Predrag Janičić Filip Marić

Programiranje 2

Osnove programiranja kroz programski jezik C

Matematički fakultet

Predrag Janičić Filip Marić

PROGRAMIRANJE 2

Osnove programiranja kroz programski jezik C

Beograd
2023.

Autori:
dr Predrag Janičić, redovni profesor na Matematičkom fakultetu u Beogradu
dr Filip Marić, redovni profesor na Matematičkom fakultetu u Beogradu

PROGRAMIRANJE 2

Izdavač: Matematički fakultet Univerziteta u Beogradu
Studentski trg 16, 11000 Beograd
Za izdavača: *prof. dr Zoran Rakić*, dekan

Recenzenti:
dr Jelena Graovac, docent na Matematičkom fakultetu u Beogradu
dr Dragan Urošević, naučni savetnik na Matematičkom institutu SANU

Obrada teksta, crteži i korice: *autori*

ISBN 978-86-7589-156-7

Sadržaj

Predgovor

Ova knjiga pisana je kao udžbenik za predmet *Programiranje 2* na smeru *Informatika* Matematičkog fakulteta u Beogradu, na osnovu materijala za predavanja koja smo na ovom predmetu držali od 2005. godine.

U ovom predmetu i u ovoj knjizi, centralno mesto ima programski jezik C, ali predmet i knjiga, zajedno sa prvim delom (*Programiranje 1 — Osnove programiranja kroz programski jezik C*, istih autora) pokušavaju da daju šire osnove programiranja, ilustrovane kroz jedan konkretan jezik. Knjigu dopunjuje i zbirka zadataka: Milena Vujošević Janičić, Jelena Graovac, Nina Radojičić, Ana Spasić, Mirko Spasić, Anđelka Zečević: *Programiranje 2 - Zbirka zadataka sa rešenjima (u programskom jeziku C)*, Matematički fakultet, 2016.

Na kraju većine poglavlja naveden je veći broj pitanja i zadataka koji mogu da služe za proveru znanja. Među ovim pitanjima su mnoga koja su zadata na testovima i ispitima iz predmeta *Programiranje 2*. Odgovori na pitanja nisu eksplicitno navođeni, jer su već implicitno sadržani u osnovnom tekstu knjige.

U pripremi knjige koristili smo mnoge izvore, pre svega sa interneta. Pomenimo ovde samo knjigu *Praksa programiranja* (*The Practice of Programming*) Brajana Kernigena i Roba Pajka (Addison-Wesley, 1999).

Na veoma pažljivom čitanju i mnoštvu sugestija zahvaljujemo se recenzentima Jeleni Graovac i Draganu Uroševiću. Zahvaljujemo asistentima koji su kreirali zadatke za pismene ispite iz predmeta *Programiranje 2*, a od kojih su neki navedeni na kraju lekcija kao zadaci za vežbu. Na brojnim sugestijama i ispravkama zahvalni smo i nastavnicima Matematičkog fakulteta Vesni Marinković, Mileni Vujošević-Janičić, Mladenu Nikoliću, Nini Radojičić Matić, Ivanu Čukiću, kolegi Saši Đorđeviću, kao i studentima Nikoli Premčevskom, Katarini Smiljanić, Nikoli Ajzenhameru, Aleksandru Zečeviću, Nemanji Mićoviću, Luki Jovičiću, Dalmi Beari, Đorđu Stanojeviću, Jovani Pejkić, Aleksandru Jakovljeviću, Davidu Dimiću, Mateji Marjanoviću, Igoru Grkavcu, Filipu Broćiću, Marku Spasiću, Pavlu Joksoviću, Stefanu Saviću, Milošu Mikoviću, Aleksandri Stojanović, Petru Vukmiroviću, Ivanu Baleviću, Vojislavu Grujiću, Tomislavu Savatijeviću, Tini Mladenović i Milošu Petričkoviću.

net strana autora. Besplatna dostupnost odražava stav autora o otvorenim sadržajima — kodu programa i sadržaju knjiga.

Beograd, april 2022. godine *Autori*

Predgovor drugom izdanju

U drugom izdanju nema novog materijala, već su samo ispravljene greške uočene u prvom izdanju. Na novim komentarima zahvalni smo koleginici Jeleni Graovac. I ovo, drugo izdanje knjige dostupno je (besplatno) u elektronskom obliku preko internet strana autora. Sadržaj štampanog i elektronskog izdanja je identičan.

Beograd, septembar 2022. godine *Autori*

Predgovor trećem izdanju

U trećem izdanju ispravljeno je nekoliko sitnih grešaka i modifikovano nekoliko primera. Na novim komentarima zahvalni smo koleginici Jeleni Graovac. I ovo izdanje knjige dostupno je (besplatno) u elektronskom obliku preko internet strana autora.

Beograd, februar 2023. godine *Autori*

Deo I

Čitljivost, ispravnost i efikasnost programa

Glava 1

Principi pisanja programa

Programi napisani na višem programskom jeziku sredstvo su komunikacije između čoveka i računara ali i između ljudi samih. Razumljivost, čitljivost programa, iako nebitna za računar, od ogromne je važnosti za kvalitet i upotrebljivost programa. Naime, u održavanje programa obično se uloži daleko više vremena i truda nego u njegovo pisanje, a održavanje sistema često ne rade oni programeri koji su program napisali. Pored toga, razumljivost programa omogućava lakšu analizu njegove ispravnosti i složenosti. Preporuke za pisanje često nisu kruta pravila, već predstavljaju samo smernice i ideje kojima se treba rukovoditi u pisanju programa, u aspektima formatiranja, nazubljivanja, imenovanja promenljivih i funkcija, itd.

U daljem tekstu će, kao na jedan primer konvencija za pisanje programa, biti ukazivano na preporuke iz teksta *Linux Kernel Coding Style*, Linusa Torvaldsa, autora operativnog sistema Linux koji je napisan na jeziku C. Nekoliko saveta i preporuka u nastavku preuzeto je iz znamenite knjige *The Practice of Programming* autora Brajana Kernigena i Roba Pajka. Preporuke navedene u nastavku često se odnose na sve programske jezike, ali ponekad samo na jezik C. I sve ove savete i preporuke treba razmatrati sa rezervom, jer postoje i mnoge druge grupe preporuka i konvencija.

1.1 Timski rad i konvencije

Za svaki obimniji projekat potrebno je usaglasiti konvencije za pisanje programa. Da bi ih se lakše pridržavalo, potrebno je detaljno motivisati i obrazložiti pravila. Ima različitih konvencija i one često izazivaju duge i zapaljive rasprave između programera. Mnogi će, međutim, reći da nije najvažnije koja konvencija se koristi, nego koliko strogo se nje pridržava. Strogo i konzistentno pridržavanje konvencije u okviru jednog projekta izuzetno je važno za njegovu uspešnost. Jedan isti programer treba da bude spreman da u različitim timovima i različitim projektima koristi različite konvencije.

Kako bi se olakšalo baratanje programom koji ima na stotine datoteka koje menja veliki broj programera, u timskom radu obično se koriste *sistemi za kontrolu verzija* (kao što su git, SVN, CVS, Mercurial, Bazaar). I ovi sistemi nameću dodatna pravila i omogućavaju dodatne konvencije koje tim treba da poštuje (na primer, konvencija može da bude da u zajedničku verziju programa ne može da se stavi datoteka sa kojom se čitav program ne kompilira uspešno).

1.2 Vizuelni elementi programa

Prva ideja o programu formira se na osnovu njegovog izgleda – njegovih vizuelnih elemenata, kao što su broj linija u datoteci, broj karaktera u liniji, nazubljivanje, grupisanje linija i slično. Vizuelni elementi programa i njegovo formatiranje često su od ključne važnosti za njegovu čitljivost. Formatiranje, konkretno nazubljivanje, u nekim jezicima (na primer, Python) čak utiče na značenje programa.

Formatiranje i vizuelni elementi programa treba da olakšaju razumevanje koda koji se čita, ali i pronalaženje potrebnog dela koda ili datoteke sa nekim delom programa. Formatiranje i vizuelni elementi programa treba da olakšaju i proces pisanja programa. U tome, pomoć autoru programa mogu da pružaju alati u okviru kojih se piše program – specijalizovani editori teksta ili editori koji su deo integrisanih razvojnih okruženja (engl. IDE, Integrated Development Environment) koja povezuju editor, kompilator, debager i druge alatke potrebne u razvoju softvera. Neke od namenskih alatki koji olakšavaju pisanje programa su: „ulepšivači“ (engl. beautifier), poput programa `indent`, koji mogu da formatiraju već kreirane datoteke sa programskim kodom; programi za proveru pravopisa, koji mogu da otkriju jednostavne leksičke i sintaksičke greške u programu i da nude moguće ispravke; „linteri“, programi koji vrše statičku analizu programa i koji mogu da ukažu na određene stilske (ali i ozbiljnije) greške, itd.

1.2.1 Broj karaktera u redu

U modernim programskim jezicima dužina reda programa nije ograničena.[1] Ipak, predugi redovi mogu da stvaraju probleme. Na primer, predugi redovi mogu da zahtevaju horizontalno „skrolovanje“ kako bi se video njihov kraj, što može da drastično oteža čitanje i razumevanje programa. Takođe, ukoliko se program štampa, dugi redovi mogu da budu presečeni i da naruše formatiranje. Zbog ovih i ovakvih problema, preporučuje se pridržavanje nekog ograničenja – obično 80 karaktera u redu. Konkretna preporuka za 80 karaktera u redu je istorijska i potiče od ograničenja na starim ekranima i štampačima. Ipak, ona je i danas široko prihvaćena kao pogodna. Ukoliko red programa ima više od 80 karaktera, to najčešće ukazuje na to da kôd treba reorganizovati uvođenjem

[1] Zapravo, često je dužina reda programa ograničena nekim velikim brojem, daleko većim od uobičajenih dužina redova. Na primer, standard C99 propisuje da kompilator mora da bude u stanju da prihvati redove dužine do 4095 karaktera, ne i redove duže od toga.

novih funkcija ili promenljivih. Broj 80 (ili bilo koji drugi) kao ograničenje za broj karaktera u redu ne treba shvatati kruto, već kao načelnu preporuku koja može biti narušena ako se tako postiže bolja čitljivost.

1.2.2 Broj naredbi u redu, zagrade i razmaci

Red programa može da bude prazan ili da sadrži jednu ili više naredbi. Prazni redovi mogu da izdvajaju blokove blisko povezanih naredbi (na primer, blok naredbi za koje se može navesti komentar o tome šta je njihova svrha). Ako se prazni redovi koriste neoprezno, mogu da naruše umesto da poprave čitljivost. Naime, ukoliko ima previše praznih linija, smanjen je deo koda koji se može videti i sagledavati istovremeno na ekranu. Po jednoj konvenciji, zagrade koje označavaju početak i kraj bloka navode se u zasebnim redovima (u istoj koloni), a po drugoj, otvorena zagrada se navodi u nastavku naredbe, a zatvorena u zasebnom redu ili u redu zajedno sa ključnom rečju `while` ili `else`. Torvalds preporučuje ovu drugu konvenciju, uz izuzetak da se otvorena vitičasta zagrada na početku definicije funkcije piše u zasebnom redu.

Naredni primer prikazuje deo koda napisan sa većim brojem praznih redova i prvom konvencijom za zagrade:

```
do
{

  printf("Unesi ceo broj: ");
  scanf("%d", &i);

  if (duzina == alocirano)
  {

    alocirano += KORAK;
    a = realloc(a, alocirano*sizeof(int));
    if (a == NULL)
      return -1;
  }

  a[duzina++] = i;
}
while (i != -1);
```

Isti deo koda može biti napisan sa manjim brojem praznih redova i drugom konvencijom za zagrade. Ovaj primer prikazuje kompaktnije zapisan kôd koji je verovatno čitljiviji većini iskusnih C programera:

```
do {
```

```
    printf("Unesi ceo broj: ");
    scanf("%d", &i);
    if (duzina == alocirano) {
      alocirano += KORAK;
      a = realloc(a, alocirano*sizeof(int));
      if (a == NULL)
        return -1;
    }
    a[duzina++] = i;
  } while (i != -1);
```

Jedan red može da sadrži i više od jedne naredbe. To je prihvatljivo samo (a tada može da bude i preporučljivo) ako se radi o jednostavnim i na neki način povezanim inicijalizacijama ili jednostavnim dodelama vrednosti članovima strukture, na primer:

```
...
int i = 10; double suma = 0;
tacka.x = 0; tacka.y = 0;
```

Ukoliko je u petlji ili u `if` bloku samo jedna naredba, onda nisu neophodne zagrade koje označavaju početak i kraj bloka i mnogi programeri ih ne pišu. Međutim, iako nisu neophodne one mogu olakšati razumevanje koda u kojem postoji višestruka `if` naredba. Dodatno, ukoliko se u blok sa jednom naredbom i bez vitičastih zagrada u nekom trenutku doda druga naredba lako može da se previdi da postaje neophodno navesti i zagrade.

Veličina blokova koda je takođe važna za preglednost, pa je jedna od preporuka da vertikalno rastojanje između otvorene vitičaste zagrade i zatvorene vitičaste zagrade koja joj odgovara ne bude veće od jednog ekrana.

Obično se preporučuje navođenje razmaka oko ključnih reči i oko binarnih operatora, izuzev `.` i `->`. Ne preporučuje se korišćenje razmaka kod poziva funkcija i unarnih operatora, izuzev (eventualno) kod operatora `sizeof` i operatora kastovanja. Ne preporučuje se navođenje nepotrebnih zagrada, posebno u okviru povratne vrednosti. Na primer:

```
if (uslov) {
   *a = -b + c + sizeof (int) + f(x);
   return -1;
}
```

1.2.3 Nazubljivanje teksta programa

Nazubljivanje teksta programa za većinu programskih jezika (uključujući jezike C/C++ i Java) nebitno je kompilatoru, ali je skoro neophodno programeru. Nazubljivanje naglašava strukturu programa i olakšava njegovo razumevanje. Red programa može biti uvučen u odnosu na početnu kolonu za nekoliko blanko

karaktera ili nekoliko tab karaktera. Tab karakter može da se u okviru editora interpretira na različite načine (tj. kao različit broj belina), te je preporučljivo u programu sve tab karaktere zameniti razmacima (za šta u većini editora postoji mogućnost) i čuvati ga u tom obliku. Na taj način, svako će videti program (na ekranu ili odštampan) na isti način.

Ne postoji kruto pravilo za broj karaktera za jedan nivo uvlačenja. Neki programeri koriste 4, a neki 2 – sa motivacijom da u redovima od 80 karaktera može da stane i kôd sa dubokim nivoima. Torvalds, sa druge strane, preporučuje broj 8, jer omogućava bolju preglednost. Za delove programa koji imaju više od tri nivoa nazubljivanja, on kaže da su ionako sporni i zahtevaju prepravku.

1.3 Imenovanje promenljivih i funkcija

Imenovanje promenljivih i funkcija veoma je važno za razumljivost programa i sve je važnije što je program duži. Pravila imenovanja mogu da olakšaju i izbor novih imena tokom pisanja programa. Imena promenljivih i funkcija (pa i datoteka programa) treba da sugerišu njihovu ulogu i tako olakšaju razumevanje programa.

Globalne promenljive, strukture i funkcije treba da imaju opisna imena, potencijalno sačinjena od više reči. U *kamiljoj* notaciji (popularnoj među Java i C++ programerima), imena od više reči zapisuju se tako što svaka nova reč (sem eventualno prve) počinje velikim slovom, na primer, `brojKlijenata`. U notaciji sa podvlakama (popularnoj među C programerima), sve reči imena pišu se malim slovima a reči su razdvojene podvlakama, na primer, `broj_klijenata`. Imena makroa i konstanti pišu se obično svim velikim slovima, a imena globalnih promenljivih počinju velikim slovom.

Lokalne promenljive, a posebno promenljive koje se koriste kao brojači u petljama treba da imaju kratka i jednostavna, a često najbolje, jednoslovna imena – jer se razumljivost lakše postiže sažetošću. Imena za brojače u petljama su često `i`, `j`, `k`, za pokazivače `p` i `q`, a za niske `s` i `t`. Preporuka je i da se lokalne promenljive deklarišu što kasnije u okviru funkcije i u okviru bloka u kojem se koriste (a ne u okviru nekog šireg bloka).

Jedan, delimično šaljiv, savet za imenovanje (i globalnih i lokalnih) promenljivih kaže da broj karaktera u imenu promenljive treba da zavisi od broja linija njenog dosega i to tako da bude proporcionalan logaritmu broja linija njenog dosega.

Za promenljive i funkcije nije dobro koristiti generička imena kao `rezultat`, `izracunaj(...)`, `uradi(...)`, već sugestivnija, kao što su, na primer, `kamata`, `izracunaj_kamatu(...)`, `odstampaj_izvestaj_o_kamati(...)`.

Imena funkcija dobro je da budu bazirana na glagolima, na primer, bolje je `izracunaj_kamatu(...)` nego `kamata(...)` i `get_time(...)` nego `time(...)`. Za funkcije koje vraćaju istinitosnu vrednost, ime treba da sugeriše u kom slučaju se vraća vrednost *tačno*, na primer, bolje je ime `is_prime(...)` nego `check_prime(...)`.

Mnoge promenljive označavaju neki broj entiteta (na primer, broj klijenata, broj studenata, broj artikala) i za njih se može usvojiti konvencija po kojoj imena imaju isti prefiks ili sufiks (na primer, `br_studenata` ili `num_students`). U ovom smislu, znatno dalje ide *mađarska notacija*[2] koja nalaže da početak imena promenljive ukazuje na njen tip (na primer, neka promenljiva tipa `unsigned int` mogla bi da se zove `uBrojKlijenata`). Ova konvencija predmet je mnogih kritika i danas se sve ređe koristi.

I programeri kojima to nije maternji jezik, iako to nije zahtev projekta, često imenuju promenljive i funkcije na osnovu reči engleskog jezika. To je posledica istorijskih razloga i dominacije engleskog jezika u programerskoj praksi, kao i samih ključnih reči skoro svih programskih jezika (koje su na engleskom). Prihvatljivo je (ako nije zahtev projekta drugačiji) imenovanje i na maternjem jeziku i na engleskom jeziku — jedino je neprihvatljivo mešanje ta dva. Imenovanje na bazi engleskog i komentari na engleskom mogu biti pogodni ukoliko postoji i najmanja mogućnost da se izvorni program koristi u drugim zemljama, ili od strane drugih timova, ili da se učini javno dostupnim i slično. Naime, u programiranju (kao i u mnogim drugim oblastima) engleski jezik je opšteprihvaćen u svim delovima sveta i tako se može osigurati da program lakše razumeju svi.

Neki programeri smatraju da se kvalitet imenovanja promenljivih i funkcija može „testirati“ na sledeći zanimljiv način: ako se kôd može pročitati preko telefona tako da ga sagovornik na drugoj strani razume, onda je imenovanje dobro.

1.4 Pisanje izraza

Za dobrog programera neophodno je da poznaje sva pravila programskog jezika jer će verovatno češće i više raditi na tuđem nego na svom kodu. S druge strane, programer u svojim programima ne mora i ne treba da koristi sva sredstva izražavanja tog programskog jezika, već može i treba da ih koristi samo delom, oprezno i uvek sa ciljem pisanja razumljivih programa. Ponekad programer ulaže veliku energiju u pisanje najkonciznijeg mogućeg koda što može da bude protraćen trud, jer je obično važnije da kôd bude jasan, a ne kratak. Sve ovo odnosi se na mnoge aspekte pisanja programa, uključujući pisanje izraza.

Preporučuje se pisanje izraza u jednostavnom i intuitivno jasnom obliku. Na primer, umesto:

```
!(c < '0') && !(c > '9')
```

bolje je:

[2] Mađarsku notaciju uveo je Karolj Simonji, američki informatičar mađarskog porekla. On je u kompaniji Microsoft dugo vodio razvoj nekih od ključnih aplikacija, te je njegova notacija bila postala široko korišćena i u okviru ove kompanije i van nje.

```
'0' <= c && c <= '9'
```

Zagrade, čak i kada nisu neophodne, nekome ipak mogu da olakšaju čitljivost. Prethodni primer može da se zapiše i na sledeći način:

```
('0' <= c) && (c <= '9')
```

Slično, naredbi

```
prestupna = g % 4 == 0 && g % 100 != 0 || g % 400 == 0;
```

ekvivalentna je naredba

```
prestupna = ((g % 4 == 0) && (g % 100 != 0)) || (g % 400 == 0);
```

koja se može smatrati znatno čitljivijom. Naravno, čitljivost je subjektivna, te su moguća i razna međurešenja. Na primer, moguće je podrazumevati da programer jasno razlikuje aritmetičke, relacijske i logičke operatore i da njihov prioritet razlikuje i bez navođenja zagrada, a da se zagrade koriste da bi se naglasila razlika u prioritetu operatora iste vrste (na primer, između logičkih operatora `&&` i `||`). Dodatno, ako ona označava godinu, bolje ime za promenljivu `g` je `godina`, pa se time dolazi do naredbe:

```
prestupna = (godina % 4 == 0 && godina % 100 != 0) ||
            (godina % 400 == 0);
```

Iako je opšta preporuka da se navodi razmak oko binarnih operatora, neke konvencije preporučuju izostavljanje tih razmaka u dužim izrazima i to oko operatora višeg prioriteta čime se zapisom sugeriše prioritet operatora, kao u sledećem primeru:

```
a*b + c*d
```

Suviše komplikovane izraze treba zameniti jednostavnijim i razumljivijim. Na primer, umesto

```
x *= (c +=  a < b ? f("a") : f("b"));
```

možemo koristiti daleko čitljiviju narednu varijantu:

```
if (a < b)
   c += f("a");
else
   c += f("b");
x *= c;
```

Kernigen i Pajk navode i primer u kojem je moguće i poželjno pojednostaviti komplikovana izračunavanja. Ako je potrebno izdvojiti tri bita najmanje težine iz broja `bitoff`, umesto izraza:

```
bitoff - ((bitoff >> 3) << 3)
```

bolje je koristiti (ekvivalentan) izraz:

```
bitoff & 0x7
```

Zbog komplikovanih, a u nekim situacijama i nedefinisanih, pravila poretka izračunavanja i dejstva sporednih efekata (kao, na primer, kod operatora inkrementiranja i dekrementiranja), dobro je pojednostaviti kôd kako bi njegovo izvršavanje bilo jednoznačno i jasno. Na primer, umesto:

```
str[i++] = str[i++] = ' ';
```

bolje je:

```
str[i++] = ' ';
str[i++] = ' ';
```

Poučan je i sledeći čuveni primer: nakon dodele `a[a[1]]=2;`, element `a[a[1]]` nema nužno vrednost 2 (ako je na početku vrednost `a[1]` bila jednaka 1, a vrednost `a[2]` različita od 2). Navedeni primer pokazuje da treba biti veoma oprezan sa korišćenjem indeksa niza koji su i sami elementi niza ili neki komplikovani izrazi.

