] = 0;
   for (svaki čvor v ≠ s u G) do
      if ({s,v} grana u G) then
         d[v] = ℓ(s,v);
      else
         d[v] = ∞;

   while (S ≠ V) do
      Odrediti čvor x ∈ V − S za koji je d[x] najmanje;
      S = S + x;
      for (svaki čvor v ∈ V − S koji je spojen granom sa x) do
         d[v] = min{d[v], d[x] + ℓ(x,v)}  // popuštanje

   return d;
```

Na slici 8.13 je ilustrovano izvršavanje algoritma `dijkstra` za jedan dati graf i čvor u tom grafu.



iteracija	x	S	$d(1)$	$d(2)$	$d(3)$	$d(4)$	$d(5)$	$d(6)$
0	–	{1}	0	20	10	∞	∞	8
1	6	{1,6}	0	20	10	12	11	8
2	3	{1,3,6}	0	17	10	12	11	8
3	5	{1,3,5,6}	0	17	10	12	11	8
4	4	{1,3,4,5,6}	0	13	10	12	11	8
5	2	{1,2,3,4,5,6}	0	13	10	12	11	8

SLIKA 8.13: Izvršavanje Dijkstrinog algoritma za graf G i $s = 1$.

Radi analize Dijkstrinog algoritma, pretpostavimo da je ulazni graf G predstavljen nizom listi susedstva i da se za realizaciju reda sa prioritetima koristi binarni min-hip. Onda se inicijalizacija hipa izvršava za vreme $O(n)$. Izbor čvora $x \notin S$ koji minimizuje rastojanja d sastoji se od uklanjanja korena iz hipa, što se izvršava za vreme $O(\log n)$. Dodavanje čvora x skupu S izvršava se za konstantno vreme ukoliko je skup S predstavljen, recimo, povezanom listom. Postupak popuštanja se sastoji od ispitivanja svake grane koja povezuje čvor x i nekog njegovog suseda v koji se nalazi izvan novog S radi provere da li je $d(x) + \ell(x,v) < d(v)$. Ako je to slučaj, mora se promeniti vrednost $d(v)$ i čvor v s tim promenjenim prioritetom „pogurati" uz hip, što se opet izvršava za vreme $O(\log n)$. Primetimo da se ovo radi najviše jedanput po svakoj grani grafa.

Prema tome, u algoritmu `dijkstra` se obavlja $n-1$ operacija uklanjanja korena iz hipa i najviše m operacija pomeranja čvora na njegovo pravo mesto u hipu. Ukupno vreme za izvršavanja tih operacija je dakle $O((n+m)\log n)$. Ako je dati graf povezan, onda je $m \geq n-1$, pa se u tom slučaju vreme izvršavanja Dijkstrinog algoritma može kraće izraziti u obliku $O(m \log n)$.

8.4 Najkraći putevi između svih čvorova

Problem najkraćih puteva između svih čvorova sastoji se u nalaženju najkraćih puteva između svaka dva čvora u težinskom grafu G. Ako u grafu nema negativnih grana, onda se ovaj problem može rešiti višestrukim izvršavanjem Dijkstrinog algoritma od svakog čvora grafa. Ukoliko G ima n čvorova i m grana, vreme potrebno za ovo rešenje je $O(nm \log n)$, pod pretpostavkom da je graf G povezan i predstavljen nizom listi susedstva. U najgorem slučaju kada je ulazni graf gust, odnosno $m = \Theta(n^2)$, ovo vreme iznosi $O(n^3 \log n)$. U najboljem slučaju kada je graf redak, odnosno $m = \Theta(n)$, ovo vreme iznosi $O(n^2 \log n)$.

U ovom odeljku ćemo pokazati drugi način za rešavanje problema nalaženja najkraćih puteva između svih čvorova koji se može primeniti čak i u slučaju kada graf sadrži grane negativne dužine (ali ne i cikluse negativne dužine). Taj način se po svom autoru naziva *Flojdov algoritam* i pripada kategoriji algortama dinamičkog programiranja. U ovom algoritmu se pretpostavlja da je ulazni graf predstavljen matricom susedstva i njegovo vreme izvršavanja je $O(n^3)$.