1.5 Korišćenje idioma

Idiomi su ustaljene jezičke konstrukcije koje predstavljaju celinu. Idiomi postoje u svim jezicima, pa i u programskim. Tipičan idiom u jeziku C je sledeći oblik `for`-petlje:

```
for (i = 0; i < n; i++)
  ...
```

Kernigen i Pajk zagovaraju korišćenje idioma gde god je to moguće. Na primer, umesto varijanti

```
i = 0;
while (i <= n-1)
  a[i++] = 1.0;
```

```
for (i = 0; i < n; )
  a[i++] = 1.0;
```

```
for (i = n; --i >= 0; )
  a[i] = 1.0;
```

smatraju da je bolja varijanta:

```
for (i = 0; i < n; i++)
  a[i] = 1.0;
```

jer je najčešća i najprepoznatljivija. Štaviše, Kernigen i Pajk predlažu, pomalo ekstremno, da se bez dobrog razloga i ne koristi nijedna forma `for`-petlji osim navedene. Kao dodatne primere, navode i idiome za beskonačnu petlju:

```
for (;;)
  ...
```

za prolazak kroz listu (videti poglavlje 6.1):

```
for (p = pocetak; p != NULL; p = p->sledeci)
  ...
```

za učitavanje karaktera sa standardnog ulaza:

```
while ((c = getchar()) != EOF)
  ...
```

Glavni argument za korišćenje idioma je da se kôd brzo razume, a i da svaki drugi („neidiomski“) konstrukt privlači dodatnu pažnju što je dobro, jer se bagovi češće kriju u njima.

1.6 Korišćenje konstanti

Konstantne vrednosti, veličina nizova, pozicije karaktera u niskama, faktori za konverzije i druge slične vrednosti koje se pojavljuju u programima često se zovu *magični brojevi* (jer obično nije jasno odakle dolaze i na osnovu čega su dobijeni). Kernigen i Pajk kažu da je, osim 0 i 1, svaki broj u programu kandidat da se može smatrati magičnim, te da treba da ima ime koje mu je pridruženo. Na taj način, ukoliko je potrebno promeniti vrednost magične konstante (na primer, maksimalna dužina imena ulice) – to je dovoljno uraditi na jednom mestu u kodu. Na primer, u narednoj deklaraciji

```
char imeUlice[50];
```

pojavljuje se magična konstanta 50, te se u nastavku programa broj 50 verovatno pojavljuje u svakoj obradi imena ulica. Promena tog ograničenja zahtevala bi mnoge izmene koje ne bi mogle da se sprovedu automatski (jer se broj 50 možda pojavljuje i u nekom drugom kontekstu). Zato je bolja, na primer, varijanta kojom se magičnom broju pridružuje simboličko ime pretprocesorskom direktivom `#define`:

```
#define MAKS_IME_ULICE 50
char imeUlice[MAKS_IME_ULICE];
```

U tom slučaju, pretprocesor zamenjuje sva pojavljivanja tog imena konkretnom vrednošću pre procesa kompilacije, te kompilator (pa i debager) nema nikakvu informaciju o simboličkom imenu koje je pridruženo magičnoj konstantni niti o njenom tipu. Zbog toga se preporučuje da se magične konstante uvode kao konstantne promenljive, ako upotreba to dozvoljava:

```
const unsigned int MAKS_IME_ULICE = 50;
```

Naglasimo da neke upotrebe ne dozvoljavaju korišćenje konstantne promenljive umesto konstantnog izraza — naime, konstantne promenljive ne smatraju se konstantnim izrazima, te se, na primer, ne mogu koristiti za dimenzije nizova[3]. Kao dimenzije nizova, mogu se, pored konstanti, konstantnih izraza i simboličkih imena uvedenih direktivnom `#define`, koristiti i nabrojive (enumerisane) konstante.

Postoji preporuka da se rezultati funkcije vraćaju kroz listu argumenata, a da povratna vrednost ukazuju na to da li je funkcija uspešno obavila zadatak. Za povratne vrednosti onda postoje dve česte konvencije: jedna je da se vraća istinitosna vrednost *tačno* (`true`, ako se koristi tip `bool` ili `1`, ako je povratni tip ceo broj), ako je funkcija uspešno obavila zadatak, a *netačno* (`false` ili `0`) inače. Druga konvencija je da se vraća nešto detaljnija informacija, te da se vraća `0` ako je izvršavanje funkcije proteklo bez problema, a nekakav celobrojni kôd greške inače. Kodovi greške nikako ne treba da budu magične konstante, već mogu biti predstavljene simboličkim imenima ili, još bolje, enumerisanim konstantama.

U većim programima, konstante od značaja za čitav program (ili veliki njegov deo) obično se čuvaju u zasebnoj datoteci zaglavlja (koju koriste sve druge datoteke kojima su ove konstante potrebne).

Konstante se u programima mogu koristiti i za kodove karaktera. To je loše ne samo zbog narušene čitljivosti, već i zbog narušene prenosivosti – naime, nije na svim računarima podrazumevana ASCII karakterska tabela. Dakle, umesto, na primer:

[3] Standard C99 uvodi pojam niza promenljive dužine (engl. variable length array, VLA): dimenziju takvog niza može da određuje promenljiva čak i ako nije konstantna. U kasnijim standardima jezika C, podrška za ovu mogućnost proglašena je za opcionu, te je u ovoj knjizi ne razmatramo.

```
if (65 <= c && c <= 90)
  ...
```

bolje je pisati

```
if ('A' <= c && c <= 'Z')
  ...
```

a još bolje koristiti funkcije iz standardne biblioteke, kad god je to moguće:

```
if (isupper(c))
  ...
```

Slično, zarad bolje čitljivosti treba pisati `NULL` (za nultu vrednost pokazivača) i `'\0'` (za završnu nulu u niskama) umesto konstante `0`.

U programima ne treba koristiti kao konstante ni veličine tipova – zbog čitljivosti a i zbog toga što se mogu razlikovati na različitim računarima. Zato, na primer, za dužinu tipa `int` nikada ne treba pisati 2 ili 4, već `sizeof(int)`. Za promenljive i elemente niza, bolje je pisati `sizeof(a)` i `sizeof(b[0])` umesto `sizeof(int)` (ako su promenljiva `a` i niz `b` tipa `int`), zbog mogućnosti da se promenljivoj ili nizu u nekoj verziji programa promeni tip.

1.7 Korišćenje makroa sa argumentima

Makroe sa argumentima obrađuje pretprocesor i u fazi izvršavanja, za razliku od funkcija, nema kreiranja stek okvira, prenosa argumenata i sličnih koraka. Zbog uštede memorije i računarskog vremena, makroi sa argumentima nekada su smatrani poželjnom alternativom funkcijama. Danas, u svetu mnogo bržih računara nego nekada, smatra se da loše strane makroa sa argumentima prevazilaze njihove dobre strane i da korišćenje makroa treba izbegavati. U loše strane makroa sa argumentima spada to što ih obrađuje pretprocesor (a ne kompilator) te nema provera tipova argumenata, debager ne može da prati definiciju makroa, lako se može napraviti greška u definiciji makroa zbog izostavljenih zagrada, lako se može napraviti greška zbog koje se neki argument izračunava više puta, itd. (više o makroima sa argumentima može se pročitati u prvom delu ove knjige, u poglavlju 9.2, „Organizacija izvornog programa“).

1.8 Pisanje komentara

Čak i ako se autor pridržavao mnogih preporuka za pisanje jasnog i kvalitetnog koda, ukoliko kôd nije dobro komentarisan njegovo razumevanje može i samom autoru predstavljati teškoću već nekoliko nedelja nakon pisanja. Komentari treba da olakšaju razumevanje koda i predstavljaju njegov svojevrsni dodatak.

Postoje alati koji olakšavaju kreiranje dokumentacije na osnovu komentara u samom kodu i delom je generišu automatski (na primer, Doxygen).

Komentari ne treba da objašnjavaju ono što je očigledno: Komentari ne treba da govore *kako* kôd radi, već *šta* radi (i zašto). Na primer, naredna dva komentara su potpuno suvišna:

```
k += 1.0;  /* k se uvecava za 1.0 */
```

```
return OK; /* vrati OK */
```

U prvom slučaju, komentar ima smisla ako objašnjava zašto se nešto radi, na primer:

```
k += 1.0;  /* u ovom slucaju, kao bonus, kamatna stopa
              uvecava se za 1.0 */
```

U slučajevima da je neki deo programa veoma komplikovan, potrebno je u komentaru objasniti zašto je komplikovan, kako radi i zašto je izabrano takvo rešenje.

Komentari treba da budu koncizni. Kako ne bi trošili preterano vreme, komentari treba da budu što je moguće kraći i jasniji, da ne ponavljaju informacije koje su već navedene drugde u komentarima ili su očigledne iz koda. Previše komentara ili predugi komentari predstavljaju opasnost za čitljivost programa.

Komentari treba da budu usklađeni sa kodom. Ako se promeni kôd programa, a ne i prateći komentari, to može da uzrokuje mnoge probleme i nepotrebne izmene u programu u budućnosti. Ukoliko se neki deo programa promeni, uvek je potrebno proveriti da li je novo ponašanje u skladu sa komentarima (za taj ali i druge delove programa). Usklađenost koda i komentara je lakše postići ako komentari ne govore ono što je očigledno iz koda.

Komentarima treba objasniti ulogu datoteka i globalnih objekata. Komentarima treba, na jednom mestu, tamo gde su definisani, objasniti ulogu datoteka, globalnih objekata kao što su funkcije, globalne promenljive i strukture. Funkcije treba komentarisati pre same definicije, a Torvalds čak savetuje da se izbegavaju komentari unutar tela funkcije. Čitava funkcija može da zaslužuje komentar (pre prvog reda), ali ako pojedini njeni delovi zahtevaju komentarisanje, onda je moguće da funkciju treba reorganizovati i/ili podeliti na nekoliko funkcija. Ni ovo pravilo nije kruto i u specifičnim situacijama prihvatljivo je komentarisanje delikatnih delova funkcije („posebno pametnih ili ružnih“).

Loš kôd ne treba komentarisati, već ga popraviti. Često kvalitetno komentarisanje *kako* i *zašto* neki loš kôd radi zahteva više truda nego pisanje tog dela koda iznova tako da je očigledno kako i zašto on radi.

Komentari treba da budu laki za održavanje: Treba izbegavati stil pisanja komentara u kojem i mala izmena komentara zahteva dodatni posao u formatiranju. Na primer, promena narednog opisa funkcije zahteva izmene u tri reda komentara:

```
/*************************************
* Funkcija area racuna povrsinu trougla *
*************************************/
```

Komentari mogu da uključuju standardne fraze. S vremenom se nametnulo nekoliko oznaka („markera“) na bazi fraza koje se često pojavljuju u okviru komentara. Njih je lako pronaći u kodu, a mnoga razvojna okruženja prepoznaju ih i prikazuju u istaknutoj boji kako bi privukli pažnju programera kao svojevrsna lista stvari koje treba obaviti. Najčešći markeri su:

TODO marker: označava zadatke koje tek treba obaviti, koji kôd treba napisati.

FIXME marker: označava deo koda koji radi ali treba ga popraviti, u smislu opštijeg rešenja, lakšeg održavanja, ili bolje efikasnosti.

HACK marker: označava deo koda u kojem je, kako bi radio, primenjen neki trik i loše rešenje koje se ne prihvata kao trajno, te je potrebno popraviti ga.

BUG marker: označava deo koda koji je gotov i očekuje se da radi, ali je pronađen bag.

XXX marker: obično označava komentar programera za sebe lično i treba biti obrisan pre nego što kôd bude isporučen drugima. Ovim markerom se obično označava neko problematično ili nejasno mesto u kodu ili pitanje programera.

Uz navedene markere i prateći tekst, često se navodi i ime onoga ko je uneo komentar, kao i datum unošenja komentara.

1.9 Modularnost

Veliki program je teško ili nemoguće razmatrati ako nije podeljen na celine. Podela programa na celine (na primer, datoteke i funkcije) neophodna je za razumevanje programa i nametnula se veoma rano u istoriji programiranja. Svi savremeni programski jezici su dizajnirani tako da je podela na manje celine ne samo moguća već tipičan način podele određuje sam stil programiranja

(na primer, u objektno orijentisanim jezicima neki podaci i metode za njihovu obradu se grupišu u takozvane klase). Podela programa na module treba da omogući:

Razumljivost: podela programa na celine popravlja njegovu čitljivost i omogućava onome ko piše i onome ko čita program da se usredsredi na ključna pitanja jednog modula, zanemarujući u tom trenutku i iz te perspektive sporedne funkcionalnosti podržane drugim modulima.

Upotrebljivost: ukoliko je kôd kvalitetno podeljen na celine, pojedine celine biće moguće upotrebiti u nekom drugom kontekstu. Na primer, proveravanje da li neki trinaestocifreni kôd predstavlja mogući JMBG (jedinstveni matični broj građana) može se izdvojiti u zasebnu funkciju koja je onda upotrebljiva u različitim programima.

Obično se program ne deli u funkcije i onda u datoteke tek onda kada je kompletno završen. Naprotiv, podela programa u dodatke i funkcije vrši se u fazi pisanja programa i predstavlja jedan od njegovih najvažnijih aspekata.

1.9.1 Modularnost i podela na funkcije

Za većinu jezika osnovna je funkcionalna dekompozicija ili podela na funkcije. U principu, funkcije treba da obavljaju samo jedan zadatak i da budu kratke. Tekst jedne funkcije treba da staje na jedan ili dva ekrana (tj. da ima manje od pedesetak redova), radi dobre preglednosti. Duge funkcije poželjno je podeliti na manje funkcije, na primer, na one koje obrađuju specijalne slučajeve. Ukoliko je brzina izvršavanja kritična, kompilatoru se može naložiti da *inlajnuje* funkcije (da prilikom kompilacije umetne kôd kratkih funkcija na pozicije gde su pozvane)[4].

Da li funkcija ima razuman obim često govori broj lokalnih promenljivih: ako ih ima više od, na primer, 10, verovatno je funkciju poželjno podeliti na nekoliko manjih. Slično važi i za broj parametara funkcije.

1.9.2 Modularnost i podela na datoteke

Veliki programi sastoje se od velikog broja datoteka koje bi trebalo da budu organizovane na razuman način u direktorijume. Jednu datoteku treba da čine definicije funkcija koje su međusobno povezane i predstavljaju nekakvu celinu.

Datoteke zaglavlja obično imaju sledeću strukturu:

- definicije tipova;
- definicije konstanti;

[4] Inlajnovanje u nekim situacijama kompilatori primenjuju i bez eksplicitnog zahteva programera.

- deklaracije globalnih promenljivih (uz navođenje kvalifikatora `extern`);
- deklaracije funkcija (uz navođenje kvalifikatora `extern`).

a izvorne datoteke sledeću strukturu:

- uključivanje sistemskih datoteka zaglavlja;
- uključivanje lokalnih datoteka zaglavlja;
- definicije tipova;
- definicije konstanti;
- deklaracije/definicije globalnih promenljivih;
- definicije funkcija.

Organizacija u datoteke treba da bude takva da izoluje delove koji se trenutno menjaju i razdvoji ih od delova koji su stabilne, zaokružene celine. U fazi razvoja, često je preporučljivo čak i napraviti privremeni deo organizacije kako bi modul koji se trenutno razvija bio izolovan i razdvojen od drugih delova.

Program treba deliti na datoteke imajući u vidu delom suprotstavljene zahteve. Jedna datoteka ne treba da bude duža od nekoliko, na primer - dve ili tri, stotine linija. Ukoliko logička struktura programa nameće dužu datoteku, onda vredi preispitati postojeću organizaciju podataka i funkcija. S druge strane, datoteke ne treba da budu prekratke i treba da predstavljaju zaokružene celine. Preterana usitnjenost (u preveliki broj datoteka) može da oteža upravljanje programom i njegovu razumljivost.

Integrisana razvojna okruženja i program `make` (videti prvi deo ove knjige, poglavlje 9.1, „Od izvornog do izvršivog programa“) značajno olakšavaju rad sa programima koji su sačinjeni od više datoteka.

1.10 Upravljanje greškama

Svaka od funkcija koje čine program ima neki specifičan zadatak. Generalno se može očekivati da će taj zadatak biti uspešno obavljen ali postoje mnogi scenariji gde to i nije tako. Na primer,

- ako se tokom izvršavanja funkcije dogodi celobrojno deljenje nulom – doći će do greške i prekida izvršavanja programa;
- ako su neki podaci poslati na štampu a štampač nije raspoloživ – doći će do greške i prekida izvršavanja programa;
- ako je prekoračena predviđena veličina programskog steka – doći će do greške i prekida izvršavanja programa;

- ako se pristupa oslobođenoj memoriji na hipu – može doći do greške i prekida izvršavanja programa;

- ako se upisuje sadržaj u neki niz nakon njegove granice – može doći do greške i prekida izvršavanja programa.

U nekim situacijama, još neugodnije, program se ne prekida, nego nastavlja sa radom dajući pogrešne rezultate (to su najčešće mesta gde standard jezika ostavlja nedefinisano ponašanje).

Neke od ovih grešaka moguće je i potrebno predupredi. U tu svrhu funkcije umesto da vraćaju samo rezultat svog rada, mogu da vraćaju (kroz povratnu vrednost, listu argumenata ili na neki drugi način) i nekakvu informaciju o tome da li je zadatak obavljen uspešno (često tu informaciju zovemo „status“). Određivanje, prenos i korišćenje takvih informacija zovemo *upravljanje greškama* (eng. *error handling*).

Generalno, program može da obrađuje i situacije koje logički ne bi smele da se dogode. Time se delovi programa štite od neispravnih ulaza i omogućava nastavak nejgovog izvršavanja i u neočekivanim okolnostima. Ovaj pristup programiranju i upravljanju greškama naziva se *odbrambeno* programiranje. Ipak, nije neophodno, pa ni preporučeno da se pokušava da se sve moguće greške predupredе jer to vodi komplikovanom kodu teškom za razumevanje i održavanje. Naime, za neke funkcije će se *pretpostavljati* da su neki preduslovi tačni i da je funkcija pozvana na predviđeni način (na primer, u funkcijama koje vrše binarnu pretragu ne proverava se da su elementi niza zaista sortirani – dužnost onoga ko poziva ovu funkciju je da obezbedi da taj preduslov bude ispunjen). Svaki program ima svoju *specifikaciju* kojom se između ostalog definiše dopušten skup ulaznih podataka. Zadatak programera je da obezbedi da program ispravno radi u slučaju kada ulazni podaci zadovoljavaju tu specifikaciju. U slučaju kada ti podaci ne zadovoljavaju specifikaciju, ponašanje programa je nedefinisano jer se ne očekuje da će program biti korišćen sa tim neispravnim ulazima (ispravnost ulaznih podataka je obaveza onoga ko poziva program). Isto važi i za svaku pojedinačnu funkciju. Na primer, funkcija koja vrši binarnu pretragu niza u slučaju kada niz nije sortiran može da vrati bilo koju vrednost. Obično se programi koji se pišu tako da ih programer koristi samostalno ili programi koji obrađuju neke podatke koji su automatski generisani i koji su sigurno ispravni mogu pisati tako da nije potrebno proveravati ispravnost tih ulaznih podataka. Drugim rečima, u mnogim programima prihvatljivo je specifikacijom suziti prostor dopuštenih ulaza i time ga pojednostaviti. S druge strane, programi koji se pišu za širi krug korisnika i programi koji treba da budu robusni i dugotrajni imaju slabije pretpostavke o ispravnosti ulaza i dužnost programera je da obezbedi proveru ispravnosti ulaza i prijavljivanje odgovarajućih grešaka kada ulaz nije ispravan. U svakom slučaju programer mora da ima jasno u vidu specifikaciju problema koji rešava i da svoje programe i funkcije tome prilagodi.

U nekim situacijama, preduslovi programa se ne proveravaju na klasičan način, ali se naglašava da su oni podrazumevani naredbom `assert(preduslov);`.

U režimu debagovanja, ukoliko `preduslov` nije ispunjen kada se dođe do ove naredbe, program će prekinuti rad. To može da pomogne u otklanjanju greške, jer kada je program u realnoj upotrebi (takozvana produkciona verzija, engl. release version), situacija u kojoj `preduslov` nije ispunjen apsolutno ne sme da se dogodi (na primer, u softveru koji upravlja avionom ne sme da se dogodi da je trenutna brzina aviona negativan broj), i to mora da obezbedi i garantuje dizajn programa. U ovom kontekstu, naredba `assert(preduslov);` ima i drugu svrhu: da eksplicitno daje informaciju o podrazumevanom uslovu. Slično kao što možemo zahtevati (i obezbediti dizajnom programa) da se neka funkcija može pozvati samo pod nekim uslovima, tako se može zahtevati (i obezbediti dizajnom programa) da se neki delovi koda jedne funkcije izvršavaju samo pod nekim uslovima, koji sprečavaju neke greške. U takvim situacijama nije potrebno (pa ni poželjno) proveravati da li dolazi do greške koja bi trebalo da je onemogućena dizajnom. Na primer, ako se funkcija binarne pretrage u programu poziva nakon poziva funkcije za sortiranje, niz će sigurno biti sortiran i ne bi imalo nikakvog smisla da program vrši eksplicitnu proveru da li je niz zaista sortiran.

Postoje i druge vrste grešaka i njima treba nekako upravljati. Na primer, treba proveriti da li je uspela dinamička alokacija memorije. U nekim programima se to ne proverava – nekad zbog nemara, nekad zbog pretpostavke da je šansa da dođe do te vrste problema zanemarljivo mala, a nekada zbog toga što i ako se greška otkrije, nema puno načina da se ona ispravi (program mora da prekine sa radom, ne završivši predviđeni posao). Ukoliko se želi moguća greška nekako obraditi, postoji za to nekoliko načina. Kao što je rečeno, funkcije informaciju o tome da li je zadatak obavljen uspešno mogu da vraćaju kroz povratnu vrednost, listu argumenata ili na neki drugi način. U ekstremnom slučaju, kad god se pozove funkcija koja vraća takvu informaciju – treba tu informaciju proveriti ukoliko može da utiče na nastavak izvršavanja programa. Takve provere mogu da znatno opterete kôd programa. Moguće je i da se status o grešci sačuva u nekoj posebnoj promenljivoj (nekoj globalnoj promenljivoj ili nekom posebnom polju strukture u kojoj se čuvaju svi podaci) i da se proverava samo u nekim situacijama (u situacijama u kojima greška može da ima značajan uticaj).

U jezicima koji su razvijeni nakon jezika C postoje bogatiji mehanizmi za upravljanje greškama, kao što je mehanizam *izuzetaka* (eng. exceptions). Programer izdaje posebnu naredbu (obično se naziva `throw`) koja se aktivira u slučaju greške, kojom se prekida kôd koji se trenutno izvršava i tok programa preusmerava se na poseban deo koda koji se bavi obradom grešaka (obično se naziva `catch`). Time se postiže da su normalan tok programa i obrada grešaka fizički razdvojeni u samom kodu, što pojednostavljuje programiranje i čini programe čitljivijim i lakšim za održavanje.

Pitanja i zadaci za vežbu

Pitanje 1.1. *Istraži na internetu koji alati za kontrolu verzija se trenutno najviše koriste i koji alati su se najviše koristili ranije.*

Pitanje 1.2. *Navedi barem dva pogodna imena za promenljivu koja označava ukupan broj klijenata. Navedi barem dva pogodna imena funkcije koja vrši izračunavanje finalne ocene.*

Pitanje 1.3. *Koje ime bi promenljiva* `int broj_cvorova` *imala u kamiljoj notaciji?*

Pitanje 1.4. *Koliko se preporučuje da najviše ima karaktera u jednoj liniji programa i koji su razlozi za to?*

Pitanje 1.5. *Ukoliko neka linija našeg programa ima 300 karaktera, šta nam to sugeriše?*

Pitanje 1.6. *Kada je prihvatljivo da jedna linija programa ima više naredbi?*

Pitanje 1.7. *U kojim situacijama se obično neka linija programa ostavlja praznom?*

Pitanje 1.8. *Od čega zavisi koliko belina zamenjuje jedan tab karakter?*

Pitanje 1.9. *Napisati sledeće naredbe na drugi način:*

```
if ( !(c == 'y' || c == 'Y') )
   return;
```

```
length = (length < BUFSIZE) ? length : BUFSIZE;
```

```
flag = flag ? 0 : 1;
```

Pitanje 1.10. *Šta su to „magične konstante“ i da li one popravljaju ili kvare kvalitet programa? Kako se izbegava korišćenje „magičnih konstanti“ u programu?*

Pitanje 1.11. *Da bi kôd bio lakši za održavanje, šta je bolje koristiti umesto deklaracije* `char ImeKorisnika[50]`*?*

Pitanje 1.12. *Istraži na internetu koji alati za automatsko generisanje tehničke dokumentacije se trenutno najviše koriste i koji alati su se najviše koristili ranije.*

Pitanje 1.13. *Kojim se komentarom/markerom obično označava mesto u kodu na kojem treba dodati kôd za neki podzadatak?*

Koji se marker u okviru komentara u kodu obično koristi za označavanje potencijalnih propusta i/ili grešaka koje naknadno treba ispraviti?

Pitanje 1.14. *Da li postoje opšte preporuke za obim jedne datoteke programa?*

Glava 2

Ispravnost programa

Jedno od centralnih pitanja u razvoju programa je pitanje njegove ispravnosti (korektnosti). Softver je u današnjem svetu prisutan na svakom koraku: softver kontroliše mnogo toga — od bankovnih računa i komponenti televizora i automobila, do nuklearnih elektrana, aviona i svemirskih letelica. U svom tom softveru neminovno su prisutne i greške. Greška u funkcionisanju daljinskog upravljača za televizor može biti tek uznemirujuća, ali greška u funkcionisanju nuklearne elektrane može imati razorne posledice. Najopasnije greške su one koje mogu da dovedu do velikih troškova, ili još gore, do gubitka ljudskih života. Neke od katastrofa koje su opštepoznate su eksplozija rakete *Ariane (fr. Ariane 5)* 1996. uzrokovana konverzijom broja iz šezdesetčetvorobitnog realnog u šesnaestobitni celobrojni zapis koja je dovela do prekoračenja, zatim greška u numeričkom koprocesoru procesora Pentium 1994. uzrokovana pogrešnim indeksima u `for` petlji u okviru softvera koji je radio dizajn čipa, kao i pad orbitera poslatog na Mars 1999. uzrokovan činjenicom da je deo softvera koristio metričke, a deo softvera engleske jedinice. Međutim, fatalne softverske greške i dalje se neprestano javljaju i one koštaju svetsku ekonomiju milijarde dolara. Evo nekih od najzanimljivijih:

- „Internet crv“[1] Moris (eng. Morris) raširio se, koristeći propuste i greške u nekoliko sistemskih programa, putem interneta 1988. godine (kao jedan od prvih takvih programa) i privukao pažnju mnogih svetskih medija. Autor crva, student Robert Moris, nije nameravao da crv bude destruktivan, već samo da se replicira i širi preko mreže, ali stepen replikacije bio je takav da su se računari „inficirali“ mnogo puta, do nivoa da više nisu mogli da funkcionišu. Na taj način, nekoliko hiljada računara prestalo je sa normalnim funkionisanjem i bilo je neoperativno nekoliko dana. Ukupna šteta je u to vreme procenjena na nekoliko miliona dolara. Suđenje au-

[1] Osnovna razlika između računarskog virusa i računarskog crva je u tome što virus mora biti aktiviran nekakvom akcijom na računaru na kojem se nalazi. S druge strane, crv je samostalni program koji može da se replicira i širi čim dopre do nekog računarskog sistema.

toru crva bilo je jedno od prvih takvih, zatvorska kazna je na kraju bila uslovna, uz novčanu kaznu i društveno koristan rad.

- Pad satelita *Kriosat (engl. Cryosat)* 2005. godine koštao je Evropsku Uniju oko 135 miliona evra. Pad je uzrokovan greškom u softveru zbog koje nije na vreme došlo do razdvajanja satelita i rakete koja ga je nosila.

- Više od pet procenata penzionera i primalaca socijalne pomoći u Nemačkoj je privremeno ostalo bez svog novca kada je 2005. godine uveden novi računarski sistem. Greška je nastala zbog toga što je sistem, koji je zahtevao desetocifreni zapis svih brojeva računa, kod starijih računa koji su imali osam ili devet cifara brojeve dopunjavao nulama, ali sa desne umesto sa leve strane kako je trebalo.

- Kompanije Dell i Apple morale su tokom 2006. godine da korisnicima zamene više od pet miliona laptop računara zbog greške u dizajnu baterije kompanije Sony koja je uzrokovala da se nekoliko računara zapali.

- Ne naročito opasan, ali veoma zanimljiv primer greške je greška u programu *Microsoft Excel 2007* koji, zbog greške u algoritmu formatiranja brojeva pre prikazivanja, rezultat izračunavanja izraza $77.1 * 850$ prikazuje kao $100,000$ (iako je interno korektno sačuvan).

- U Los Anđelesu je 14. septembra 2004. godine više od četiristo aviona u blizini aerodroma istovremeno izgubilo vezu sa kontrolom leta. Na sreću, zahvaljujući rezervnoj opremi unutar samih aviona, do nesreće ipak nije došlo. Uzrok gubitka veze bila je greška prekoračenja u brojaču milisekundi u okviru sistema za komunikaciju sa avionima. Da ironija bude veća, ova greška je bila otkrivena ranije, ali pošto je do otkrića došlo kada je već sistem bio isporučen i instaliran na nekoliko aerodroma, njegova jednostavna popravka i zamena nije bila moguća. Umesto toga, preporučeno je da se sistem resetuje svakih 30 dana kako do prekoračenja ne bi došlo. Procedura nije ispoštovana i greška se javila posle tačno 2^{32} milisekundi, odnosno 49.7 dana od uključivanja sistema.

2.1 Osnovni pristupi ispitivanju ispravnosti programa

Postupak pokazivanja da je program ispravan naziva se *verifikovanje programa*. U razvijanju tehnika verifikacije programa, potrebno je najpre precizno formulisati pojam ispravnosti programa. Ispravnost programa počiva na pojmu *specifikacije*. Specifikacija je, neformalno, opis željenog ponašanja programa koji treba napisati. Specifikacija se obično zadaje u terminima *preduslova* tj. uslova koje ulazni parametri programa moraju da zadovolje, kao i *postuslova* tj. uslova koje rezultati izračunavanja moraju da zadovolje. Kada je poznata specifikacija, potrebno je verifikovati program, tj. dokazati da on zadovoljava specifikaciju. Dva osnovna pristupa verifikaciji su:

dinamička verifikacija koja podrazumeva proveru ispravnosti u fazi izvršavanja programa, najčešće putem testiranja;

statička verifikacija koja podrazumeva analizu izvornog koda programa, često korišćenjem formalnih metoda i matematičkog aparata.

U okviru verifikacije programa, veoma važno pitanje je pitanje zaustavljanja programa. *Parcijalna korektnost* podrazumeva da neki program, ukoliko se zaustavi, daje korektan rezultat (tj. rezultat koji zadovoljava specifikaciju). *Totalna korektnost* podrazumeva da se program za sve (specifikacijom dopuštene) ulaze zaustavlja, kao i da su dobijeni rezultati parcijalno korektni.

2.2 Dinamičko verifikovanje programa

Dinamičko verifikovanje programa podrazumeva proveravanje ispravnosti u fazi izvršavanja programa. Najčešći vid dinamičkog verifikovanja programa je testiranje.

2.2.1 Testiranje

Najznačajnija vrsta dinamičkog ispitivanja ispravnosti programa je testiranje. Testiranje može da obezbedi visok stepen pouzdanosti programa. Neka tvrđenja o programu je moguće testirati, dok neka nije. Na primer, tvrđenje „program ima prosečno vreme izvršavanja 0.5 sekundi“ je (u principu) proverivo testovima, pa čak i tvrđenje „prosečno vreme između dva pada programa je najmanje 8 sati sa verovatnoćom 95%“. Međutim, tvrđenje „prosečno vreme izvršavanja programa je dobro“ suviše je neodređeno da bi moglo da bude testirano. Primetimo da je, na primer, tvrđenje „prosečno vreme između dva pada programa je najmanje 8 godina sa verovatnoćom 95%“ u principu proverivo testovima ali nije praktično izvodivo.

U idealnom slučaju, treba sprovesti iscrpno testiranje rada programa za sve moguće ulazne vrednosti i proveriti da li izlazne vrednosti zadovoljavaju specifikaciju. Međutim, ovakav iscrpan pristup testiranju skoro nikada nije praktično primenljiv. Na primer, iscrpno testiranje korektnosti programa koji sabira dva 32-bitna broja, zahtevalo bi ukupno $2^{32} \cdot 2^{32} = 2^{64}$ različitih testova. Pod pretpostavkom da svaki test traje jednu nanosekundu, iscrpno testiranje bi zahtevalo približno $1.8 \cdot 10^{10}$ sekundi što je oko 570 godina. Dakle, testiranjem nije praktično moguće dokazati ispravnost netrivijalnih programa. S druge strane, testiranjem je moguće dokazati da program nije ispravan tj. pronaći greške u programima.

S obzirom na to da iscrpno testiranje nije praktično primenljivo, obično se koristi tehnika testiranja tipičnih ulaza programa kao i specijalnih, karakterističnih ulaznih vrednosti za koje postoji veća verovatnoća da dovedu do neke greške. U slučaju pomenutog programa za sabiranje, tipični slučaj bi se odnosio na testiranje korektnosti sabiranja nekoliko slučajno odabranih parova brojeva,

dok bi za specijalne slučajeve mogli biti proglašeni slučajevi kada je neki od sabiraka 0, 1, -1, najmanji negativan broj, najveći pozitivan broj i slično.

Postoje različite metode testiranja, a neke od njih su:

Testiranje zasebnih jedinica (engl. unit testing) U ovom metodu testiranja, nezavisno se testovima proverava ispravnost zasebnih jedinica koda. „Jedinica“ je obično najmanji deo programa koji se može testirati. U proceduralnom jeziku kao što je C, „jedinica“ je obično jedna funkcija. Svaki *jedinični test* treba da bude nezavisan od ostalih, ali puno jediničnih testova može da bude grupisano u baterije testova, u jednoj ili više funkcija sa ovom namenom. Jedinični testovi treba da proveravaju ponašanje funkcije, za tipične, granične i specijalne slučajeve. Ova metoda veoma je važna u obezbeđivanju veće pouzdanosti kada se mnoge funkcije u programu često menjaju i zavise jedna od drugih. Kad god se promeni željeno ponašanje neke funkcije, potrebno je ažurirati odgovarajuće jedinične testove. Ova metoda veoma je korisna zbog toga što često otkriva trivijalne greške, ali i zbog toga što jedinični testovi predstavljaju svojevrsnu specifikaciju.

Postoje specijalizovani softverski alati i biblioteke koje omogućavaju jednostavno kreiranje i održavanje ovakvih testova. *Jedinične testove* obično pišu i koriste, u toku razvoja softvera, sami autori programa ili testeri koji imaju pristup kodu.

Regresiono testiranje (engl. regression testing) U ovom pristupu, proveravaju se izmene programa kako bi se utvrdilo da se nova verzija ponaša isto kao stara (na primer, generiše se isti izlaz). Za svaki deo programa implementiraju se testovi koji proveravaju njegovo ponašanje. Pre nego što se napravi nova verzija programa, ona mora da uspešno prođe sve stare testove kako bi se osiguralo da ono što je ranije radilo radi i dalje, tj. da nije narušena ranija funkcionalnost programa.

Regresiono testiranje primenjuje se u okviru samog implementiranja softvera i obično ga sprovode testeri.

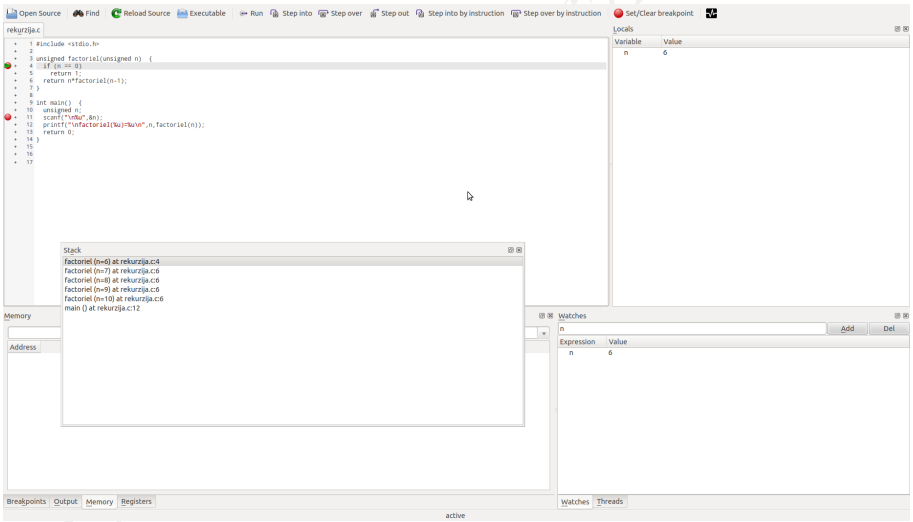
Integraciono testiranje (engl. integration testing) Ovaj vid testiranja primenjuje se kada se više programskih modula objedinjuje u jednu celinu i kada je potrebno proveriti kako funkcioniše ta celina i komunikacija između njenih modula. Integraciono testiranje obično se sprovodi nakon što su pojedinačni moduli prošli kroz druge vidove testiranja. Kada nova programska celina, sastavljena od više modula uspešno prođe kroz integraciono testiranje, onda ona može da bude jedna od komponenti celine na višem nivou koja takođe treba da prođe integraciono testiranje.

Testiranje valjanosti (engl. validation testing) Testiranje valjanosti treba da utvrdi da sistem ispunjava zadate zahteve i izvršava funkcije za koje je namenjen. Testiranje valjanosti vrši se na kraju razvojnog procesa, nakon što su uspešno završene druge procedure testiranja i utvrđivanja

ispravnosti. Testovi valjanosti koji se sprovode su testovi visokog nivoa koji treba da pokažu da se u obradama koriste odgovarajući podaci i u skladu sa odgovarajućim procedurama, opisanim u specifikaciji programa.

2.2.2 Debagovanje

Pojednostavljeno rečeno, testiranje je proces proveravanja ispravnosti programa, sistematičan pokušaj da se u programu (za koji se pretpostavlja da je ispravan) pronađe greška. S druge strane, debagovanje se primenjuje kada se zna da program ima grešku. Debager je alat za praćenje izvršavanja programa radi otkrivanja konkretne greške (baga, engl. bug). To je program napravljen da olakša detektovanje, lociranje i ispravljanje grešaka u drugom programu. On omogućava programeru da ide korak po korak kroz izvršavanje programa, prati vrednosti promenljivih, stanje programskog steka, sadržaj memorije i druge elemente programa.



Slika 2.1: Ilustracija rada debagera `kdbg`

Slika 2.1 ilustruje rad debagera `kdbg`. Uvidom u prikazane podatke, programer može da uoči traženu grešku u programu. Da bi se program debagovao, potrebno je da bude preveden za *debag* režim izvršavanja. Za to se, u kompilatoru `gcc` koristi opcija `-g`. Ako je izvršivi program `mojprogram` dobijen na taj način, može se debagovati navođenjem naredbe:

```
kdbg mojprogram
```

2.2.3 Otkrivanje curenja memorije

Jedna od posledica neispravnosti programa može da bude i curenje memorije. Curenje memorije je problem koji je često teško primetiti (sve dok ima memorije na raspolaganju) i locirati u izvornom kodu. Postoji više programa koji mogu pomoći u proveravanju da li u programu postoji curenje memorije i u lociranju mesta u programu koje je odgovorno za to. Jedan od takvih programa je `valgrind` (videti i poglavlje 3.1.3) koji ima alatku `memcheck` sa ovom svrhom.

Razmotrimo sledeći jednostavan program.

Program 2.1.

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

int main()
{
  int *p;
  p = malloc(1000);
  if (p == NULL)
    return -1;
  p = malloc(1000);
  if (p == NULL)
    return -1;
  else
    free(p);
  return 0;
}
```

Drugim pozivom funkcije `malloc` biće obrisana prethodna vrednost pokazivača `p`, čime će biti izgubljena mogućnost pristupanja bloku koji je alociran prvim pozivom funkcije `malloc`, tj. doći će do curenja memorije.

Ukoliko je navedeni program kompiliran u *debag* modu i ukoliko je izvršiva verzija nazvana `mojprogram`, alat `valgrind` može se, za detektovanje curenja memorije, pozvati na sledeći način:

```
valgrind --tool=memcheck --leak-check=yes ./mojprogram
```

Curenje memorije će biti uspešno otkriveno i `valgrind` daje sledeći izlaz:

```
==9697== Memcheck, a memory error detector
==9697== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==9697== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==9697== Command: ./mojprogram
==9697==
==9697==
==9697== HEAP SUMMARY:
==9697==     in use at exit: 1,000 bytes in 1 blocks
==9697==   total heap usage: 2 allocs, 1 frees, 2,000 bytes allocated
==9697==
```

```
==9697== 1,000 bytes in 1 blocks are definitely lost in loss record 1 of 1
==9697==    at 0x402BE68: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==9697==    by 0x8048428: main (curenje_memorije.c:6)
==9697==
==9697== LEAK SUMMARY:
==9697==    definitely lost: 1,000 bytes in 1 blocks
==9697==    indirectly lost: 0 bytes in 0 blocks
==9697==      possibly lost: 0 bytes in 0 blocks
==9697==    still reachable: 0 bytes in 0 blocks
==9697==         suppressed: 0 bytes in 0 blocks
==9697==
==9697== For counts of detected and suppressed errors, rerun with: -v
==9697== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Alati za otkrivanje curenja memorije su programerima jako korisni, ali oni ipak nisu svemoćni u otkrivanju curenja memorije. Naime, u procesu traženja curenja memorije prati se samo jedno konkretno izvršavanje programa i na neke naredbe koje su odgovorne za curenje memorije u tom konkretnom izvršavanju možda se uopšte neće naići.

2.3 Statičko ispitivanje ispravnosti programa

Statičko ispitivanje ispravnosti programa, (tj. statička verifikacija) podrazumeva analizu izvornog koda programa (bez njegovog izvršavanja). Takve analize mogu da sprovode i ljudi samostalno („na papiru“), ali ih obično sprovode ljudi uz pomoć namenskih alata ili namenski programi, potpuno automatski.

Proces verifikacije može biti neformalan i formalan. Neformalna verifikacija sprovodi se od strane ljudi, analizom koda, posebno nekih kritičnih delova. Formalna verifikacija zasniva se na precizno definisanoj semantici programskog jezika i strogom logičkom okviru u kojem se dokazi korektnosti izvode. Formalno dokazana ispravnost vodi do najvišeg mogućeg nivoa pouzdanosti programa. Formalno dokazivanje ispravnosti je obično veoma zahtevno, te se ono retko primenjuje, obično samo za bezbednosno kritične programe (kao što je, na primer, program za upravljanje metroom).

2.3.1 Primeri neformalne verifikacije: proveravanje kritičnih mesta u programima

Proveravanje graničnih vrednosti. Najveći broj bagova javlja se na granicama opsega petlje, graničnim indeksima niza, graničnim vrednostima argumenata aritmetičkih operacija i slično. Zbog toga je testiranje na graničnim vrednostima izuzetno važno i program koji prolazi takve ulazne veličine često je ispravan i za druge. U nastavku će biti razmotren primer iz knjige Kernigana i Pajka – kôd koji čita i upisuje u niz karaktere sa standardnog ulaza sve dok ne dođe do kraja reda ili dok ne popuni niz:

```
int i;
char s[MAX];
```

```
for (i = 0; (s[i] = getchar()) != '\n' && i < MAX-1; ++i)
  ;
s[--i] = '\0';
```

Jedna od prvih stvari koje treba proveriti je da li kôd radi u graničnom slučaju i to najjednostavnijem – kada je na ulazu prazan red (tj. red koji sadrži samo karakter `'\n'`). Petlja se zaustavlja u početnoj iteraciji i vrednost `i` neće biti inkrementirana, tj. ostaće jednaka 0, te će u poslednjoj naredbi biti promenjen element `s[-1]`, van granica niza, što je greška.

Ukoliko se navedeni kôd napiše korišćenjem idiomske forme `for` petlje, on postaje:

```
for (i=0; i < MAX-1; i++)
  if ((s[i] = getchar()) == '\n')
    break;
s[i] = '\0';
```

U ovom slučaju, lako se proverava da kôd ispravno radi za početni test. Drugom granicom mogu se smatrati ulazni redovi koji su veoma dugi ili pre čijeg kraja se nalazi kraj toka podataka. Navedeni kôd radi neispravno u ovom drugom slučaju, te je potrebna nova verzija:

```
for (i=0; i < MAX-1; i++)
  if ((s[i] = getchar()) == '\n' || s[i]==EOF)
    break;
s[i] = '\0';
```

Naredni testovi koje treba napraviti za navedeni kôd odnose se na redove koji su dugi koliko i niz `s` ili kraći za jedan karakter, duži za jedan karakter i slično.

Proveravanje pre-uslova. Da bi neko izračunavanje imalo smisla često je potrebno da važe neki pre-uslovi. Na primer, ako je u kodu potrebno izračunati prosečan broj poena n studenata, pitanje je šta raditi ukoliko je n jednako 0 i ponašanje programa treba da bude testirano u takvim situacijama. Jedno moguće ponašanje programa je, na primer, da se ako je n jednako 0, vrati 0 kao rezultat. Drugo moguće ponašanje programa je da se ne dozvoli pozivanje modula za izračunavanje proseka ako je n jednako 0. U ovom drugom pristupu, pre koda za izračunavanje proseka može da se navede naredba `assert(n > 0);` koja će da doprinese daljem testiranju (videti poglavlje 1.10).

Proveravanje povratnih vrednosti. Čest izvor grešaka tokom izvršavanja programa je neproveravanje povratnih vrednosti funkcija kao što su funkcije za alokaciju memorije, za rad sa datotekama itd. Povratne vrednosti

ovih funkcija ukazuju na potencijalni problem i ukoliko se ignorišu – problem će samo postati veći. Opšti savet je da se uvek proverava povratna vrednost ovakvih funkcija.