Flojdov algoritam. Flojdov algoritam za rešenje problema nalaženja najkraćih puteva između svih čvorova je konceptualno jednostavniji od Dijkstrinog algoritma, mada rešava očigledno teži problem.

Neka je $G = (V, E)$ težinski graf i neka su svi čvorovi grafa G proizvoljno numerisani tako da je $V = \{v_1, v_2, \ldots, v_n\}$. Flojdov algoritam se zasniva na ideji da se nalaženje najkraćeg puta između bilo koja dva čvora v_i i v_j podeli na nalaženje dva najkraća puta: jednog puta od v_i do nekog posrednog čvora i drugog puta od tog posrednog čvora do v_j. Tačnije, najkraći put od v_i do v_j određuje se koristeći na tom putu samo posredne čvorove iz skupa prvih k čvorova $V_k = \{v_1, v_2, \ldots, v_k\}$. Ali taj postupak se ponavlja redom za svako $k = 1, 2, \ldots, n$, odnosno za niz skupova $V_1, V_2, \ldots, V_n$, tako da se na kraju za $V_n = V$ dobija globalno najkraći put između v_i i v_j.

Da bismo pojednostavili opis Flojdovog algoritma, pretpostavimo opet da se traže samo dužine najkraćih puteva, a ne i sami putevi. Dodatno, radi pogodnosti označimo $V_0 = \emptyset$. Dužina najkraćeg puta $D(i, j, k)$ od čvora v_i do čvora v_j, uzimajući u obzir posredne čvorove na putevima od v_i do v_j samo one iz skupa V_k, izračunava se za $k = 0, 1, 2, \ldots, n$ na sledeći način. Početno, za $k = 0$,

$$D(i, j, 0) = \begin{cases} 0, & \text{ako } i = j \\ \ell(v_i, v_j), & \text{ako } (v_i, v_j) \in E \\ \infty, & \text{inače} \end{cases}$$

a za $k = 1, 2, \ldots, n$,

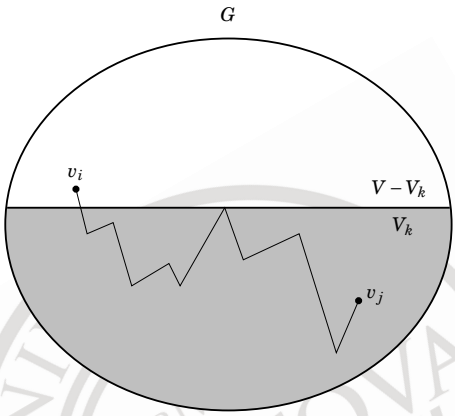
$$D(i, j, k) = \min\{D(i, j, k-1),\ D(i, k, k-1) + D(k, j, k-1)\}.$$

Rečima, najkraći put koji ide od v_i do v_j i prolazi samo kroz čvorove iz skupa V_k je jednak kraćem od dva moguća najkraća puta. Prva mogućnost je da najkraći put uopšte ne prolazi kroz čvor v_k — to je onda dužina najkraćeg puta od v_i do v_j koji prolazi samo kroz čvorove iz skupa V_{k-1}. Druga mogućnost je da najkraći put (jedanput) prolazi kroz čvor v_k — to je onda zbir dužina najkraćeg puta od v_i do v_k koji prolazi samo kroz čvorove iz skupa V_{k-1} i najkraćeg puta od v_k do v_j koji prolazi samo kroz čvorove iz skupa V_{k-1}.

Pored toga, ne postoji drugi kraći put između v_i i v_j od ova dva koji prolazi kroz čvorove iz V_k. Ako bi postojao takav kraći put koji ne sadrži čvor v_k, tada bi to protivrečilo definiciji $D(i, j, k-1)$, a ako bi postojao takav kraći put koji sadrži čvor v_k, tada bi to protivrečilo definiciji $D(i, k, k-1)$ ili

$D(k,j,k-1)$. U stvari, lako se proverava da ovo važi i ukoliko postoje grane grafa G negativne dužine, ali ne i ciklusi negativne dužine.