2.3.2 Formalno ispitivanje ispravnosti programa

Ne postoji algoritam koji za proizvoljan algoritam može da dokaže da zadovoljava svoju specifikaciju (baš kao što ne postoji ni algoritam koji može da ispita da li se proizvoljni program zaustavlja). Najveći problemi u tome su zaustavljanje i analiza petlji. Ono što jeste moguće je postojanje algoritama koji u nekim slučajevima i sa nekim pojednostavljivanjima mogu da dokažu ispravnost zadatog programa.

Jedan od izazova za verifikaciju predstavlja činjenica da se semantika uobičajenih tipova podataka i operacija u programima razlikuje od uobičajene semantike matematičkih operacija nad celim i realnim brojevima (iako velike sličnosti postoje). Na primer, iako tip `int` podseća na skup celih brojeva, a operacija sabiranja dva podatka tipa `int` na sabiranje dva cela broja, razlike su evidentne — domen tipa `int` je konačan, a operacija se vrši „po modulu“ tj. u nekim slučajevima dolazi do prekoračenja. Mnoga pravila koja važe za cele brojeve ne važe za podatke tipa `int`. Na primer, $x \geq 0 \wedge y \geq 0 \Rightarrow x + y \geq 0$ važi ako su x i y celi brojevi, ali ne važi ako su podaci tipa `int`. U nekim slučajevima, prilikom verifikacije ovakve razlike se apstrahuju i zanemaruju. Time se, naravno, gubi potpuno precizna karakterizacija ponašanja programa i „dokazi“ korektnosti prestaju da budu dokazi korektnosti u opštem slučaju. Ipak, ovim se značajno olakšava proces verifikacije i u većini slučajeva ovakve aproksimacije ipak mogu značajno da podignu stepen pouzdanosti programa. U nastavku teksta, ovakve aproksimacije će biti često vršene.

Funkcionalni programi

U principu, dokazivanje korektnosti najjednostavnije je za programe koji su sačinjeni od rekurzivno definisanih funkcija koje ne koriste elemente imperativnog programiranja (kao što su dodele vrednosti promenljivama, petlje i slično). Takve programe moguće je pisati i u jeziku C i zovemo ih funkcionalni programi. Glavni razlog za jednostavnije dokazivanje ispravnosti ovakvih programa je činjenica da se njihove funkcije mogu jednostavno modelovati matematičkim funkcijama (koje za iste argumente uvek daju iste vrednosti). Naime, u funkcionalnim programima funkcije za iste ulazne argumente takođe uvek vraćaju istu vrednost. To je tako zbog pretpostavke da nema eksplicitne dodele vrednosti promenljivama, kao i da kontekst poziva i globalne promenljive ne utiču na izvršavanje funkcija. Dokazivanje korektnosti ovakvih programa teče nekim oblikom matematičke indukcije.

Primer 2.1. Jednostavno se može dokazati da naredna funkcija zaista izračunava faktorijel svog argumenta (pod pretpostavkom da nema prekoračenja):

```
unsigned faktorijel(unsigned n)
{
  if (n == 0)
    return 1;
  else
    return n*faktorijel(n-1);
}
```

Indukcijom pokazujemo da važi $faktorijel(n) = n!$.

Baza indukcije: U slučaju da je $n = 0$, važi da je

$$faktorijel(n) = faktorijel(0) = 1 = 0! \, .$$

Induktivni korak: Pretpostavimo da je n sledbenik nekog broja, tj. da je $n > 0$ i da tvrđenje važi za broj $n - 1$, tj. $faktorijel(n-1) = (n-1)!$. Tada važi

$$faktorijel(n) = n \cdot faktorijel(n-1) = n \cdot (n-1)! = n!.$$

Primer 2.2. Razmotrimo primer funkcije koja vrši množenje svođenjem na sabiranje. Dokažimo korektnost ove funkcije pod pretpostavkom da je $x \geq 0$ i pod pretpostavkom da nema prekoračenja tokom izračunavanja.

```
int mnozi(int x, int y)
{
   if (x == 0)
      return 0;
   else
      return mnozi(x - 1, y) + y;
}
```

Indukcijom pokazujemo da važi $mnozi(x, y) = x \cdot y$.

Baza indukcije: U slučaju da je $x = 0$, važi da je

$$mnozi(x, y) = mnozi(0, y) = 0 = 0 \cdot y \, .$$

Induktivni korak: Pretpostavimo da je x sledbenik nekog broja, tj. da je $x > 0$ i da tvrđenje važi za broj $x - 1$, tj. $mnozi(x-1, y) = (x-1) \cdot y$. Tada važi

$$mnozi(x, y) = mnozi(x-1, y) + y = (x-1) \cdot y + y = x \cdot y \, .$$

Primer 2.3. Stepenovanje broja može biti izraženo rekurzivno.

bazni slučaj: $x^0 = 1$ (tj. za $k = 0$ važi $x^k = 1$)

rekurzivni korak: za $k > 0$ važi: $x^k = x \cdot x^{k-1}$

Vrednost x^k za nenegativne celobrojne izložioce može se jednostavno izračunati sledećom funkcijom (naravno, može se izračunati i iterativno).:

```
float stepen_sporo(float x, unsigned k)
{
  if (k == 0)
    return 1.0f;
  else
    return x * stepen_sporo(x, k - 1);
}
```

Ispravnost navedene funkcije dokazuje se lako, kao u prethodnim primerima. Sledeće rekurzivno rešenje znatno je brže (zahteva mnogo manje množenja):

```
float stepen_brzo(float x, unsigned k)
{
  if (k == 0)
    return 1.0f;
  else if (k % 2 == 0)
    return stepen_brzo(x * x, k / 2);
  else
    return x * stepen_brzo(x, k - 1);
}
```

Dokažimo ispravnost navedene funkcije. Kako je u ovom slučaju funkcija definisana opštom rekurzijom, dokaz će biti zasnovan na totalnoj indukciji i pratiće sledeću shemu: „Da bi se pokazalo da vrednost $stepen\_brzo(x, k)$ zadovoljava neko svojstvo, može se pretpostaviti da za $k \neq 0$ i k parno vrednost $stepen\_brzo(x \cdot x, k/2)$ zadovoljava to svojstvo, kao i da za $k \neq 0$ i k neparno vrednost $stepen\_brzo(x, k-1)$ zadovoljava to svojstvo, i onda tvrđenje treba dokazati pod tim pretpostavkama“. Slične sheme indukcije se mogu dokazati i za druge funkcije definisane opštom rekurzijom i, u principu, one dozvoljavaju da se prilikom dokazivanja korektnosti dodatno pretpostavi da svaki rekurzivni poziv vraća korektan rezultat.

Pređimo na sâm dokaz činjenice da je $stepen\_brzo(x, k) = x^k$ (za nenegativnu vrednost k).

Slučaj $k = 0$: Tada je $stepen\_brzo(x, k) = stepen\_brzo(x, 0) = 1 = x^0$.

Slučaj $k \neq 0$: Tada je k ili paran ili neparan.

> **Slučaj k je paran:** Tada je $stepen\_brzo(x, k) = stepen\_brzo(x \cdot x, k/2)$. Na osnovu prvog dela induktivne pretpostavke, $stepen\_brzo(x \cdot x, k/2) = (x \cdot x)^{k/2}$. Dalje, elementarnim aritmetičkim transformacijama sledi da je $stepen\_brzo(x, k) = (x \cdot x)^{k/2} = (x^2)^{k/2} = x^k$.

Slučaj k je neparan: Tada je $stepen\_brzo(x,k) = x stepen\_brzo(x,k–1)$. Na osnovu drugog dela induktivne pretpostavke, $stepen\_brzo(x,k–1) = x^{k-1}$. Dalje, elementarnim aritmetičkim transformacijama sledi da je $stepen\_brzo(x,k) = x \cdot x^{k-1} = x^k$.

Verifikacija imperativnih programa i invarijante petlji

U slučaju imperativnih programa (programa koji sadrže naredbu dodele i petlje), aparat koji se koristi za dokazivanje korektnosti mora biti znatno složeniji. Semantiku imperativnih konstrukata znatno je teže opisati u odnosu na (jednostavnu jednakosnu) semantiku čisto funkcionalnih programa. Sve vreme dokazivanja mora se imati u vidu tekući kontekst tj. stanje programa koje obuhvata tekuće vrednosti svih promenljivih koje se javljaju u programu. Program implicitno predstavlja relaciju prelaska između stanja i dokazivanje korektnosti zahteva dokazivanje da će program na kraju stići u neko stanje u kojem su zadovoljeni uslovi zadati specifikacijom. Dodatnu otežavajuću okolnost čine propratni efekti dodela, kao i činjenica da pozivi funkcija mogu da vrate različite vrednosti za iste prosleđene ulazne parametre (u zavisnosti od globalnog konteksta u kojima se poziv izvršio). Zbog toga je dokaz korektnosti složenog programa teže razložiti na elementarne dokaze korektnosti pojedinih funkcija.

Kao najkompleksniji programski konstrukt, petlje predstavljaju jedan od najvećih izazova u verifikaciji. Umesto pojedinačnog razmatranja svakog stanja kroz koje se prolazi prilikom izvršavanja petlje, obično se formulišu uslovi (*invarijante petlji*) koji precizno karakterišu taj skup stanja. *Invarijanta petlje* je logička formula koja uključuje vrednosti promenljivih koje se javljaju u nekoj petlji i koja važi pri svakom ispitivanju uslova petlje (tj. neposredno pre, za vreme i neposredno nakon izvršavanja petlje).

Da bi se pokazalo da je neka formula invarijanta petlje, dovoljno je pokazati da (i) tvrđenje važi pre prvog ulaska u petlju i (ii) da tvrđenje ostaje na snazi nakon svakog izvršavanja tela petlje. Iz ova dva uslova, induktivnim rezonovanjem po broju izvršavanja tela petlje, moguće je pokazati da će tada tvrđenje važiti i nakon izvršavanja petlje. Slučaj prvog ulaska u petlju odgovara bazi indukcije, dok slučaj izvršavanja tela petlje odgovara induktivnom koraku.

Svaka petlja ima više invarijanti, pri čemu su neki uslovi „preslabi“ a neki „prejaki“ tj. ne objašnjavaju ponašanje programa. Na primer, bilo koja valjana formula (na primer, $x \cdot 0 = 0$ ili $(x \geq y) \vee (y \geq x)$) je uvek invarijanta petlje. Međutim, da bi se na osnovu invarijante petlje moglo rezonovati o korektnosti programa, potrebno je da invarijanta bude takva da se jednostavno može pokazati iz svojstava programa pre ulaska u petlju, kao i da obezbeđuje željena svojstva programa nakon izlaska iz petlje.

Primer 2.4. Razmotrimo naredni kôd kojim se izračunava vrednost $\mathtt{x}^{\mathtt{k}}$ i smešta se u promenljivu `m`:

```
unsigned i = 0;
float m = 1.0f;
while (i < k) {
  m *= x;
  i++;
}
```

Moguće je pokazati da u svakom koraku važi da je $m = x^i$ te je ovo jedna invarijanta petlje. Dokažimo to:

1. Pokažimo da invarijanta važi pre ulaska u petlju. Pre ulaska u petlju promenljive imaju vrednosti $m = 1$ i $i = 0$, te invarijanta, trivijalno, važi.

2. Pokažimo da invarijanta ostaje održana nakon svakog izvršavanja tela petlje. Pretpostavimo da invarijanta važi za promenljive m i i (promenljive x i k se ne menjaju), tj. da važi $m = x^i$. Nakon izvršavanja tela petlje, promenljive imaju vrednosti $m' = m \cdot x$ i $i' = i + 1$. Potrebno je pokazati da ove nove vrednosti zadovoljavaju invarijantu, tj. da važi $m' = x^{i'}$. Zaista, $m \cdot x = x^{i+1}$ što je tačno na osnovu induktivne pretpostavke. Dakle, $m = x^i$ je invarijanta petlje.

Na kraju, pokažimo da dokazana invarijanta obezbeđuje korektnost. Pošto je k nenegativan (konačan) ceo broj, promenljiva i će u konačnom broju iteracija dostignuti tu vrednost, tj. nakon konačnog broja iteracija izlazi se iz petlje. Kada se izađe iz petlje, važi i je jednako k. Preciznije formulisano, važi i invarijanta $i \leq k$, pa pošto ne važi uslov petlje $i < k$, po izlasku iz petlje mora da važi da je $i = k$. Kombinovanjem sa invarijantom $x = m^i$, dobija se da tada važi $m = x^k$, što je i trebalo dokazati.

Postoji efikasniji rekurzivni, ali i iterativni način da se izračuna x^k (videti primere 2.3 i 4.10).

Primer 2.5. Razmotrimo program koji vrši množenje nenegativnog celog broja x i celog broja y svođenjem na sabiranje.

```
z = 0; n = 0;
while (n < x) {
  z = z + y;
  n = n + 1;
}
```

Nakon n izvršavanja tela petlje promenljiva z je n puta uvećana za vrednost y, pri čemu promenljiva n sadrži upravo broj izvršavanja tela petlje. Dakle, važi da je $z = n \cdot y$. Kako je x nenegativan ceo broj, važi da je sve vreme $n \leq x$. Dokažimo da je formula

$$(n \leq x) \wedge (z = n \cdot y) \, .$$

zaista invarijanta petlje.

1. Pokažimo da invarijanta važi pre ulaska u petlju. Pre ulaska u petlju je $x \geq 0$ (na osnovu preduslova), $z = 0$ i $n = 0$ te invarijanta, trivijalno, važi.

2. Pokažimo da invarijanta ostaje održana nakon svakog izvršavanja tela petlje. Pretpostavimo da invarijanta važi za promenljive z, n, tj. da važi $(n \leq x) \wedge (z = n \cdot y)$. Nakon izvršavanja tela petlje, promenljive imaju vrednosti $z' = z + y$ i $n' = n + 1$. Potrebno je pokazati da ove nove vrednosti zadovoljavaju invarijantu, tj. da važi $n' \leq x \wedge z' = n' \cdot y$. Zaista, pošto je telo petlje izvršavano, važi da je $n < x$. Dakle, $n' = n + 1 \leq x$. Takođe, pošto je $z' = z + y$, a invarijanta je važila za z i n, važi da je $z' = n \cdot y + y$. Dalje, $z' = (n + 1) \cdot y = n' \cdot y$, te je invarijanta zadovoljena i nakon izvršenja tela petlje.

Na kraju, pokažimo da dokazana invarijanta obezbeđuje korektnost. Pošto je x nenegativan (konačan) ceo broj, promenljiva n će u konačnom broju iteracija dostignuti tu vrednost, tj. nakon konačnog broj iteracija izlazi se iz petlje. Kada se izađe iz petlje, uslov nije bio ispunjen tako da važi $n \geq x$. Kombinovanjem sa invarijantom, dobija se da je $n = x$, tj. da je $z = x \cdot y$.

Primer 2.6. Razmotrimo kôd koji uzastopnim oduzimanjem izračunava rezultat celobrojnog deljenja celog broja x celim brojem y.

```
r = x; q = 0;
while (y <= r) {
  r = r - y;
  q = q + 1;
}
```

Moguće je pokazati da u svakom koraku važi da je $x = q \cdot y + r$ te je ovo jedna invarijanta petlje. S obzirom na to da je na eventualnom kraju izvršavanja programa $r < y$, na osnovu ove invarijante direktno sledi korektnost (tj. q sadrži količnik, a r ostatak pri deljenju broja x brojem y).

Primer 2.7. Razmotrimo program koji vrši pronalaženje minimuma niza brojeva a, koji ima n elemenata.

```
m = a[0];
i = 1;
while (i < n) {
  if (a[i] < m)
    m = a[i];
  i++;
}
```

Ovaj program ima smisla samo za neprazne nizove. Invarijanta petlje je da promenljiva m sadrži minimum dela niza a od pozicije 0 do pozicije $i-1$. Nakon završetka petlje, važi da je $i = n$, pa iz invarijante sledi korektnost programa.

Formalni dokazi i Horova logika

Sva dosadašnja razmatranja o korektnosti programa vršena su zapravo poluformalno, tj. nije postojao precizno opisan formalni sistem u kojem se vrši dokazivanje korektnosti imperativnih programa. Jedan od najznačajnijih formalnih sistema ovog tipa opisao je Toni Hor (Tony Hoare).

Formalni dokazi (u jasno preciziranom logičkom okviru) su važni jer mogu da se generišu automatski uz pomoć računara ili barem interaktivno u saradnji čoveka sa računarom. U oba slučaja, formalni dokaz može da se proveri automatski (dok automatska provera neformalnog dokaza nije moguća). Softver čija je ispravnost dokazana i proverena automatski od strane namenskih programa je najpouzdaniji softver i zahtevi za takvim nivoom pouzdanosti postavljaju se za neke bezbednosno kritične aplikacije (kao što je, na primer, kontrolni softver za metro).

Kada se program i dokaz njegove korektnosti istovremeno razvijaju, programer bolje razume sam program i njegova svojstva. Metodologija formalnog ispitivanja ispravnosti utiče i na preciznost, konzistentnost i kompletnost specifikacije, na jasnoću implementacije i sklad implementacije i specifikacije. Zahvaljujući tome dobija se pouzdaniji softver, čak i onda kada se formalni dokaz ne izvede eksplicitno.

Semantika određenog programskog koda može se zapisati trojkom oblika

$$\{\varphi\}\texttt{P}\{\psi\}$$

gde je `P` niz naredbi, a $\{\varphi\}$ i $\{\psi\}$ su logičke formule koje opisuju veze između promenljivih koje se javljaju u tim naredbama. Trojku $(\varphi, \texttt{P}, \psi)$ nazivamo *Horova trojka*. Interpretacija trojke je sledeća: „Ako izvršenje niza naredbi P počinje sa vrednostima ulaznih promenljivih (kažemo i ‚u stanju') koje zadovoljavaju uslov $\{\varphi\}$ i ako P završi rad u konačnom broju koraka, tada vrednosti programskih promenljivih (‚stanje') zadovoljavaju uslov $\{\psi\}$". Uslov $\{\varphi\}$ naziva se *preduslov*, a uslov $\{\psi\}$ naziva se *postuslov (posleuslov)*.

Na primer, trojka $\{x = 1\}$`y:=x`$\{y = 1\}$[2], opisuje dejstvo naredbe dodele i kaže da, ako je vrednost promenljive x bila jednaka 1 pre izvršavanja naredbe dodele, i ako se naredba dodele izvrši, tada će vrednost promenljive y biti jednaka 1. Ova trojka je tačna. S druge strane, trojka $\{x = 1\}$`y:=x`$\{y = 2\}$ govori da će nakon dodele vrednost promenljive y biti jednaka 2 i ona nije tačna.

Formalna specifikacija programa može se zadati u obliku Horove trojke. U tom slučaju preduslov opisuje uslove koji važe za ulazne promenljive, dok postu-

[2]U ovom poglavlju, umesto C-ovske, biće korišćena sintaksa slična sintaksi korišćenoj u originalnom Horovom radu.

slov opisuje uslove koje bi trebalo da zadovolje rezultati izračunavanja. Na primer, program P za množenje brojeva x i y, koji rezultat smešta u promenljivu z bi trebalo da zadovolji trojku $\{\top\}\mathtt{P}\{z = x \cdot y\}$ ($\top$ označava iskaznu konstantu *tačno*, i ovom primeru znači da nema preduslova koje vrednosti x i y treba da zadovoljavaju). Ako se zadovoljimo time da program može da množi samo nenegativne brojeve, specifikacija se može oslabiti u $\{x \geq 0 \wedge y \geq 0\}\mathtt{P}\{z = x \cdot y\}$.

Jedno od ključnih pitanja za verifikaciju je pitanje da li je neka Horova trojka tačna (tj. da li program zadovoljava datu specifikaciju). Hor je dao formalni sistem (aksiome i pravila izvođenja) koji omogućava da se tačnost Horovih trojki dokaže na formalan način (slika 2.2). Za svaku sintaksnu konstrukciju programskog jezika koji se razmatra formiraju se aksiome i pravila izvođenja koji daju rigorozan opis semantike odgovarajućeg konstrukta programskog jezika.[3]

Aksioma dodele (assAx):

$$\{\varphi[x \to E]\}\mathtt{x} := E\{\varphi\}$$

Pravilo posledice (Cons):

$$\frac{\varphi' \Rightarrow \varphi \quad \{\varphi\}\mathtt{P}\{\psi\} \quad \psi \Rightarrow \psi'}{\{\varphi'\}\mathtt{P}\{\psi'\}}$$

Pravilo kompozicije (Comp):

$$\frac{\{\varphi\}\mathtt{P}_1\{\mu\} \quad \{\mu\}\mathtt{P}_2\{\psi\}}{\{\varphi\}\mathtt{P}_1\mathtt{;}\ \mathtt{P}_2\{\psi\}}$$

Pravilo grananja (if):

$$\frac{\{\varphi \wedge c\}\mathtt{P}_1\{\psi\} \quad \{\varphi \wedge \neg c\}\mathtt{P}_2\{\psi\}}{\{\varphi\}\mathtt{if\ c\ then\ P}_1\ \mathtt{else\ P}_2\{\psi\}}$$

Pravilo petlje (while):

$$\frac{\{\varphi \wedge c\}\mathtt{P}\{\varphi\}}{\{\varphi\}\mathtt{while\ c\ do\ P}\{\neg c \wedge \varphi\}}$$

Slika 2.2: Horov formalni sistem

Opis aksiome i pravila Horove logike:

Aksioma dodele Ova aksioma definiše semantiku naredbe dodele. Izraz $\varphi[x \to E]$ označava logičku formulu koja se dobije kada se u formuli φ sva slo-

[3] U svom originalnom radu, Hor je razmatrao veoma jednostavan programski jezik (koji ima samo naredbu dodele, naredbu grananja, jednu petlju i sekvencijalnu kompoziciju naredbi), ali u nizu radova drugih autora Horov originalni sistem je proširen pravilima za složenije konstrukte programskih jezika (funkcije, pokazivače, itd.).

bodna pojavljivanja promenljive x zamene izrazom E. Na primer, jedna od instanci ove sheme je i $\{x+1=2\}y := x+1\{y=2\}$. Zaista, preduslov $x+1=2$ se može dobiti tako što se sva pojavljivanja promenljive kojoj se dodeljuje (u ovom slučaju y) u izrazu postuslova $y=2$ zamene izrazom koji se dodeljuje (u ovom slučaju $x+1$). Nakon primene pravila posledice, moguće je izvesti trojku $\{x=1\}y := x+1\{y=2\}$. Potrebno je naglasiti da se podrazumeva da izvršavanje dodele i sračunavanje izraza na desnoj strani ne proizvodi nikakve propratne efekte do samog efekta dodele (tj. izmene vrednosti promenljive sa leve strane dodele) koji je implicitno i opisan navedenom aksiomom.

Pravilo posledice Ovo pravilo govori da je moguće ojačati preduslov i oslabiti postuslov svake trojke. Na primer, od tačne trojke $\{x=1\}$`y:=x`$\{y=1\}$, moguće je dobiti tačnu trojku $\{x=1\}$`y:=x`$\{y>0\}$. Slično, na primer, ako program `P` zadovoljava $\{x\geq 0\}$`P`$\{z=x\cdot y\}$, tada će zadovoljavati i trojku $\{x\geq 0\wedge y\geq 0\}$`P`$\{z=x\cdot y\}$.

Pravilo kompozicije Pravilom kompozicije opisuje se semantika sekvencijalnog izvršavanja dve naredbe. Kako bi trojka $\{\varphi\}$`P`$_1$`;``P`$_2\{\psi\}$ bila tačna, dovoljno je da postoji formula $\mu$ koja je postuslov programa `P`$_1$ za preduslov $\varphi$ i preduslov programa `P`$_2$ za postuslov $\psi$. Na primer, iz trojki $\{x=1\}$`y := x + 1`$\{y=2\}$ i $\{y=2\}$`z := y + 2`$\{z=4\}$, može da se zaključi $\{x=1\}$`y := x + 1; z := y + 2`$\{z=4\}$.

Pravilo grananja Pravilom grananja definiše se semantika `if-then-else` naredbe. Korektnost ove naredbe se svodi na ispitivanje korektnosti njene `then` grane (uz mogućnost korišćenja uslova grananja u okviru preduslova) i korektnosti njene `else` grane (uz mogućnost korišćenja negiranog uslova grananja u okviru preduslova). Opravdanje za ovo, naravno, dolazi iz činjenice da ukoliko se izvršava `then` grana uslov grananja je ispunjen, dok, ukoliko se izvršava `else` grana, uslov grananja nije ispunjen.

Pravilo petlje Pravilom petlje definiše se semantika `while` naredbe. Uslov φ u okviru pravila predstavlja invarijantu petlje. Kako bi se dokazalo da je invarijanta zadovoljena nakon izvršavanja petlje (pod pretpostavkom da se petlja zaustavlja), dovoljno je pokazati da telo petlje održava invarijantu (uz mogućnost korišćenja uslova ulaska u petlju u okviru preduslova). Opravdanje za ovo je, naravno, činjenica da se telo petlje izvršava samo ako je uslov ulaska u petlju ispunjen.

Dokazi ispravnosti programa u okviru Horove logike koriste instance aksioma (aksiome primenjene na neke konkretne naredbe) i iz njih, primenom pravila, izvode nove zaključke. Dokazi se mogu pogodno prikazati u vidu stabla, u čijem su listovima primene aksioma.

Primer 2.8. Dokažimo u Horovom sistemu da je klasični algoritam razmene (`swap`) vrednosti dve promenljive korektan.

(1)	$\{x = X \wedge y = Y\}$ `t := x` $\{t = X \wedge y = Y\}$	assAx
(2)	$\{t = X \wedge y = Y\}$ `x := y` $\{t = X \wedge x = Y\}$	assAx
(3)	$\{x = X \wedge y = Y\}$ `t := x; x := y` $\{t = X \wedge x = Y\}$	Comp (1) i (2)
(4)	$\{t = X \wedge x = Y\}$ `y := t` $\{y = X \wedge x = Y\}$	assAx
(5)	$\{x = X \wedge y = Y\}$ `t := x; x := y; y := t` $\{y = X \wedge x = Y\}$	Comp (3) i (4)

Primer 2.9. Dokažimo u Horovom sistemu da kôd

```
z = 0; n = 0;
while (n < x) {
  z = z + y;
  n = n + 1;
}
```

vrši množenje brojeva x i y, pod uslovom da su x i y celi brojevi i da je x nenegativan. Izvođenje je dato u tabeli 2.1.

Primer 2.10. Naredna dva primera ilustruju izvođenja u Horovoj logici zapisana u vidu stabla:

$$\dfrac{\dfrac{}{\{x = x\}\ \texttt{y := x}\ \{y = x\}}\ \text{assAx}}{\{\top\}\ \texttt{y := x}\ \{y = x\}}\ \text{Cons} \qquad \dfrac{}{\{y = x\}\ \texttt{z := y}\ \{z = x\}}\ \text{assAx}}{\{\top\}\ \texttt{y := x; z := y}\ \{z = x\}}\ \text{Comp}$$

$$\dfrac{\dfrac{}{\{x \geq x \wedge x \geq y\}\ \texttt{m := x}\ \{m \geq x \wedge m \geq y\}}\ \text{assAx}}{\{\top \wedge x \geq y\}\ \texttt{m := x}\ \{m \geq x \wedge m \geq y\}}\ \text{Cons} \qquad \dfrac{}{\{y \geq y \wedge y > x\}\ \texttt{m := y}\ \{m \geq y \wedge m > x\}}\ \text{assAx}}{\{\top \wedge \neg(x \geq y)\}\ \texttt{m := y}\ \{m \geq x \wedge m \geq y\}}\ \text{Cons}}{\{\top\}\ \texttt{if(x >= y) then m := x; else m := y;}\ \{m \geq y \wedge m \geq x\}}\ \text{if}$$

2.3.3 Ispitivanje zaustavljanja programa

Halting problem je neodlučiv, tj. ne postoji opšti postupak kojim se za proizvoljni zadati program može utvrditi da li se on zaustavlja za zadate vrednosti argumenata (više na ovu temu može se naći u prvom delu ove knjige, u glavi 3, "Algoritmi i izračunljivost"). Ipak, za mnoge konkretne programe, može se utvrditi da li se zaustavljaju ili ne. Kako ne postoji opšti postupak koji bi se primenio na sve programe, zaustavljanje svakog programa mora se ispitivati zasebno i koristeći specifičnosti tog programa.

U programima u kojima su petlje jedine naredbe koje mogu dovesti do nezaustavljanja potrebno je dokazati zaustavljanje svake pojedinačne petlje. Ovo se obično radi tako što se definiše dobro zasnovana relacija[4] takva da su susedna stanja kroz koje se prolazi tokom izvršavanja petlje međusobno u relaciji. Kod elementarnih algoritama ovo se obično radi tako što se izvrši neko preslikavanje skupa stanja u skup prirodnih brojeva i pokaže da se svako susedno stanje preslikava u manji prirodan broj.[5] Pošto je relacija $>$ na skupu

[4] Za relaciju $\succ$ se kaže da je *dobro zasnovana* (engl. well founded) ako ne postoji beskonačan opadajući lanac elemenata $a_1 \succ a_2 \succ \ldots$.

[5] Smatramo da i nula pripada skupu prirodnih brojeva.

(1) $\{x \geq 0 \wedge 0 = 0\}$ `z = 0;` $\{x \geq 0 \wedge z = 0\}$
assAx

(2) $\{x \geq 0\}$ `z = 0;` $\{x \geq 0 \wedge z = 0\}$,
Cons (1) jer $x \geq 0 \Rightarrow x \geq 0 \wedge 0 = 0$

(3) $\{x \geq 0\}$ `z = 0;` $\{x \geq 0 \wedge z = 0 \wedge 0 = 0\}$
Cons (2) jer $x \geq 0 \wedge z = 0 \Rightarrow x \geq 0 \wedge z = 0 \wedge 0 = 0$

(4) $\{x \geq 0 \wedge z = 0 \wedge 0 = 0\}$ `n = 0;` $\{x \geq 0 \wedge z = 0 \wedge n = 0\}$
assAx

(5) $\{x \geq 0\}$ `z = 0; n = 0;` $\{x \geq 0 \wedge z = 0 \wedge n = 0\}$
Comp (3) i (4)

(6) $\{x \geq 0\}$ `z = 0; n = 0;` $\{n \leq x \wedge z = n * y\}$
Cons (5) jer $x \geq 0 \wedge z = 0 \wedge n = 0 \Rightarrow n \leq x \wedge z = n \cdot y$

(7) $\{n \leq x \wedge z + y = (n+1) \cdot y \wedge n < x\}$ `z = z+y;` $\{n \leq x \wedge z = (n+1) \cdot y \wedge n < x\}$
assAx

(8) $\{n \leq x \wedge z = n \cdot y \wedge n < x\}$ `z = z+y;` $\{n \leq x \wedge z = (n+1) \cdot y \wedge n < x\}$
Cons(7) jer $n \leq x \wedge z = n \cdot y \wedge n < x \Rightarrow n \leq x \wedge z + y = (n+1) \cdot y \wedge n < x$

(9) $\{n \leq x \wedge z = n \cdot y \wedge n < x\}$ `z = z+y;` $\{n + 1 \leq x \wedge z = (n+1) \cdot y\}$
Cons(8) jer $n \leq x \wedge z = (n+1) \cdot y \wedge n < x \Rightarrow n + 1 \leq x \wedge z = (n+1) \cdot y$

(10) $\{n + 1 \leq x \wedge z = (n+1) \cdot y\}$ `n = n+1;` $\{n \leq x \wedge z = n \cdot y\}$
assAx

(11) $\{n \leq x \wedge z = n \cdot y \wedge n < x\}$ `z = z+y; n = n+1;` $\{n \leq x \wedge z = n \cdot y\}$
Comp (9) i (10)

(12) $\{n \leq x \wedge z = n \cdot y\}$ `while(n < x) { z = z + y; n = n + 1; }` $\{n \geq x \wedge n \leq x \wedge z = n * y\}$
(while)

(13) $\{n \leq x \wedge z = n \cdot y\}$ `while(n < x) { z = z + y; n = n + 1; }` $\{z = x \cdot y\}$
Cons(12) jer $n \geq x \wedge n \leq x \wedge z = n * y \Rightarrow z = x * y$

(14) $\{x \geq 0\}$ `z = 0; n = 0; while(n < x) {z = z + y; n = n + 1;}` $\{z = x \cdot y\}$
Comp (6) i (13)

Tabela 2.1: Primer dokaza u Horovoj logici

prirodnih brojeva dobro zasnovana, i ovako definisana relacija na skupu stanja biće dobro zasnovana.

Primer 2.11. Algoritam koji vrši množenje uzastopnim sabiranjem se zaustavlja. Zaista, u svakom koraku petlje vrednost $x - n$ je prirodan broj (jer invarijanta kaže da je $n \leq x$). Ova vrednost opada kroz svaki korak petlje (jer se x ne menja, a n raste), pa u jednom trenutku mora da dostigne vrednost 0.

Primer 2.12. Ukoliko se ne zna širina podatka tipa `unsigned int`, nije poznato da li se naredna funkcija zaustavlja za proizvoljnu ulaznu vrednost n:

```
void f(unsigned n)
{
  while (n > 1) {
    if (n % 2)
      n = 3*n+1;
    else
      n = n/2;
  }
}
```

Opšte uverenje je da se funkcija zaustavlja za svaku ulaznu vrednost n (to tvrdi još uvek nepotvrđena *Kolacova (Collatz) hipoteza* iz 1937). Navedeni primer pokazuje kako pitanje zaustavljanja čak i za neke veoma jednostavne programe može da bude ekstremno komplikovano.

Naravno, ukoliko je poznata širina podatka `unsigned int`, i ukoliko se testiranjem za sve moguće ulazne vrednosti pokaže da se `f` zaustavlja, to bi dalo odgovor na pitanje u specijalnom slučaju.

Pitanja i zadaci za vežbu

Pitanje 2.1. *Šta je cilj verifikacije programa?*

Pitanje 2.2. *U čemu je razlika između parcijalne i totalne ispravnosti programa?*

Pitanje 2.3. *Kako se zove provera ispravnosti u fazi izvršavanja programa?*

Pitanje 2.4. *Koji je najčešći vid dinamičke verifikacije programa?*

Pitanje 2.5. *U veb-formularu korisnik popunjava 10 polja i prijavljuje grešku ako je polje pogrešno popunjeno. Pseudo-kodom se takav program može opisati na sledeći način.*

```
if (!ispravno(polje1))
   prijavi_gresku(polje1);
if (!ispravno(polje2))
   prijavi_gresku(polje2);
...
if (!ispravno(polje10))
   prijavi_gresku(polje10);
```

Koliko bi, metodom iscrpnog testiranja, bilo potrebno izvršiti testova da bi se ispitali sve moguće putanje kroz naredbe ovog programa?.

Pitanje 2.6. *Da li se testiranjem programa može*
(a) *dokazati da je program korektan za sve ulaze?*
(b) *opovrgnuti da je program korektan za sve ulaze?*
(c) *dokazati da se program zaustavlja za sve ulaze?*
(d) *dokazati da se program zaustavlja za neke ulaze?*
(e) *dokazati da se program ne zaustavlja za neke ulaze?*
(f) *dokazati da se program ne zaustavlja za sve ulaze?*

Pitanje 2.7. *Da li uz pomoć debagera može da se:*
(a) *efikasnije kompilira program?*
(b) *lakše otkrije greška u programu?*
(c) *izračuna složenost izvršavanja programa?*
(d) *registruje „curenje memorije“ u programu?*
(e) *umanji efekat fragmentisanja memorije?*

Pitanje 2.8. *Proveriti ispravnost narednog koda koji bi trebalo da poredi dva cela broja* `i` *i* `j`*:*

```
if (i > j)
   printf("%d je vece od %d.\n", i, j);
else
   printf("%d je manje od %d.\n", i , j);
```

Pitanje 2.9. *Proveriti ispravnost narednog koda koji bi trebalo da odredi maksimum tri broja.*

```
int max(int a, int b, int c) {
   if (a>b && a>c)
      return a;
   else if (b>a && b>c)
      return b;
   else
      return c;
}
```

Pitanje 2.10. *Proveriti ispravnost narednog koda koji bi trebalo da štampa karaktere niske* `s` *po jedan u redu:*

```
  i=0;
  do {
    putchar(s[i++]);
    putchar('\n');
  } while (s[i] != '\0');
```

Pitanje 2.11. *Kako se dokazuje ispravnost rekurzivnih funkcija?*

Pitanje 2.12. *Dokazati da naredna funkcija vraća vrednost* `x+y` *(pod pretpostavkom da nema prekoračenja):*

```
unsigned f(unsigned x, unsigned y) {
  if (x == 0) return y;
    else return f(x-1, y) + 1;
}
```

Pitanje 2.13. *Dokazati da naredna funkcija vraća vrednost* `x*y` *(pod pretpostavkom da nema prekoračenja):*

```
int mnozi(int x, int y) {
  if (x == 0) return 0;
    else return mnozi(x - 1, y) + y;
}
```

Pitanje 2.14. *Kako se zove logički uslov koji uključuje vrednosti promenljivih koje se javljaju u nekoj petlji i koja važi pri svakom ispitivanju uslova petlje?*

Pitanje 2.15. *Ako je $x \geq 0$ i celobrojno, navesti invarijantu koja obezbeđuje ispravnost sledećeg algoritma sabiranja:*
`z = y; n = 0; while (n < x) { z = z + 1; n = n + 1; }`

Pitanje 2.16. *Ako je $x \geq 0$ i celobrojno, navesti invarijantu koja obezbeđuje ispravnost sledećeg algoritma stepenovanja:*
`z = 1; n = 0; while (n < x) { z = z * y; n = n + 1; }`

Pitanje 2.17. *Kako se interpretira trojka $\{\varphi\}P\{\psi\}$?*

Pitanje 2.18. *Da li je zadovoljena sledeća Horova trojka:*
$\{x > 1\}n := x\{n > 1 \wedge x > 0\}$

Pitanje 2.19. *Dopuniti sledeću Horovu trojku tako da ona bude zadovoljena:*
(a) { _____ } `a:=b; c:=a;` $\{c > 3\}$
(b) { _____ } `n:=x;` $\{n > 3\}$
(c) { _____ } `if (a>b) then z:=a; else z:=b;` $\{z < 9\}$
(d) { _____ } `if (a<b) then z:=a; else z:=b;` $\{z > 0\}$

Pitanje 2.20. *Navesti primer petlje za koju se ne zna da li se zaustavlja.*

Glava 3

Efikasnost programa i složenost izračunavanja

Pored svojstva ispravnosti programa, veoma je važno i pitanje koliko program zahteva vremena (ili izvršenih instrukcija) i memorije za svoje izvršavanje. Često nije dovoljno imati informaciju o tome koliko se neki program izvršava za neke konkretne ulazne vrednosti, već je potrebno imati neku opštiju procenu za proizvoljne ulazne vrednosti. Štaviše, potrebno je imati i opšti način za opisivanje i poređenje *efikasnosti* (ili *složenosti*) različitih algoritama. Obično se razmatraju:

- vremenska složenost algoritma;
- prostorna (memorijska) složenost algoritma.

Vremenska i prostorna složenost mogu se razmatrati

- u terminima konkretnog vremena/prostora utrošenog za neku konkretnu ulaznu veličinu;
- u terminima asimptotskog ponašanja vremena/prostora kada veličina ulaza raste.

U nekim situacijama vremenska ili prostorna složenost programa nisu mnogo važne (ako se zadatak izvršava brzo, ne zahteva mnogo memorije, ima dovoljno raspoloživog vremena itd), ali u nekim je dragocena ušteda svakog sekunda ili bajta. U takvim situacijama (ali i inače), dobro je najpre razviti najjednostavniji program koji obavlja dati zadatak, a onda ga modifikovati ako je potrebno da se uklopi u zadata vremenska ili prostorna ograničenja.

Vremenska složenost algoritma određuje i njegovu praktičnu upotrebljivost tj. najveće ulazne vrednosti za koje je moguće da će se algoritam izvršiti u nekom razumnom vremenu. Analogno važi i za prostornu složenost.

Prostorna složenost odnosi se i na prostor koji zahvataju ulazni podaci. Kada se govori o potrebnom memorijskom prostoru ne računajući prostor koji zahvataju ulazni podaci, onda se govori o *dodatnoj prostornoj složenosti*.

U nastavku teksta najčešće će se govoriti o vremenskoj složenosti algoritama, ali u većini situacija potpuno analogna razmatranja mogu se primeniti na prostornu složenost.

3.1 Merenje i procenjivanje korišćenih resursa

Vreme izvršavanja programa može biti procenjeno ili izmereno za neke konkretne ulazne vrednosti i neko konkretno izvršavanje.

3.1.1 Merenje utrošenog vremena

Najjednostavnija mera vremenske efikasnosti programa (ili neke funkcije) je njegovo vreme izvršavanja za neke konkretne vrednosti. U standardnoj biblioteci raspoloživa je, kroz datoteku zaglavlja `time.h`, jednostavna podrška za merenje vremena. Struktura `time_t` služi za predstavljanje vremena, ali standard ne precizira način na koji se to čini. Funkcija `time`, deklarisana na sledeći način

```
time_t time(time_t *time);
```

vraća, u vidu vrednosti tipa `time_t`, vreme proteklo od početka *epohe*, obično od početka 1970. godine, izraženo u sekundama. Ukoliko vrednost parametra `time` nije `NULL`, onda se isto to vreme upisuje u promenljivu na koju ukazuje parametar.

Funkcija `difftime`:

```
double difftime(time_t end, time_t start);
```

vraća razliku između dva vremena, izraženu u sekundama. Ta razlika obično je celobrojna (ako je tip `time_t` celobrojan), ali je, zbog opštosti, povratni tip ipak `double`.

Struktura `clock_t` služi za predstavljanje vremena rada tekućeg procesa, ali standard ne propisuje način na koji se to čini. Funkcija `clock`:

```
clock_t clock(void);
```

vraća trenutno vreme, ali zbog različitosti mogućih implementacija, jedino ima smisla razmatrati razliku dva vremena dobijena ovom funkcijom. Razlika predstavlja broj vremenskih „tikova“, specifičnih za dati sistem, pa se broj sekundi između dva vremena može dobiti kao razlika dve vrednosti tipa `clock_t` podeljena konstantom `CLOCKS_PER_SEC`. Vreme izmereno na ovaj način je vreme koje je utrošio sam program (a ne i drugi programi koji se istovremeno izvršavaju) i može se razlikovati od proteklog apsolutnog vremena (može biti

kraće ako, na primer, ima više programa koji rade istovremeno, ili duže ako, na primer, program koristi više raspoloživih jezgara procesora).

Navedene funkcije mogu se koristiti za merenje vremena koje troši neka funkcija ili operacija. Precizniji rezultati dobijaju se ako se meri izvršavanje koje se ponavlja veliki broj puta. S druge strane, umesto u sekundama, vreme koje troši neka funkcija ili operacija često se pogodnije izražava u nanosekundama, kao u narednom primeru. U merenjima ovog tipa treba biti oprezan jer i naredbe petlje troše nezanemariv udeo vremena. Dodatno, prilikom kompilacije treba isključiti sve optimizacije (na primer, za kompilator `gcc`, to je opcija `-O0`) kako se ne bi merilo vreme za optimizovanu verziju programa. Naredni program ilustruje merenje vremena koje troši funkcija `f`.

Program 3.1.