Na slici 8.14 je ilustrovano izračunavanje dužine $D(i,j,k)$ prema prethodnim formulama. Na toj slici, krajnji čvorovi v_i i v_j mogu pripadati ili ne pripadati skupu V_k koji je predstavljen sivim delom grafa G.

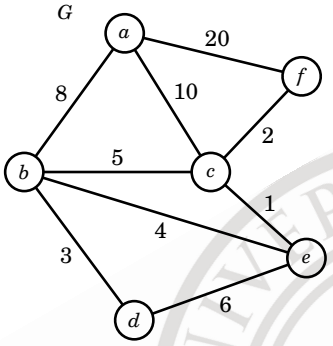


SLIKA 8.14: Izračunavanje $D(i,j,k)$ u Flojdovom algoritmu.

Flojdov algoritam na psudo jeziku je jednostavna iterativna implementacija prethodne rekurzivne formule za izračunavanje dužine najkraćeg puta između svaka dva čvora. Ulazni graf G sa n čvorova je predstavljen $n \times n$ matricom susedstva M kod koje je element $M(i,j)$ jednak dužini grane između čvorova v_i i v_j ili je jednak ∞ ukoliko između tih čvorova ne postoji grana. Pored toga, $M(i,i) = 0$.

```
// Ulaz:  težinski graf G predstavljen matricom susedstva M
// Izlaz: matrica D takva da D(i,j,n) = d(v_i,v_j)
algorithm floyd(G)

   for i = 1 to n do
      for j = 1 to n do
         D[i,j,0] = M[i,j];

   for k = 1 to n do
      for i = 1 to n do
         for j = 1 to n do
            D[i,j,k] = min {D[i,j,k-1], D[i,k,k-1] + D[k,j,k-1]};

   return D;
```

Obratite pažnju na to da se u Flojdovom algoritmu izračunava trodimenzionalna matrica $D(i,j,k)$, ali to se može zamisliti kao izračunavanje niza dvodimenzionalnih matrica $D_k(i,j)$ za $k = 1,2,\ldots,n$. Matrica D_k se izračunava u k-toj iteraciji prve od tri ugnježđene `for` petlje. U nastavku je prikazan ovaj niz dvodimenzionalnih matrica koje se dobijaju za dati graf G u svakoj iteraciji te petlje.



D_0	a	b	c	d	e	f
a	0	8	10	∞	∞	20
b	8	0	5	3	4	∞
c	10	5	0	∞	1	2
d	∞	3	∞	0	6	∞
e	∞	4	1	6	0	∞
f	20	∞	2	∞	∞	0

D_1	a	b	c	d	e	f
a	0	8	10	∞	∞	20
b	8	0	5	3	4	28
c	10	5	0	∞	1	2
d	∞	3	∞	0	6	∞
e	∞	4	1	6	0	∞
f	20	28	2	∞	∞	0

D_2	a	b	c	d	e	f
a	0	8	10	11	12	20
b	8	0	5	3	4	28
c	10	5	0	8	1	2
d	11	3	8	0	6	31
e	12	4	1	6	0	32
f	20	28	2	31	32	0

D_3	a	b	c	d	e	f
a	0	8	10	11	11	12
b	8	0	5	3	4	7
c	10	5	0	8	1	2
d	11	3	8	0	6	10
e	11	4	1	6	0	3
f	12	7	2	10	3	0

D_4	a	b	c	d	e	f
a	0	8	10	11	11	12
b	8	0	5	3	4	7
c	10	5	0	8	1	2
d	11	3	8	0	6	10
e	11	4	1	6	0	3
f	12	7	2	10	3	0

D_5	a	b	c	d	e	f
a	0	8	10	11	11	12
b	8	0	5	3	4	7
c	10	5	0	7	1	2
d	11	3	7	0	6	9
e	11	4	1	6	0	3
f	12	7	2	9	3	0

D_6	a	b	c	d	e	f
a	0	8	10	11	11	12
b	8	0	5	3	4	7
c	10	5	0	7	1	2
d	11	3	7	0	6	9
e	11	4	1	6	0	3
f	12	7	2	9	3	0

Ispravnost Flojdovog algoritma može se lako pokazati indukcijom po broju iteracija prve od tri ugnježđene for petlje: nakon k iteracija te petlje, $D(i,j,k)$ sadrži dužinu najkraćeg puta od čvora v_i do čvora v_j koji prolazi samo kroz čvorove iz skupa V_k. Pored toga, ako je graf G predstavljen matricom susedstva, vreme izvršavanja Flojdovog algoritma je očigledno $O(n^3)$. To sledi na osnovu toga što je u algoritmu vremenski dominantno izvršavanje trostruko ugnježđenih for petlji.

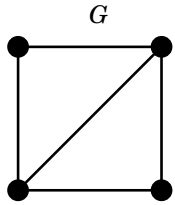
U slučaju kada svaka grana grafa ima nenegativnu dužinu, Flojdov algoritam se može poboljšati tako da se umesto trodimenzionalne matrice koristi dvodimenzionalna matrica. Naime, onda su elementi u k-toj vrsti i k-toj koloni matrice D_k isti kao oni u matrici D_{k-1}, pošto je $D_{k-1}(k,k)$ uvek nula.

Preciznije rečeno, za svako $k = 1,2,\ldots,n$ važi $D_k(i,k) = D_{k-1}(i,k)$ za svako $i = 1,2,\ldots,n$ i važi $D_k(k,j) = D_{k-1}(k,j)$ za svako $j = 1,2,\ldots,n$. Kako se nijedan element čiji je jedan ili drugi indeks jednak k ne menja u k-toj iteraciji prve od tri ugnježđene for petlje, nova vrednost za $D_k(i,j)$ može se izračunati kao minimum stare vrednosti $D_{k-1}(i,j)$ i zbira novih vrednosti $D_k(i,k)$ i $D_k(k,j)$, a ne kao minimum stare vrednosti $D_{k-1}(i,j)$ i zbira starih vrednosti $D_{k-1}(i,k)$ i $D_{k-1}(k,j)$. To znači da se u Flojdovom algoritmu umesto trodimenzionalne matrice D reda $n \times n \times n$ može koristiti dvodimenzionalna matrica D reda $n \times n$.

Ova verzija Flojdovog algoritma je jednostavna modifikacija originalne verzije i ostavlja se čitaocima za vežbu. Obratite pažnju na to da treba zapravo samo zanemariti treći indeks matrice D.

Zadaci

1. Odredite sva povezujuća stabla grafa G na slici 8.15.



SLIKA 8.15.

2. Dokažite da je graf povezan ako i samo ako ima povezujuće stablo.

3. *„Inverzni" Kruskalov algoritam.* Pokažite ispravnost sledećeg algoritma za konstruisanje minimalnog povezujućeg stabla težinskog grafa $G = (V, E)$. U algoritmu se polazi od celog grafa (V, E) i zatim se uklanjaju grane u opadajućem redosledu njihove težine. Specifično, kako se dolazi do neke grane e u ovom redosledu (počinjući od grane najveće težine), grana e se uklanja ukoliko bi bez nje dotadašnji graf i dalje bio povezan, a u suprotnom slučaju se grana e zadržava. (*Savet:* Dokažite sledeće *svojstvo ciklusa grafa*: Ako je e jedinstvena grana najveće težine u nekom ciklusu grafa G, onda e ne pripada nijednom minimalnom povezujućem stablu grafa G.)

4. Napišite i analizirajte algoritam za nalaženje povezanih komponenti grafa G koji ne koristi pretragu u dubinu ili širinu. (*Savet:* Ideja algoritma je slična Kruskalovom algoritmu.)

5. Napištite drugu verziju Primovog algoritma pod pretpostavkom da je ulazni graf G sa n čvorova predstavljen težinskom matricom susedstva. Ta verzija Primovog algoritma treba da se izvršava za vreme $O(n^2)$ nezavisno od broja grana.

6. Pokažite kako se najkraći putevi od datog čvora s do svakog drugog čvora u grafu mogu rekonstruisati u Dijkstrinom algoritmu usput kako se određuju dužine tih puteva.

7. Navedite kontraprimer grafa sa negativnim granama za koji Dijkstrin algoritam ne radi ispravno.

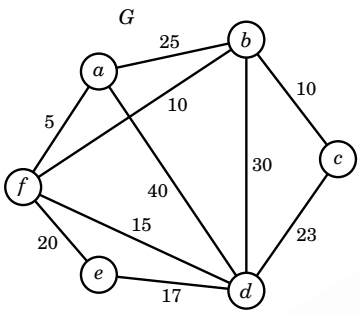
8. Ukoliko je težinski graf G na slici 8.16 ulazni graf za algoritme iz ovog poglavlja, slikovito ilustrujte izvršavanje:

a) Kruskalovog algoritma;

b) Primovog algoritma od početnog čvora b;

c) Dijkstrinog algoritma za dati čvor a;

d) Flojdovog algoritma.



SLIKA 8.16.

9. Odredite vreme izvršavanja algoritma `dijkstra` ukoliko se koristi matrica susedstva za ulazni graf G, povezane liste za skupove S i $V - S$, kao i niz d za dužine najkraćih puteva od datog čvora s.

10. Napišite i analizirajte algoritam za problem najkraćih puteva od jednog čvora u težinskom usmerenom acikličkom grafu.

11. Napišite Flojdov algoritam u kome se koristi samo jedna dvodimenzionalna matrica za dužine najkraćih puteva između svaka dva čvora grafa sa nenegativnim granama.

12. *Voršalov algoritam.* Problem povezanosti običnog (netežinskog) usmerenog grafa G sastoji se od utvrđivanja da li postoji put između svaka dva čvora. Ako je graf G predstavljen matricom susedstva M, problem povezanosti grafa G svodi se na izračunavanje druge matrice P takve da je $P(i,j) = 1$ ukoliko postoji put od čvora v_i do čvora v_j i $P(i,j) = 0$ u suprotnom slučaju. Napišite i analizirajte algoritam kojim se izračunava matrica P. (*Savet:* Modifikujte Flojdov algoritam.)

UNIVERZITET
PRIVATNI
AKREDITOVANI
UNIVERZITET
SINGIDUNUM

Literatura

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1987.

[2] Kenneth P. Bogart, Clifford Stein, and Robert Drysdale. *Discrete Mathematics for Computer Science*. Key College, 2005.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

[4] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, second edition, 1994.

[5] Dejan Živković. *Osnove dizajna i analize algoritama*. Računarski fakultet i CET, Beograd, 2007.

[6] Dejan Živković. *Foundations of Algorithm Design and Analysis*. VDM Verlag, 2009.

[7] Miodrag Živković. *Algoritmi*. Matematički fakultet, Beograd, 2000.

[8] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison-Wesley, 2005.

[9] Anany Levitin. *Introduction to the Design and Analysis of Algorithms*. Addison-Wesley, second edition, 2006.

[10] James A. Storer. *An Introduction to Data Structures and Algorithms*. Birkhauser Boston, 2002.

[11] Milo Tomašević. *Algoritmi i strukture podataka*. Akademska misao, Beograd, 2008.

[12] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Addison-Wesley, second edition, 2007.

UNIVERZITET
PRIVATNI
AKREDITOVANI
UNIVERZITET
SINGIDUNUM

Indeks

Odlukom Senata Univerziteta "Singidunum", Beograd, broj 636/08 od 12.06.2008, ovaj udžbenik je odobren kao osnovno nastavno sredstvo na studijskim programima koji se realizuju na integrisanim studijama Univerziteta "Singidunum".

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.421(075.8)
510.5(075.8)
004.6(075.8)

ЖИВКОВИЋ, Дејан, 1956-
Uvod u algoritme i strukture podataka / Dejan Živković. - 1. izd. - Beograd : Univerzitet Singidunum, 2010 (Loznica : Mladost Grup). - IV, 257 str. : ilustr. ; 24 cm

Tiraž 240. - Bibliografija: str. 251. - Registar.

ISBN 978-86-7912-239-1

a) Алгоритми b) Подаци (рачунарство)
COBISS.SR-ID 172433420