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

#define BROJ_POZIVA 1000
#define NS_U_S      1000000000 /* nanosekundi u sekundi */

int f(void)
{
...
}

int main()
{
  int i;
  double ukupno_sekundi, ukupno_nanosekundi, f_nanosekundi;

  clock_t t_pocetak = clock();
  for(i = 0; i < BROJ_POZIVA; i++)
    f();
  clock_t t_kraj = clock();

  ukupno_sekundi = (double)(t_kraj-t_pocetak) / CLOCKS_PER_SEC;
  ukupno_nanosekundi = NS_U_S * ukupno_sekundi;
  f_nanosekundi = ukupno_nanosekundi / BROJ_POZIVA;

  printf("Procena vremena rada jednog poziva funkcije f: ");
  printf("%.2lf ns\n", f_nanosekundi);
  return 0;
}
```

U zaglavlju `time.h` postoje i funkcije koje pružaju podršku za baratanje

datumima, ali one nisu relevantne za merenje vremena rada programa pa neće ovde biti opisivane.

3.1.2 Procenjivanje potrebnog vremena

Vreme izvršavanja pojedinih delova koda (na nekom konkretnom računaru) može da se proceni ukoliko je raspoloživa procena vremena izvršavanja pojedinih operacija. Na primer, na računaru sa procesorom Intel Core i7-2670QM 2.2GHz koji radi pod operativnim sistemom Linux, operacija množenja dve vrednosti tipa `int` troši oko jednu nanosekundu (dakle, za jednu sekundu se na tom računaru može izvršiti oko milijardu množenja celih brojeva). Procena vremena izvršavanja pojedinih operacija i kontrolnih struktura na konkretnom računaru može se napraviti na način opisan u poglavlju 3.1.1.

Procene vremena izvršavanja programa na osnovu procena vremena izvršavanja pojedinačnih instrukcija treba uzimati sa velikom rezervom jer možda ne uzimaju u obzir sve procese koji se odigravaju tokom izvršavanja programa. Dodatno, treba imati na umu da vreme izmereno na jednom konkretnom računaru zavisi i od operativnog sistema pod kojim računar radi, od jezika i od kompilatora kojim je napravljen izvršivi program za testiranje, itd.

3.1.3 Profajliranje

Postoje mnogi alati za analizu i unapređenje performansi programa i najčešće se zovu *profajleri* (engl. profiler). Njihova osnovna uloga je da pruže podatke o tome koliko puta je koja funkcija pozvana tokom (nekog konkretnog) izvršavanja, koliko je utrošila vremena i slično. Ukoliko se razvijeni program ne izvršava željeno brzo, potrebno je unaprediti neke njegove delove. Prvi kandidati za izmenu su delovi koji troše najviše vremena.

Za operativni sistem Linux, popularan je sistem `valgrind` koji objedinjuje mnoštvo alatki za dinamičku analizu rada programa, uključujući profajler `callgrind`. Profajler `callgrind` se poziva na sledeći način:

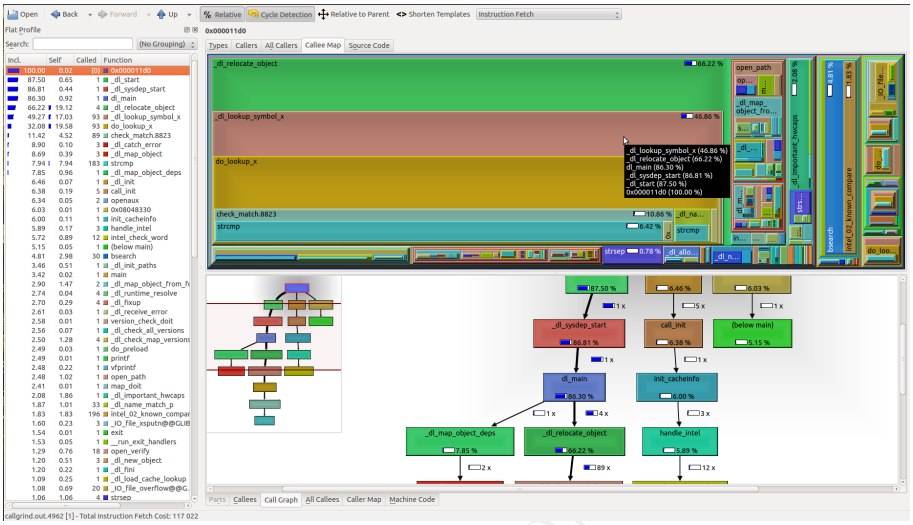
```
valgrind --tool=callgrind mojprogram argumenti
```

gde `mojprogram` označava izvršivi program koji se analizira, a `argumenti` njegove argumente (komandne linije). Program `callgrind` izvršava zadati program `mojprogram` sa argumentima `argumenti` i registruje informacije o tome koja funkcija je pozivala koje funkcije (uključujući sistemske funkcije i funkcije iz standardne biblioteke), koliko je koja funkcija utrošila vremena itd. Detaljniji podaci mogu se dobiti ako je program preveden u debag režimu (`gcc` kompilatorom, debag verzija se dobija korišćenjem opcije `-g`). Prikupljene informacije program `callgrind` čuva u datoteci sa imenom, na primer, `callgrind.out.4873`. Ove podatke može da na pregledan način prikaže, na primer, program `kcachegrind`:

```
kcachegrind callgrind.out.4873
```

Slika 3.1 ilustruje rad programa `kcachegrind`. Uvidom u prikazane podatke, programer može da uoči funkcije koje troše najviše vremena i da pokuša da ih

unapredi i slično.



Slika 3.1: Ilustracija prikaza u programu `kcachegrind` podataka dobijenih profajliranjem

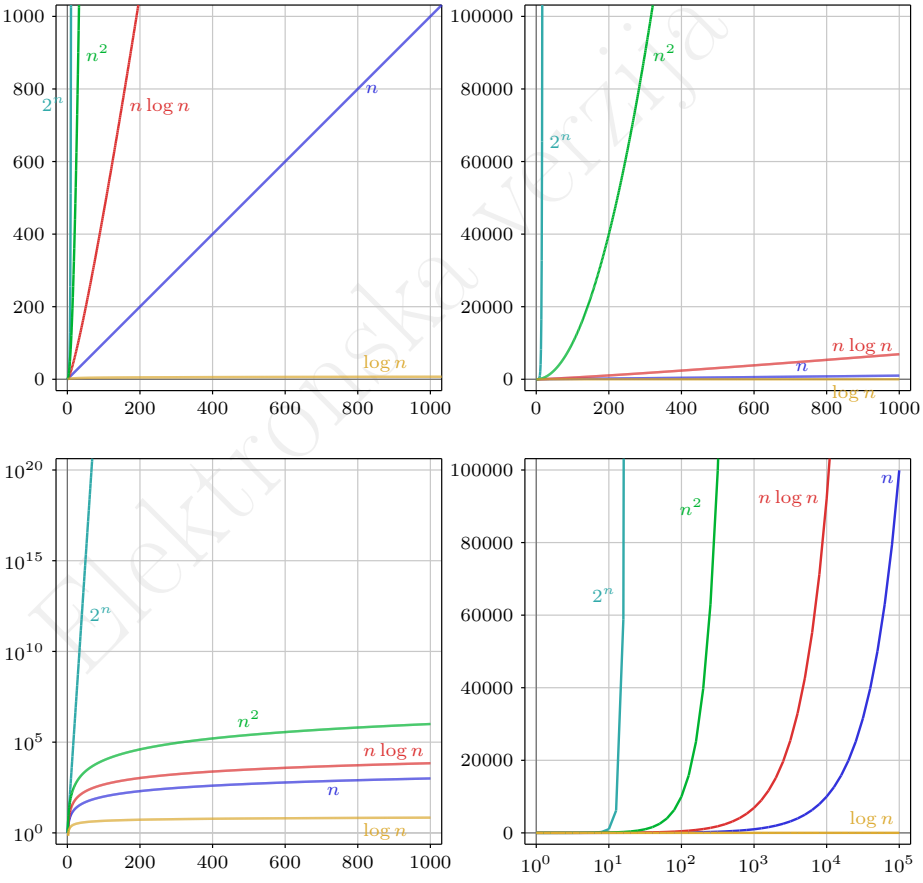
3.2 Asimptotsko ponašanje i red složenosti algoritma

Vreme izvršavanja programa može biti procenjeno ili izmereno za neke konkretne vrednosti ulaza, kao i za i neko konkretno izvršavanje. No, vreme izvršavanja programa može biti opisano opštije, u vidu funkcije koja zavisi od ulaznih argumenata. Relevantna veličina ulaza može biti sâm ulazni broj koji treba obraditi, a može biti veličina ulazne vrednosti, na primer — broj elemenata niza koji treba obraditi, broj bitova potrebnih za zapisivanje ulaznog argumenta koji treba obraditi, itd. U teorijskim analizama složenosti, pod veličinom ulaza obično se podrazumeva broj bitova potrebnih za zapisivanje tog ulaza. Potrebno je uvek eksplicitno navesti u odnosu na koju veličinu se razmatra rad algoritma.

Često se algoritmi ne izvršavaju isto, ne samo za ulaze različitih veličina, nego ni za sve ulaze istih veličina, pa je potrebno naći način za opisivanje i poređenje efikasnosti različitih algoritama. *Analiza najgoreg slučaja* zasniva procenu složenosti algoritma na najgorem slučaju (na slučaju za koji se algoritam najduže izvršava — u analizi vremenske složenosti, ili na slučaju za koji algoritam koristi najviše memorije — u analizi prostorne složenosti). Takva procena može da bude varljiva, ali predstavlja dobar opšti način za poređenje efikasnosti algoritama. U nekim situacijama moguće je izračunati prosečno vreme izvršavanja algoritma, ali i takva procena bi često mogla da bude varljiva.

Analiziranje najboljeg slučaja obično nema mnogo smisla. U nastavku će, ako nije rečeno drugačije, biti podrazumevana analiza najgoreg slučaja.

Neka je funkcija $f(n)$ jednaka broju instrukcija koje zadati algoritam izvrši za ulaz veličine n. Pod grubom pretpostavkom da izvršavanje svake instrukcije zahteva isto vreme, funkcija $f(n)$ predstavlja i meru ukupnog vremena potrebnog za izvršavanje algoritma (zato se broj instrukcija i vreme potrebno da se one izvrše analiziraju na veoma sličan način). Prilikom analize mnogih algoritama funkcija $f(n)$ izražava se kao neka kombinacija logaritamske funkcije $\log(n)$, polinomskih funkcija n, n^2, n^3, ..., eksponencijalnih funkcija 2^n, 3^n, ..., faktorijelne funkcije $n!$ i slično. Podsetimo se, zato, osnovnih matematičkih svojstava ovih funkcija.



Slika 3.2: Grafici funkcija koji se često javljaju u analizi složenosti, prikazani na različitim skalama

Priroda funkcija u matematici lakše se proučava i razume kada su dati njihovi grafici. U zavisnosti od relevantnih veličina ulaznih vrednosti ili od vrednosti funkcija (vrednosti funkcija mogu davati broj instrukcija ili vreme izvršavanja), nekad je pogodno za prikaz funkcija umesto uobičajenog Dekartovog sistema koristiti neku modifikovanu verziju. Na primer, možemo birati različite raspone x i y koordinata koje će biti prikazane na grafiku (u slučaju da je raspon neke od koordinata mnogo veći od one druge, odnos jediničnih podeoka se podešava tako da grafik i dalje zadrži skoro kvadratni oblik). Drugo, jedna od osa (pa i obe) može biti logaritamski transformisana — tada se od jedne do druge istaknute tačke skale ne dolazi sabiranjem sa određenom konstantom (1 u uobičajenom Dekartovom sistemu), već množenjem sa određenom konstantom (na primer, 2 ili 10). Na slici 3.2 prikazani su grafici funkcija koje se javljaju često u analizi složenosti.

Sa svih grafika jasno je uočljiv relativni odnos brzina rasta ovih funkcija. Nesporno je da među prikazanim funkcijama eksponencijalna funkcija raste najbrže, da je naredna po brzini rasta funkcija n^2, zatim $n \log n$, potom n i na kraju $\log n$ koja raste jako sporo. Sa druge strane, može se primetiti da se zaključci o međusobnom odnosu brzina rasta moraju izvoditi veoma pažljivo, jer se u zavisnosti od odabranog raspona x i y koordinata mogu izvući različiti zaključci. Na primer, na prvom grafiku i x i y su odabrani tako da idu do 1000, a na drugom je odabrano da x ide do 1000, a y do 100000. Sa prvog grafika se može steći utisak da je funkcija $n \log n$ po brzini rasta negde tačno između n i n^2, dok se sa drugog već taj odnos može preciznije proceniti i vidi se da funkcija n^2 mnogo brže raste nego $n \log n$ i n. Prvi prikazani grafik zapravo je samo mali deo drugog (dobija se tako što se y osa „saseče“ pri samom dnu, na vrednosti 1000). Stoga prilikom analize algoritama grafici treba da se podese tako da x-osa pokazuje dimenzije ulaza zaista relevantne za problem koji se rešava, a da se na y osi prikazuje realna procena broja instrukcija tj. vremena izvršavanja koje se u realnom kontekstu dopuštaju algoritmu. Ako prikazane funkcije mere broj instrukcija, onda je drugi grafik je u tom smislu mnogo bolji od prvog, jer se na prvom ne prikazuju izvršavanja nakon 1000 instrukcija, što je za današnje računare zanemarivo malo. Možemo reći da bi se još bolji grafik dobio kada bi se broj instrukcija povećao na milione, pa čak i milijarde, jer su današnji računari u stanju da izvrše milijarde instrukcija u nekoliko sekundi. Grafici sa logaritamskim skalama mnogo bolje mogu da predstave funkcije i njihove odnose, ali treba steći dobar osećaj za to kako ih treba čitati.

Pokušajmo da sada sa terena apstraktne matematike prebacimo na realan konktekst analize algoritama na savremenim računarima. Tabela 3.1 prikazuje potrebno vreme izvršavanja algoritma ako se pretpostavi da jedna instrukcija traje jednu nanosekundu – 10^{-9} sekundi, tj. 0.001 mikrosekundi (μs). Ova procena je gruba, ali nije previše pogrešna i daje dobru procenu realnih vremena na današnjim računarima.

Pokušajmo sada da podatke iz ove tabele predstavimo grafički (slika 3.3). Usled jako velike razlike u brzinama rasta analiziraćemo samo “susedne” funkcije.

n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	$0.003\,\mu s$	$0.01\,\mu s$	$0.033\,\mu s$	$0.1\,\mu s$	$1\,\mu s$	$3.63\,ms$
20	$0.004\,\mu s$	$0.02\,\mu s$	$0.086\,\mu s$	$0.4\,\mu s$	$1\,ms$	$77.1\,god$
30	$0.005\,\mu s$	$0.03\,\mu s$	$0.147\,\mu s$	$0.9\,\mu s$	$1\,s$	$8.4 \times 10^{15}\,god$
40	$0.005\,\mu s$	$0.04\,\mu s$	$0.213\,\mu s$	$1.6\,\mu s$	$18.3\,min$	
50	$0.006\,\mu s$	$0.05\,\mu s$	$0.282\,\mu s$	$2.5\,\mu s$	$13\,dan$	
10^2	$0.007\,\mu s$	$0.1\,\mu s$	$0.644\,\mu s$	$10\,\mu s$	$4 \times 10^{13}\,god$	
10^3	$0.010\,\mu s$	$1.0\,\mu s$	$9.966\,\mu s$	$1\,ms$		
10^4	$0.013\,\mu s$	$10\,\mu s$	$130\,\mu s$	$100\,ms$		
10^5	$0.017\,\mu s$	$0.10\,\mu s$	$1.67\,ms$	$10\,s$		
10^6	$0.020\,\mu s$	$1\,ms$	$19.93\,ms$	$16.7\,min$		
10^7	$0.023\,\mu s$	$0.01\,s$	$0.23\,s$	$1.16\,dan$		
10^8	$0.027\,\mu s$	$0.10\,s$	$2.66\,s$	$115.7\,dan$		
10^9	$0.030\,\mu s$	$1\,s$	$29.9\,s$	$31.7\,god$		

Tabela 3.1: Ilustracija vremena izvršavanja

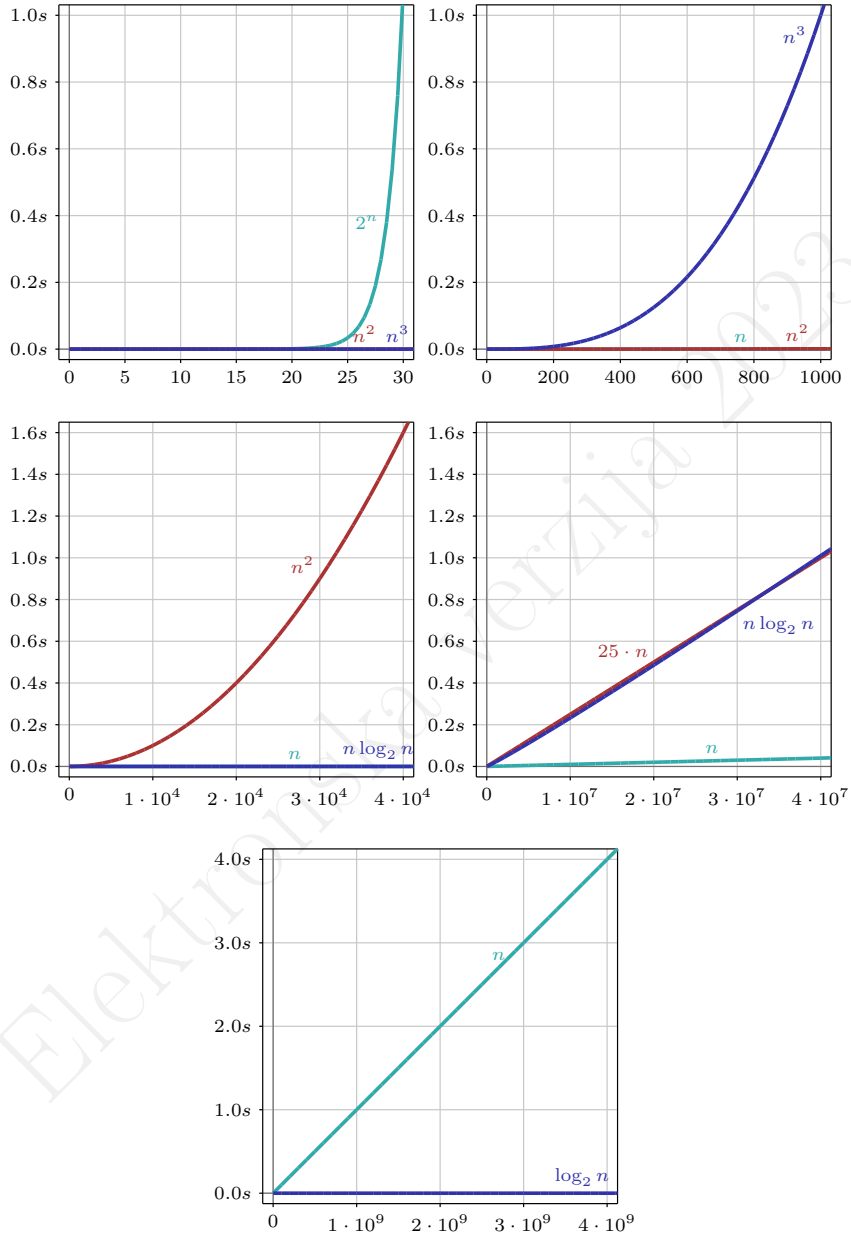
Na prvom grafiku na slici 3.3 prikazan je odnos vremena izvršavanja eksponencijalnih i polinomskih algoritama. Već za dimenziju 30 eksponencijalni algoritam zahteva oko jedne sekunde, dok je vreme izvršavanja polinomijalnih algoritama praktično zanemarivo (čak i u slučaju polinoma većeg stepena, poput n^3).

Na drugom grafiku prikazan je odnos brzina rasta polinomskih algoritama. Povećanjem stepena prati jako veliki porast vremena. Tako je algoritam koji zahteva n^3 instrukcija mnogo sporiji nego onaj koji zahteva n^2 instrukcija (kubnom je već za problem dimenzije oko 1000 potrebno oko jedne sekunde, dok kvadratni i linearni na tim dimenzijama posao obavljaju praktično momentalno).

Na trećem grafiku prikazan je odnos tri funkcije veoma česte u analizi algoritama: n^2, $n \log n$ i n. Sa grafika se može uočiti da na dimenzijama na kojima su kvadratnom algoritmu potrebne već sekunde, nema nikakve primetne razlike između algoritama kojima je potrebno $n \log n$ i n instrukcija – oba praktično momentalno završavaju posao.

Da bi se razlike između takvih algoritama opazila, potrebno je da se dimenzija ulaza znatno poveća, što je i prikazano na četvrtom grafiku. Tek kada dimenzija ulaza dostigne milione, vidi se da je algoritam koji zahteva $n \log n$ instrukcija nešto sporiji. Sa grafika se vidi da svojim oblikom ta funkcija jako liči na linearnu (otuda i naziv "kvazilinearna" funkcija). Sa grafika se može videti i da razlika između linearne i kvazilinearne funkcije nije "drastična", jer se povećavanjem konstantnog faktora uz linearnu funkciju (faktora 25 na ovom grafiku) može desiti da na ulazima razmatrane dimenzije broj koraka bude veći nego kod osnovne kvazilinearne funkcije.

Na petom grafiku se vidi da je vreme izvršavanja algoritama kod kojih broj koraka logaritamski zavisi od dimenzije ulaza praktično zanemarivo (čak



Slika 3.3: Grafička ilustracija vremena izvršavanja (na x-osi je dimenzija problema, a na y-osi je vreme u sekundama)

i za ogromne ulaze od milijardu elemenata). Treba imati na umu da je samo za učitavanje tolikog ulaza potrebno linearno vreme, pa prednost algoritama logaritamske složenosti dolazi tek kod problema kod kojih se nakon učitavanja podaci obrađuju veliki broj puta ili kod algoritama kod kojih nema potrebe za učitavanjem svih podataka.

Zaključak svih prikazanih grafika je da unapređivanje algoritma tako da se broj koraka umesto jednom funkcijom iz našeg razmatranog niza funkcija izražava prethodnom, može da donosi drastično smanjenje vremena izvršavanja i mogućnost obrade mnogo većih ulaza. Izuzetak delom predstavljaju funkcije $n \log n$ i n, koje su veoma bliske i njihova razlika dolazi do izražaja tek kod jako velikih ulaza (ovo je jasno kada se pogleda količnik svake dve susedne funkcije – kod ove dve funkcije on je ubedljivo najmanji).

U svim dosadašnjim analizama smo pretpostavili da je broj koraka tj. vreme izvršavanja određeno tačno nekom od funkcija prikazanih na graficima i u tabeli, međutim, to je retko slučaj. Broj koraka se obično izračunava kao neka kombinacija tih funkcija. Lako se može zaključiti da u tim slučajevima vodeći član u funkciji $f(n)$ određuje potrebno vreme izvršavanja. Tako, na primer, ako je broj instrukcija $n^2 + 2n$, onda za ulaz veličine 1000000, član n^2 odnosi 16.7 minuta dok član $2n$ odnosi samo dodatne dve milisekunde. Vremenska (a i prostorna) složenost je, dakle, skoro potpuno određena „vodećim“ (ili „dominantnim“) članom u izrazu koji određuje broj potrebnih instrukcija. Na upotrebljivost algoritma ne utiču mnogo ni multiplikativni i aditivni konstantni faktori u broju potrebnih instrukcija, koliko asimptotsko ponašanje broja instrukcija u zavisnosti od veličine ulaza. Za ovakav pojam složenosti veoma je važna sledeća definicija.[1]

Definicija 3.1. *Ako postoje pozitivna realna konstanta c i prirodan broj n_0 takvi da za funkcije f i g nad prirodnim brojevima važi*

$$f(n) \leq c \cdot g(n) \quad \textit{za sve prirodne brojeve } n \textit{ veće od } n_0$$

onda pišemo

$$f(n) = O(g(n))$$

i čitamo „f je veliko o od g“.

Naglasimo da $O(g(n))$ ne označava neku konkretnu funkciju, već klasu funkcija i uobičajeni zapis $f(n) = O(g(n))$ zapravo znači $f(n) \in O(g(n))$. Pored toga, kako O predstavlja odnos između funkcija, a ne njihovih konkretnih vrednosti, pod zapisom $f(n) \in O(g(n))$, zapravo podrazumevamo $f \in O(g)$ (jer su f i g funkcije, a $f(n)$ i $g(n)$ njihove vrednosti).

Lako se pokazuje da aditivne i multiplikativne konstante ne utiču na klasu kojoj funkcija pripada (na primer, u izrazu $5n^2 + 1$, za konstantu 1 kažemo da

[1] Pojmovi „veliko o“ i „veliko teta“ mogu da se uvedu i za funkcije nad realnim brojevima, ali za potrebe analize složenosti izračunavanja dovoljne su verzije za funkcije nad prirodnim brojevima.

je aditivna, a za konstantu 5 da je multiplikativna). Drugim rečima, ako $f(n)$ pripada klasi $O(g(n))$, onda $a \cdot f(n) + b$ takođe pripada klasi $O(g(n))$. Zato se može reći da je neki algoritam složenosti $O(n^2)$, ali se obično ne govori da pripada, na primer, klasi $O(5n^2+1)$, jer aditivna konstanta 1 i multiplikativna 5 nisu od suštinske važnosti. Zaista, ako jedan algoritam zahteva $5n^2+1$ instrukcija, a drugi n^2, i ako se prvi algoritam izvršava na računaru koji je šest puta brži od drugog, on će biti brže izvršen za svaku veličinu ulaza. No, ako jedan algoritam zahteva n^2 instrukcija, a drugi n, ne postoji računar na kojem će prvi algoritam da se izvršava brže od drugog za svaku veličinu ulaza.

Primer 3.1. Može se dokazati da važi:

- $n^2 = O(n^2)$
  Tvrđenje važi jer $c = 1$ (ali i za veće vrednosti c), važi $n^2 \leq c \cdot n^2$ za sve vrednosti n veće od 0.

- $n^2 + 10 = O(n^2)$
  Za $c = 2$, važi $n^2 + 10 \leq c \cdot n^2$ za sve vrednosti n veće od 3 (jer za takve vrednosti n važi $n^2 \leq n^2$ i $10 \leq n^2$).

- $5n^2 + 10 = O(n^2)$
  Za $c = 6$, važi $5n^2 + 10 \leq c \cdot n^2$ za sve vrednosti n veće od 3 (jer za takve vrednosti n važi $5n^2 \leq 5n^2$ i $10 \leq n^2$).

- $7n^2 + 8n + 9 = O(n^2)$
  Za $c = 16$, važi $7n^2 + 8n + 9 \leq c \cdot n^2$ za sve vrednosti n veće od 2 (jer za takve vrednosti n važi $7n^2 \leq 7n^2$ i $8n \leq 8n^2$ i $9 \leq n^2$).

- $n^2 = O(n^3)$
  Za $c = 1$, važi $n^2 \leq c \cdot n^3$ za sve vrednosti n veće od 0.

- $7 \cdot 2^n + 8 = O(2^n)$
  Za $c = 8$, važi $7 \cdot 2^n + 8 \leq c \cdot 2^n$ za sve vrednosti n veće od 2 (jer za takve vrednosti n važi $7 \cdot 2^n \leq 7 \cdot 2^n$ i $8 \leq 2^n$).

- $2^n + n^2 = O(2^n)$
  Za $c = 2$, važi $2^n + n^2 \leq c \cdot 2^n$ za sve vrednosti n veće od 3 (jer za takve vrednosti n važi $2^n \leq 2^n$ i $n^2 \leq 2^n$; da važi $n^2 \leq 2^n$ za $n > 3$ može se dokazati, na primer, matematičkom indukcijom).

- $5 \cdot 3^n + 7 \cdot 2^n = O(3^n)$
  Za $c = 12$, važi $5 \cdot 3^n + 7 \cdot 2^n \leq c \cdot 3^n$ za sve vrednosti n veće od 0 (jer za takve vrednosti n važi $5 \cdot 3^n \leq 5 \cdot 3^n$ i $7 \cdot 2^n \leq 7 \cdot 3^n$; da važi $2^n \leq 3^n$ za $n > 0$ može se dokazati, na primer, matematičkom indukcijom).

- $2^n + 2^n n = O(2^n n)$
  Za $c = 2$, važi $2^n + 2^n n \leq c \cdot 2^n n$ za sve vrednosti n veće od 0 (jer za takve vrednosti n važi $2^n \leq 2^n n$ i $2^n n \leq 2^n n$).

Teorema 3.1.

- *Ako su a i b realni brojevi i $a > 0$, onda važi $af(n) + b = O(f(n))$ (tj. multiplikativne i aditivne konstante ne utiču na klasu kojoj funkcija pripada).*
- *Ako važi $f_1(n) = O(g_1(n))$ i $f_2(n) = O(g_2(n))$, onda važi i $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.*
- *Ako važi $f_1(n) = O(g_1(n))$ i $f_2(n) = O(g_2(n))$, onda važi i $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$.*

Definicija 3.2. *Ako postoje pozitivna realna konstanta c i prirodan broj n_0 takvi da za funkcije f i g nad prirodnim brojevima važi*

$$c \cdot g(n) \leq f(n) \text{ za sve prirodne brojeve } n \text{ veće od } n_0$$

onda pišemo

$$f(n) = \Omega(g(n))$$

i čitamo „f je veliko omega od g“.

Definicija 3.3. *Ako postoje pozitivne realne konstante c_1 i c_2 i prirodan broj n_0 takvi da za funkcije f i g nad prirodnim brojevima važi*

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ za sve prirodne brojeve } n \text{ veće od } n_0$$

onda pišemo

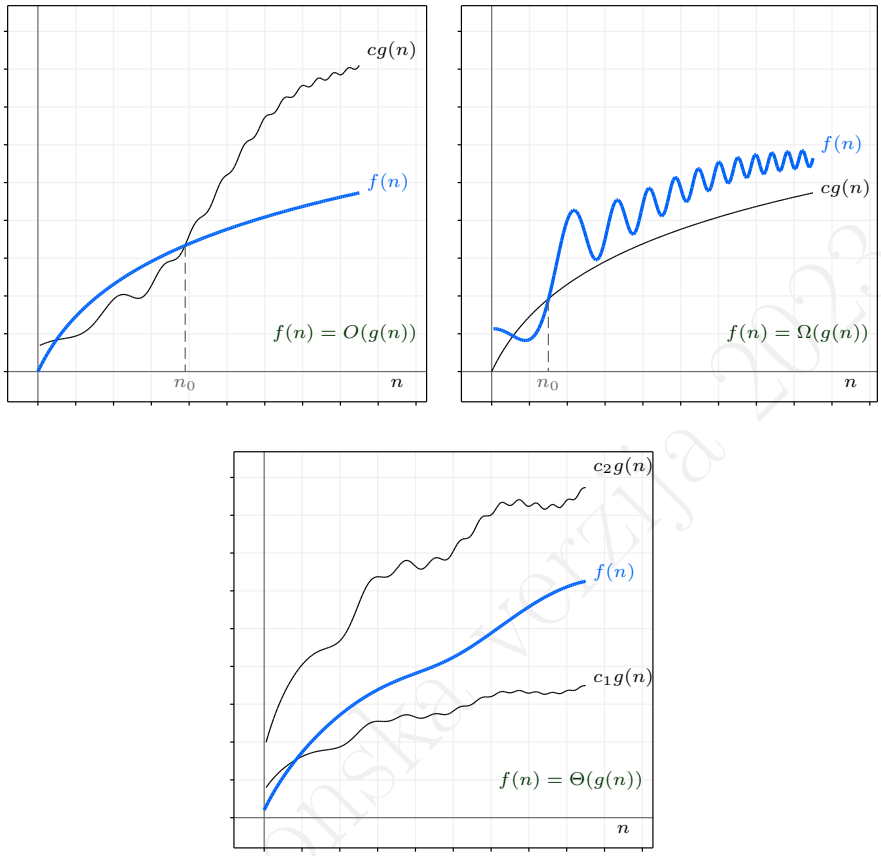
$$f(n) = \Theta(g(n))$$

i čitamo „f je veliko teta od g“.

Pojmovi *veliko o*, *veliko omega* i *veliko teta* ilustrovani su na slici 3.4.

Primer 3.2. Može se dokazati da važi:

- $5 \cdot 2^n + 9 = \Theta(2^n)$
  Za $c_1 = 5$, važi $c_1 \cdot 2^n \leq 5 \cdot 2^n + 9$ za sve vrednosti n veće od 0. Za $c_2 = 6$, važi $5 \cdot 2^n + 9 \leq c_2 2^n$ za sve vrednosti n veće od 3. Iz navedena dva tvrđenja sledi zadato tvrđenje.
- $2^n + 2^n n = \Theta(2^n n)$
  Za $c_1 = 1$, važi $c_1 \cdot 2^n n \leq 2^n + 2^n n$ za sve vrednosti n veće od 0. Za $c_2 = 2$, važi $2^n + 2^n n \leq c_2 2^n n$ za sve vrednosti n veće od 0. Iz navedena dva tvrđenja sledi zadato tvrđenje.



Slika 3.4: Ilustracija pojmova „veliko o“, „veliko omega“ i „veliko teta“

Teorema 3.2.

- *Ako važi* $f(n) = O(g(n))$ *i* $g(n) = O(h(n))$*, onda važi i* $f(n) = O(h(n))$ *(ovakva tranzitivnost važi i za* Θ *i* Ω*).*
- *Važi* $f(n) = \Theta(g(n))$ *akko važi* $f(n) = O(g(n))$ *i* $f(n) = \Omega(g(n))$*.*
- *Ako važi* $f(n) = O(g(n))$*, onda važi i* $g(n) = \Omega(f(n))$*.*
- *Ako važi* $f(n) = \Theta(g(n))$*, onda važi i* $f(n) = O(g(n))$ *i* $g(n) = O(f(n))$*.*
- *Ako važi* $f(n) = \Theta(g(n))$*, onda važi i* $g(n) = \Theta(f(n))$*.*

Definicija 3.4. *Ako je $T(n)$ vreme izvršavanja algoritma A (čiji ulaz karakteriše prirodan broj n) i ako važi $T(n) = O(g(n))$, onda kažemo da je algoritam A* složenosti *ili* reda $O(g(n))$ *ili da algoritam A* pripada klasi $O(g(n))$.

Analogno prethodnoj definiciji definiše se kada algoritam A pripada klasi $\Omega(g(n))$ i kada pripada klasi $\Theta(g(n))$. Složenost algoritama najčešće se izražava u terminima O (što važi i za nastavak ove knjige).

Informacija o složenosti algoritma u terminima Θ (koja daje i gornju i donju granicu) preciznija je nego informacija u terminima O (koja daje samo gornju granicu) ili informacija u terminima Ω (koja daje samo donju granicu). Međutim, obično je složenost algoritma jednostavnije iskazati u terminima O nego u terminima Θ. Štaviše, za neke algoritme složenost se ne može lako iskazati u terminima Θ. Na primer, ako za neke ulaze algoritam troši n, a za neke n^2 vremenskih jedinica, za taj algoritam se ne može reći ni da je reda $\Theta(n)$ ni reda $\Theta(n^2)$, ali jeste reda $O(n^2)$ (pa i, na primer, reda $O(n^3)$). Kada se kaže da algoritam pripada klasi $O(g(n))$ obično se podrazumeva da je g najmanja takva klasa (ili makar — najmanja za koju se to može dokazati). I O i Θ notacija se koriste i u analizi najgoreg slučaja i u analizi prosečnog slučaja.

Složenost algoritama u terminima Ω razmatra se ređe nego složenost u terminima Θ i O, ali ponekad takođe može da bude veoma važna. Složenost u terminima O daje gornju granicu za neku funkciju, a složenost u terminima Ω daje donju granicu. To se može razumeti i ovako: složenost u terminima O govori nam koliko je neki algoritam dobar („asimptotski nije lošiji nego...“), a složenost u terminima Ω govori nam koliko je neki algoritam loš („asimptotski nije bolji nego...“). Na primer, može se pokazati da je prosečno vreme izvršavanja nekog algoritma za sortiranje reda $\Omega(n^2)$ i to govori da je taj algoritam loš (jer postoje algoritmi čije vreme izvršavanje pripada klasi $O(n \log n)$).

Lako se dokazuje da funkcija koja je konstantna pripada klasama $O(1)$, $\Omega(1)$ i $\Theta(1)$. Štaviše, svaka funkcija koja je ograničena odozgo nekom konstantom pripada klasama $O(1)$ i $\Theta(1)$. Treba imati na umu da i velike konstantne vrednosti kao, na primer, 1000000 pripadaju klasi $O(1)$.

Za algoritme složenosti $O(1)$ kažemo da imaju *konstantnu* složenost, za algoritme složenosti $O(n)$ kažemo da imaju *linearnu*, za $O(n \log n)$ da imaju *kvazilinearnu*, za $O(n^2)$ *kvadratnu*, za $O(n^3)$ *kubnu*, za $O(n^k)$ za neko k *polinomsku* (negde se kaže i *polinomijalnu*), za $O(a^n)$ za neko a *eksponencijalnu*, a za $O(\log n)$ *logaritamsku složenost*.

Priroda parametra klase složenosti (na primer, n u $O(n)$ ili m i k u $O(2^{m+k})$) zavisi od samog algoritma. Složenost nekih algoritama zavisi od vrednosti argumenata, nekih od broja argumenata, a nekad (posebno u teorijskim analizama složenosti) se pogodno iskazuje u zavisnosti od broja bitova potrebnih da se zapiše ulaz. Na primer, složenost funkcije za izračunavanje faktorijela ulazne vrednosti m zavisi od m i jednaka je (za razumnu implentaciju) $O(m)$. Složenost funkcije koja računa prosek k ulaznih brojeva ne zavisi od vredno-

sti tih brojeva, već samo od toga koliko ih ima i jednaka je $O(k)$. Složenost funkcije koja sabira dva broja (fiksne širine) je konstantna tj. pripada klasi $O(1)$. Složenost izračunavanja neke funkcije može da zavisi i od više parametara. Na primer, algoritam koji za n ulaznih tačaka proverava da li pripadaju unutrašnjosti nekog od m zadatih trouglova, ima složenost $O(mn)$. Potrebno je uvek eksplicitno navesti u odnosu na koju veličinu ili koje veličine se razmatra složenost algoritma.

3.3 Izračunavanje složenosti funkcija

Izračunavanje (vremenske i prostorne) složenosti funkcija zasniva se na određivanju tačnog ili približnog broja instrukcija koje se izvršavaju i memorijskih jedinica koje se koriste. Tačno određivanje tih vrednosti najčešće je veoma teško ili nemoguće, te se obično koriste razna pojednostavljivanja. Na primer, u ovom kontekstu pojam „jedinična instrukcija“ se ponekad pojednostavljuje, pa se smatra da sve pojedinačne naredbe (bez poziva funkcija) troše jednako vremena. Ono što je važno je da takva pojednostavljivanja ne utiču na klasu složenosti kojoj algoritam pripada (jer, kao što je rečeno, konstantni aditivni i multiplikativni faktori ne utiču na red algoritma). Zbog toga se u asimptotskoj analizi složenosti obično i ne broje pojedinačne naredbe, već je dovoljno znati da one troše konstantno vreme za izvršavanje.

Ukoliko se deo programa sastoji od nekoliko instrukcija bez grananja, onda se procenjuje da je njegovo vreme izvršavanja uvek isto, konstantno, te da pripada klasi $O(1)$. Ukoliko program ima dva dela, vremenske složenosti $O(f)$ i $O(g)$[2], koji se izvršavaju jedan za drugim, ukupna složenost je $O(f+g)$[3]. Isto važi i u slučaju kada se u tim delovima javljaju rekurzivni pozivi (tada, na primer, vremenska složenost izražena u terminima vrednosti $f(n)$ može da zavisi od vremenske složenosti izražene u terminima vrednosti $f(n-1)$), ali tada efektivno izračunavanje složenosti zahteva korišćenje dodatnih tehnika (videti primer 3.4 i poglavlje 4.3). Ukoliko deo programa sadrži jedno grananje i ukoliko vreme izvršavanja jedne grane pripada klasi $O(f)$ a druge grane pripada klasi $O(g)$, onda je ukupno vreme izvršavanja tog dela programa ograničeno vremenom koje zahteva složenija grana, pa pripada klasi $O(\max(f,g))$, a ona je jednaka klasi $O(f+g)$. Ukoliko deo programa sadrži petlju koja se izvršava n puta, a vreme izvršavanja tela petlje je konstantno, onda ukupno vreme izvršavanja tog dela programa pripada klasi $O(n)$. Ukoliko deo programa sadrži petlju koja se izvršava za vrednosti i od 1 do n, a vreme izvršavanja tela petlje je $O(f(i))$, onda ukupno vreme izvršavanja tog dela programa pripada klasi $O(f(1)+f(2)+\ldots+f(n))$. Ukoliko deo programa sadrži dvostruku petlju – jednu koja se izvršava m puta i, unutar nje, drugu koja se izvršava n puta i ukoliko je vreme izvršavanja tela unutrašnje petlje konstantno, onda ukupno

[2]Gde su f i g funkcije koje zavise od jednog ili više parametara programa.

[3]Na primer, ako je prvi deo složenosti $O(n)$ a drugi složenosti $O(n^2)$, onda je ukupna vremenska složenost $O(n+n^2)$, što je opet $O(n^2)$. Tada kažemo da drugi deo programa dominira u vremenu izvršavanja.

vreme izvršavanja tog dela programa pripada klasi $O(m \cdot n)$. Ova pravila mogu se dalje uopštiti. Analogno se računa vremenska složenost za druge vrste kombinovanja linearnog koda, grananja i petlji.

Što se tiče prostorne složenosti, ukoliko deo programa ne sadrži pozive funkcija, dinamičku alokaciju, niti deklaracije objekata čija veličina zavisi od nekih parametera, onda je prostorna složenost tog dela programa konstanta tj. pripada klasi $O(1)$. Ukoliko neki deo programa vrši dinamičku alokaciju nekih n objekata veličine $O(1)$ – onda to doprinosi njegovoj prostornoj složenosti $O(n)$. Ukoliko program ima dva dela, prostorne složenosti $O(f)$ i $O(g)$, koji se izvršavaju jedan za drugim, ukupna složenost je $O(f + g)$.[4] Ukoliko se u programu javljaju pozivi funkcija, svaki poziv zauzima neki fiksni prostor na programskom steku (veličina tog prostora može biti ograničena jednom konstantom zajedničkom za sve funkcije) kao i dodatni prostor koji zauzimaju lokalne promenljive te funkcije (koje su smeštene na stek, a ne u registre procesora). Ukoliko se javljaju rekurzivni pozivi, onda u prostornu složenost ulazi maksimalni broj stek okvira koji se mogu naći na programskom steku tokom izvršavanja (više o tome u glavi 4) i čija je veličina konstantna.[5] Ukoliko deo programa prostorne složenosti $O(f)$ poziva funkciju prostorne složenosti $O(g)$, onda ukupna prostorna složenost tog dela programa pripada klasi $O(f + g)$. Analogno se računa prostorna složenost za druge vrste kombinovanja koda.

Primer 3.3. Izračunajmo vremensku složenost naredne funkcije koja ispisuje trougaoni deo tablice množenja:

```
void mnozenje(int n)
{
  int i, j;
  if (n == 0)
    return;
  else {
    for(i=1; i<=n; i++) {
      for(j=i; j<=n; j++) {
        printf("%i*%i = %i \t", i, j, i*j);
      }
```

[4] Primetimo da ovo važi i ako prvi deo oslobađa prostor koji je zauzeo. Naime, $O(f + g)$ daje prostornu složenost u najgorem slučaju koja istovremeno ograničava odozgo i $O(f)$ i $O(g)$. U ovakvoj situaciji prostorna složenost ponaša se drugačije od vremenske: ako postoje dva dela programa koja se izvršavaju jedan za drugim, ukupno vreme izvršavanja (ne asimptotsko ograničenje nego konkretno utrošeno vreme) jednako je zbiru dva vremena izvršavanja, a ukupni zauzeti prostor jednak je maksimumu dva zauzeta prostora: jedan deo programa je koristio i oslobodio neki prostor a drugi deo programa je onda delom koristio isti taj prostor (na primer, na programskom steku).

[5] Treba imati na umu da je veličina stek okvira za svaku funkciju konstanta, ali ona može biti veoma velika, na primer – ako je u funkciji deklarisan niz velike dimenzije. Dodatno, treba imati na umu da stek okviri funkcija zauzimaju prostor na programskom steku, a prostor za njega je obično daleko manji od memorije u ostalim segmentima.

```
      printf("\n");
    }
  }
}
```

Za `n` jednako 5, funkcija daje izlaz:

```
1*1 = 1     1*2 = 2     1*3 = 3     1*4 = 4     1*5 = 5
2*2 = 4     2*3 = 6     2*4 = 8     2*5 = 10
3*3 = 9     3*4 = 12    3*5 = 15
4*4 = 16    4*5 = 20
5*5 = 25
```

Smatraćemo da se telo unutrašnje petlje u `else` grani izvršava konstantno vreme c. Unutrašnja petlja ima $n-i+1$ iteracija, te je njeno vreme izvršavanja $(n-i+1)c$. Spoljašnja petlja ima n iteracija, a i-ta iteracija ima vreme izvršavanja $(n-i+1)c$. Ukupno vreme izvršavanja spoljašnje petlje je

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

Grana `if` izvršava se konstantno vreme c', pa je ukupna složenost funkcije `mnozenje` $O(c' + \frac{n(n+1)c}{2}) = O(n(n+1)) = O(n^2)$.

Funkcija `mnozenje` nema poziva drugih funkcija i ne koristi dinamičku alokaciju, niti objekte čija veličina zavisi od ulaznih parametara, te je njena prostorna složenost konstantna, tj. pripada redu $O(1)$.

Primer 3.4. Izračunajmo vremensku složenost naredne rekurzivne funkcije koja izračunava faktorijel svog argumenta:

```
unsigned faktorijel(unsigned n)
{
  if (n == 0)
    return 1;
  else
    return n * faktorijel(n-1);
}
```

Neka $T(n)$ označava broj instrukcija koje zahteva poziv funkcije `faktorijel` za ulaznu vrednost n. Za $n = 0$, važi $T(n) = a$, gde je a neka konstanta. Za $n > 0$, važi $T(n) = b + T(n-1)$, gde je b neka konstanta (u tom slučaju izvršava se jedno poređenje, jedno oduzimanje, broj instrukcija koje zahteva funkcija `faktorijel` za argument $n-1$, jedno množenje i jedna naredba `return`). Dakle,

$$T(n) = b + T(n-1) = b + b + T(n-2) = \ldots = \underbrace{b + b + \ldots + b}_{n} + T(0) = bn + a$$

Dakle, navedena funkcija ima linearnu vremensku složenost.

Razmotrimo i prostornu složenost $S(n)$ funkcije `faktorijel`. Jedna instanca funkcije `faktorijel` zauzima (na programskom steku) konstantan prostor c. Najviše prostora je zauzeto kada se izvršava rekurzivni poziv, te za prostornu složenost važi:

$S(0) = c$

$S(n) = S(n-1) + c$

odakle se dobija da $S(n)$ pripada klasi $O(n)$.

Za izračunavanje složenosti komplikovanijih rekurzivnih funkcija potreban je matematički aparat za rešavanje *rekurentnih jednačina*, opisan u poglavlju 4.3.

3.4 Klase složenosti

Neka od najvažnijih otvorenih pitanja matematike i informatike vezana su za složenost izračunavanja. Jedan takav problem biće opisan u nastavku teksta.

Šef protokola na jednom dvoru treba da organizuje bal za predstavnike ambasada. Kralj traži da na bal bude pozvan Peru ili da ne bude pozvan Katar (Qatar). Kraljica zahteva da budu pozvani Katar ili Rumunija (ili i Katar i Rumunija). Princ zahteva da ne bude pozvana Rumunija ili da ne bude pozvan Peru (ili da ne budu pozvani ni Rumunija ni Peru). Da li je moguće organizovati bal i zadovoljiti zahteve svih članova kraljevske porodice?

Ako su p, q i r bulovske (logičke) promenljive (koje mogu imati vrednosti *tačno* ili *netačno*), navedeni zadatak može biti formulisan na sledeći način: da li je *zadovoljiv* logički iskaz

$$(p \vee \neg q) \wedge (q \vee r) \wedge (\neg r \vee \neg p)$$

Ovaj zadatak je jedna instanca, jedan konkretan primerak opštijeg problema – problema ispitivanja da li je zadovoljiva data iskazna formula u konjunktivnoj normalnoj formi[6]. Ovaj opšti problem zove se *SAT* (od engleskog *satisfiability*). Zadovoljivost takvog iskaza može biti ispitana tako što bi bile ispitane sve moguće dodele vrednosti promenljivama (p, q i r u gore navedenom primeru), za n promenljivih ima ih 2^n. Dati iskaz je zadovoljiv ako i samo ako je u nekoj interpretaciji vrednost datog logičkog iskaza *tačno*. Za izabranu, fiksiranu interpretaciju može se u polinomskom vremenu u odnosu na dužinu zadate formule utvrditi da li je istinitosna vrednost te formule *tačno*. Pitanje je, međutim, da li se zadovoljivost (bez fiksiranja interpretacije) može ispitati u polinomskom vremenu u odnosu na dužinu zadate formule. Nijedan trenutno poznat algoritam za SAT nema polinomsku složenost, svi imaju eksponencijalnu složenost. Formulišimo ovo pitanje malo sažetije, koristeći pojam *klase P*.

[6] Kažemo da je iskazna formula u *konjunktivnoj normalnoj formi*, ako je ona konjunkcija formula koje su disjunkcije nad promenljivama ili njihovim negacijama.

Definicija 3.5 (Klasa P). *Za algoritam sa veličinom ulaza n kažemo da je* polinomske složenosti *ako je njegovo vreme izvršavanja $O(p(n))$ gde je $p(n)$ polinom po n. Klasa problema za koje postoje polinomski algoritmi koji ih rešavaju označava se sa P.*

Pitanje je, dakle, da li važi SAT $\in$ P. Odgovor na ovo pitanje, otvoreno od 1971. godine, još uvek nije poznat. Odgovor, bilo pozitivan bilo negativan, imao bi ogroman uticaj na računarstvo i matematiku.[7] Ukoliko bi za problem SAT bio pronađen polinomski algoritam, onda bi polinomski algoritmi postojali i za mnoge druge važne i teške probleme. S druge strane, ukoliko se zna da za problem SAT ne postoji polinomski algoritam, onda polinomski algoritmi ne mogu postojati ni za mnoge druge važne i teške probleme.

Pored klase P, izučavaju se i mnoge druge klase problema i algoritama na osnovu njihove prostorne i vremenske složenosti.

3.5 Popravljanje vremenske složenosti

Ukoliko performanse programa nisu zadovoljavajuće, treba razmotriti zamenu ključnih algoritama – algoritama koji dominantno utiču na složenost. Postoji niz važnih i elementarnih i naprednih tehnika koje mogu pomoći da se snizi složenost algoritama, međutim, njima se nećemo u značajnoj meri baviti u ovom udžbeniku.

Ukoliko zamena algoritama efikasnijim ne uspeva tj. ukoliko se smatra da je asimptotsko ponašanje najbolje moguće, preostaje da se pokuša popravljanje efikasnosti snižavanjem konstantnih faktora u funkciji koja opisuje složenost (time se ne menja asimptotsko ponašanje, ali se ponekad može donekle smanjiti ukupno utrošeno vreme – na primer, program se može modifikovanti tako da izvršava $6n+3$ instrukcija umesto $7n+2$). Ubrzavanje programa često zahteva specifična rešenja, ali postoje i neke ideje koje su primenljive u velikom broju slučajeva.

Koristiti optimizacije kompilatora. Moderni kompilatori i u podrazumevanom režimu rada generišu veoma efikasan kôd ali mogu da se posebno podese tako da primene i dodatne tehnike optimizacije. Korišćenje ovih tehnika veoma značajno može da poboljša efikasnost programa (one najčešće utiču na konstantni faktor, ali postoje primeri kada se vremenska ili memorijska složenost mogu poboljšati i asimptotski). U najvećem broju slučajeva korišćenje naprednih optimizacija je poželjno, ali ono može dovesti i do određenih poteškoća.

[7] Odgovor na ovo pitanje, bilo pozitivan bilo negativan, neposredno bi vodio do odgovora da li je klasa P jednaka izvesnoj klasi NP (takozvani nedeterministički polinomski problemi). Matematički institut Klej nudi nagradu od milion dolara za svaki od izabranih sedam najznačajnijih otvorenih matematičkih i informatičkih problema a problem da li su klase P i NP jednake je prvi na toj listi.

Nakon naprednih optimizacija izvorni kôd transformiše se u asembler na netrivijalan način, što narušava vezu između izvornog i izvršivog programa. Zbog toga je optimizovane programe teško ili nemoguće analizirati degaberom i profajlerom.

Optimizacijama koje pruža kompilator mogu se dobiti ubrzanja i od nekoliko desetina procenata, ali treba imati na umu da postoji i mogućnost da optimizovani program radi sporije nego neoptimizovani (jer neka tehnika optimizacije u tom konkretnom slučaju ne daje dobre rezultate).

Proces kompilacija sa intenzivnim optimizovanjem može biti znatno sporiji od osnovne kompilacije. Zato se preporučuje da se optimizovanje programa primenjuje tek u završnim fazama razvoja programa — nakon intenzivnog testiranja i otklanjanja svih otkrivenih grešaka, ali i da se testiranje ponovi sa optimizovanom verzijom izvršivog programa.

Kompilatori obično imaju mogućnost da se eksplicitno izabere neka tehnika optimizacije ili nivo optimizovanja (što uključuje ili ne uključuje više pojedinačnih tehnika). Na primer, za kompilator `gcc` nivo optimizacije se bira korišćenjem opcije `-O` iza koje može biti navedena jedna od vrednosti:

0 za podrazumevani režim kompilacije, samo sa bazičnim tehnikama optimizacije.

1 kompilator pokušava da smanji i veličinu izvršivog programa i vreme izvršavanja, ali ne primenjuje tehnike optimizacije koje mogu da bitno uvećaju vreme kompilacije.

2 kompilator primenjuje tehnike optimizacije koje ne zahtevaju dodatni memorijski prostor u zamenu za veću brzinu (u fazi izvršavanja programa).

3 kompilator primenjuje i tehnike optimizacije koje zahtevaju dodatni memorijski prostor u zamenu za veću brzinu (u fazi izvršavanja programa).

s kompilator primenjuje tehnike optimizovanja koje smanjuju izvršivi program, a ne njegovo vreme izvršavanja.

Optimizovati samo bitne delove programa. Programeri često nemaju jasnu predstavu koji delovi programa troše najviše resursa. Takođe, nešto što se na prvi pogled učini kao korisna optimizacija može zapravo biti potpuno beskorisna transformacija programa, koja program čini komplikovanijim, nerazumljivijim, težim za održavanje i podložnijim greškama. Stoga je tokom optimizovanja jako bitno koristiti merenje vremena izvršavanja na pažljivo odabranim test-primerima i koristiti profajler koji može da ukaže na delove programa koji ima najviše smisla optimizovati. Optimizacije za koje se empirijski ne pokaže da doprinose poboljšanju efikasnosti programa obično nema smisla primenjivati (pogotovo ako komplikuju program). Ukoliko je merenjem potvrđeno da neki deo programa

neznatno utiče na njegovu efikasnost – ne vredi unapređivati ga, bolje je da on ostane u jednostavnom i lako razumljivom obliku. Uvek se treba usredsrediti na delove programa koji troše najveći udeo vremena.[8]

Čuvene su reči Donalda Knuta: „Programeri troše enormne količine vremena razmišljajući ili brinući o brzini nekritičnih delova svojih programa i to zapravo stvara jak negativan uticaj u fazama debagovanja i održavanja. Treba da zaboravimo na male efikasnosti, recimo 97% vremena dok programiramo: prerana optimizacija je koren svih zala. Ipak, ne treba da propustimo mogućnosti u preostalih kritičnih 3%."

Izdvojiti izračunavanja koja se ponavljaju. Identično skupo izračunavanje ne bi trebalo da se ponavlja. U sledećem primeru:

```
x = x0*cos(0.01) - y0*sin(0.01);
y = x0*sin(0.01) + y0*cos(0.01);
```

vrednosti funkcije cos i sin se u originalnom programu izračunavaju po dva puta za istu vrednost argumenta. Ove funkcije su vremenski zahtevne i bolje je u pomoćnim promenljivama sačuvati njihove vrednosti pre korišćenja. Dakle, umesto navedenog, donekle je bolji naredni kôd:

```
cs = cos(0.01);
sn = sin(0.01);
x = x0*cs - y0*sn;
y = x0*sn + y0*cs;
```

Ipak, ovo je jedna od osnovnih tehnika optimizacije koje kompilator primenjuje i sasvim je realno očekivati da će kompilator ovu optimizaciju sam izvršiti. Štaviše, pošto se radi o konstantnim izrazima, njihova vrednost biće izračunata već u fazi kompilacije. Recimo i da ovakva optimizacija u opštem slučaju nije trivijalna, jer iako je `0.01` konstantni izraz, što se lako može primetiti, potrebna je napredna analiza kojom se može

[8] Ilustrujmo ovaj savet jednim Lojdovim (Samuel Loyd, 1841-1911) problemom: „Ribolovac je sakupio 1kg crva. Sakupljeni crvi imali su 1% suve materije i 99% vode. Sutra, nakon sušenja, crvi su imali 95% vode u sebi. Kolika je tada bila ukupna masa crva?" Na početku, suva materija činila je 1% od 1kg, tj. 10gr. Sutradan, vode je bilo 19 puta više od suve materije, tj. 190gr, pa je ukupna masa crva bila 200 gr.

Ako bi program činile funkcije f i g i ako bi f pokrivala 99% a funkcija g 1% vremena izvršavanja, glavni kandidat za optimizovanje bila bi, naravno, funkcija f. Ukoliko bi njen udeo u konačnom vremenu pao sa 99% na 95%, onda bi ukupno vreme izvršavanja programa bilo svedeno na 20% početnog.

Gledano drugačije, ako se funkcija f ubrza tako da se izvršava dvostruko brže, vreme izvršavanja programa se sa t smanjuje na $49,5\%t + 1\%t = 50,5\%t$ tj. i ceo program se izvršava skoro dvostruko brže. S druge strane, ako se (samo) funkcija g ubrza tako da se izvršava dvostruko brže, vreme izvršavanja programa se sa t smanjuje na $99\%t + 0,5\%t = 99,5\%t$ tj. efekat ubrzanja na vreme izvršavanja celog programa je praktično zanemariv.

zaključiti da funkcije `sin` i `cos` nemaju propratnih efekata i da će vratiti uvek istu vrednost za isti argument. Takva provera može biti ekstremno teška ili nemoguća za neke funkcije.

Kompilator GCC vrši ove optimizacije već na prvom nivou optimizacije (`-O1`), što se može proveriti u generisanom asemblerskom programu. Ovakva optimizacija daje nezanemariv dobitak samo ukoliko se modifikovani kôd izvršava veliki broj puta.

Izdvajanje koda izvan petlje. Ova preporuka je u istom duhu kao prethodna. Iz petlje je potrebno izdvojiti izračunavanja koja su ista u svakom prolasku kroz petlju. Na primer, umesto:

```
for (i=0; i < strlen(s); i++)
  if (s[i] == c)
    ...
```

daleko je bolje:

```
len = strlen(s);
for (i = 0; i < len; i++)
  if (s[i] == c)
    ...
```

U prvoj verziji koda, kroz petlju se prolazi n puta, gde je n dužina niske `s`. Međutim, u svakom prolasku kroz petlju se poziva funkcija `strlen` za argument `s` i u svakom tom pozivu se prolazi kroz celu nisku `s` (do završne nule). Zbog toga je složenost prve petlje (barem) $O(n^2)$. S druge strane, u drugoj verziji koda, funkcija `strlen` poziva se samo jednom i složenost tog poziva i petlje koja sledi zajedno može da bude $O(n)$. Dakle, ovo je primer optimizacije koja ne menja samo konstantni faktor, već snižava asimptotsku složenost programa.

Naprednom analizom bi se moglo utvrditi i da će niska `s` biti konstantna tokom izvšavanja petlje iz prvog primera, tako da je vrlo verovatno da će savremeni kompilatori izdvojiti poziv funkcije `strlen` ispred petlje. Na primer, kompilator GCC vrši ovu transformaciju kada se uključi drugi nivo optimizacije (`-O2`), a zanimljivo je da čak i zamenjuje poziv funkcije `strlen` telom ove funkcije.

Slično, od koda:

```
for (i=0; i < N; i++) {
  x = x0*cos(0.01) - y0*sin(0.01);
  y = x0*sin(0.01) + y0*cos(0.01);
  ...
  x0 = x;
```

```
  y0 = y;
}
```

bolji je naredni kôd:

```
cs = cos(0.01);
sn = sin(0.01);
for (i = 0; i < N; i++) {
  x = x0*cs - y0*sn;
  y = x0*sn + y0*cs;
  ...
  x0 = x;
  y0 = y;
}
```

Zameniti skupe operacije jeftinim. Poželjno je izraze zameniti izrazima koji su im ekvivalentni a koriste jeftinije operacije i efikasnije se izračunavaju. Na primer, uslov $\sqrt{x_1x_1 + y_1y_1} > \sqrt{x_2x_2 + y_2y_2}$ je ekvivalentan uslovu $x_1x_1 + y_1y_1 > x_2x_2 + y_2y_2$, ali je u programu daleko bolje umesto uslova `sqrt(x_1*x_1+y_1*y_1) > sqrt(x_2*x_2+y_2*y_2)` koristiti `x_1*x_1+y_1*y_1 > x_2*x_2+y_2*y_2`, jer se njime izbegava pozivanje veoma skupe funkcije `sqrt`. Slično, ukoliko je moguće dobro je izbegavati skupe trigonometrijske funkcije, umesto „većih“ tipova dobro je koristiti manje, poželjno celobrojne, itd.

Ne vršiti u fazi izvršavanja izračunavanja koja se mogu obaviti ranije. Ukoliko se tokom izvršavanja programa koriste vrednosti iz malog skupa, uštedu može da donese njihovo izračunavanje unapred i uključivanje rezultata u izvorni kôd programa. Na primer, ako se u nekom programu koriste vrednosti kvadratnog korena od 1 do 100, te vrednosti se mogu izračunati unapred i njima se može inicijalizovati konstantni niz. Ovaj pristup prihvatljiv je ako je kritični resurs vreme a ne prostor.

Napisati kritične delove kôda na asembleru. Savremeni kompilatori generišu veoma kvalitetan kôd. Ukoliko se koriste i raspoložive optimizacije, kôd takvog kvaliteta može da napiše retko koji programer. Tradicionalno, u nekim situacijama, za neke vremenski kritične delove programa, jedna opcija je bila pisanje tih delova programa na asembleru (što, naravno, podrazumeva jako dobro poznavanje asemblerskog programiranja i specifičnosti hardvera na kom će se program izvršavati). U današnjem svetu kada se sve više programira na većim nivoima apstrakcije, u višim programskim jezicima i okruženjima koji se oslanjaju na bogate biblioteke gotovog koda, ova tehnika se sve ređe i ređe koristi (osim u nekim slučajevima sistemskog programiranja).

Izvršiti paralelizaciju izračunavanja. U današnje vreme sasvim je realno da na raspolaganju imamo veliki broj procesora i računara na kojima je moguće paralelno vršiti neko izračunavanje. Iako pisanje programa koji se mogu izvršavati na taj način zahteva specifične programerske veštine i poznavanje specijalizovanih biblioteka, ono je sve češće opcija.

Naglasimo još jednom da su moderni kompilatori u stanju da samostalno primene izmene koda slične nabrojanim u mnogim situacijama. Zato treba imati na umu pre svega opšti duh navedenih primera i način razmišljanja koji ih prati. Pored toga, nije dobro uvek se oslanjati na to da će kompilator sve optimizacije izvršiti automatski. Zadatak programera je da proveri da li se to događa i ako se ne događa, da program samostalno optimizuje, ako je to moguće. Takođe, treba imati u vidu da će se program u nekim situacijama prevoditi na drugoj platformi, pomoću drugačijeg prevodioca, za koji nije sigurno kakve će optimizacije vršiti. Zato, opšti savet može biti da u delovima programa za koje programer očekuje da mogu predstavljati usko grlo, programer od početnih faza piše program tako da izbegne neefikasnosti, ukoliko taj način pisanja programa ne dovodi do komplikovanog i nerazumljivog koda. Nakon inicijalnog razvoja korektnog programa, trebalo bi pažljivo izmeriti vreme izvršavanja programa i njegovih pojedinih delova (profilisanje), utvrditi koji kritični delovi koda nisu automatski optimizovani dovoljno kvalitetno i zatim pokušati njihovu optimizaciju samostalno.

3.6 Popravljanje prostorne složenosti

Za razliku od nekadašnjih računara, na savremenim računarima memorija obično nije kritični resurs. Optimizacije se obično usredsređuju na štednju vremena ili energije, a ne prostora. Ipak, postoje situacije u kojima je potrebno štedeti memoriju, na primer, onda kada program barata ogromnim količinama podataka, kada je sâm kôd programa veliki i zauzima značajan deo radne memorije (što je danas veoma retko) ili kada se program izvršava na nekom specifičnom uređaju koji ima malo memorije.

Naravno, ključni put ka optimizaciji je ponovo optimizacija asimptotske memorijske složenosti programa tj. zamena algoritama i struktura podataka. Ipak, u nekim slučajevima popravljanje konstantnog faktora može biti značajno (na primer, razlika samo u faktoru 2 može činiti razliku da li će izračunavanje biti uspešno ili neuspešno). Često ušteda memorije zahteva specifična rešenja, ali postoje i neke ideje koje su primenljive u velikom broju slučajeva:

Koristiti najmanje moguće tipove. Za celobrojne podatke, umesto tipa `int` često je dovoljan tip `short` ili čak `char`. Za predstavljanje realnih brojeva, ukoliko preciznost nije kritična, može se, umesto tipa `double` koristiti tip `float`. Za predstavljanje logičkih vrednosti dovoljan je jedan bit a više takvih vrednosti može da se čuva u jednom bajtu (i da im se pristupa koristeći bitovske operatore).

Ne čuvati ono što može da se lako izračuna. U prethodnom delu je, u slučaju da je kritična brzina, a ne prostor, dobro da se vrednosti koje se često koriste u programu izračunaju unapred i uključe u izvorni kod programa. Ukoliko je kritična memorija, treba uraditi upravo suprotno i ne čuvati nikakve vrednosti koje se mogu izračunati u fazi izvršavanja.

Smanjenje prostorne složenosti često se postiže povećavanjem vremenske i obratno, ali postoje i mnoge situacije u kojima je dobrim rešenjem moguće popraviti istovremeno i vremensku i prostornu složenosti.

Pitanja i zadaci za vežbu

Pitanje 3.1. *Svaka instrukcija na računaru se izvršava za* $1 \cdot 10^{-9} s$. *Algoritam* A_1 *za veličinu ulaza* n *zahteva* n^2 *instrukcija, a algoritam* A_2 *zahteva* n^3 *instrukcija.*
(a) *Koliko se vremena izvršavaju algoritmi* A_1 *i* A_2 *za veličinu ulaza* 10^6*?*
(b) *Za koje veličine ulaza* n *algoritmi* A_1 *i* A_2 *mogu da završe rad za jedan minut?*

Pitanje 3.2. *Kada se kaže da važi* $f(n) \in O(g(n))$*? Kada se kaže da važi* $f(n) \in \Theta(g(n))$*? Kada se kaže da važi* $f(n) \in \Omega(g(n))$*?*
Kada kažemo da složenost algoritma A *pripada klasi* $O(f(n))$*, ili klasi* $\Theta(f(n))$*, ili klasi* $\Omega(f(n))$*?*

Pitanje 3.3. *Dokazati da važi:*
(a) $n^2 + 2n = O(n^2)$;
(b) $2^n + n = O(2^n)$;
(c) $2^n + n^3 = O(2^n)$;
(d) $3n + 2n^2 = O(n^2)$;
(e) $5^n + 2^n = O(5^n)$.

Pitanje 3.4. *Da li funkcija* $3n \log n + 5n + 100$ *pripada sledećim klasama složenosti:*
(a) $O(n)$;
(b) $O(n \log n)$;
(c) $O(n^2 \log n)$;
(d) $O(n \log^2 n)$;
(e) $O(\log n)$;
(f) $O(n + \log n)$;
(g) $O(15n)$;
(e) $O(108)$ *?*

Pitanje 3.5. *Da li funkcija* $n^6 + 2^n + 10^{10}$ *pripada sledećim klasama složenosti:*
(a) $O(n)$;
(b) $O(2^n)$;
(c) $O(n^6)$;

(d) $O(n^{10})$;
(e) $O(10^{10})$;
(f) $O(n^6 + 2^n)$;
(g) $O(2^n + 10^{10})$;
(h) $O(2^n 10^{10})$ *?*

Pitanje 3.6. *Ako $a(n)$ pripada klasi $O(n \log n)$, a $b(n)$ pripada klasi $O(n^2)$, da li onda $a(n) + b(n)$ pripada sledećim klasama:* (a) $O(n \log n)$;
(b) $O(n^2)$;
(c) $O(n \log n + n^2)$;
(d) $O(n^2 \log n)$ *?*

Pitanje 3.7. *Ako je složenost algoritma A za ulaznu vrednost n jednaka $O(n^3)$, a složenost algoritma B za ulaznu vrednost n jednaka $O(n^4)$, kolika je složenost algoritma C koji se izvršava tako što se izvršava prvo algoritam A, pa algoritam B:*
(a) $O(n^3)$;
(b) $O(n^4)$;
(c) $O(n^7)$;
(d) $O(n^{12})$ *?*

Pitanje 3.8. *U kojem vremenu može biti izvršeno pronalaženje minimuma niza od n elemenata? U kojem vremenu može biti izvršeno pronalaženje drugog po veličini elementa niza n elemenata?*

Deo II

Osnove algoritmike

Glava 4

Rekurzija

U matematici i računarstvu, rekurzija je pristup u kojem se neki pojam, objekat ili funkcija definiše na osnovu jednog ili više baznih slučajeva i na osnovu pravila koja složene slučajeve svode na jednostavnije. Na primer, pojam *predak* može se definisati na sledeći način:

bazni slučaj: roditelj osobe je predak te osobe;

rekurzivni korak: roditelj bilo kog pretka neke osobe takođe je predak te osobe.

Kaže se da je prethodna definicija rekurzivna (ili induktivna). Izostanak bilo baznog koraka (koji obezbeđuje „zaustavljanje“ primene definicije) bilo rekurzivnog koraka čini definiciju nepotpunom[1].

Korišćenjem rekurzije mnogi algoritamski problemi mogu biti rešeni elegantno. Ipak, treba imati na umu da svako izračunavanje, svaki problem koji može da se sprovede, reši korišćenjem rekurzije može da se reši i bez korišćenja rekurzije. U nekim slučajevima to se radi jednostavnim iterativnim postupcima (korišćenjem petlji), dok je u komplikovanijim slučajevima potrebno koristiti posebne strukture podataka (kao stek, o čemu će više reči biti u nastavku).

4.1 Matematička indukcija i rekurzija

Rekurzija je tesno povezana sa matematičkom indukcijom. Dokaze zasnovane na matematičkoj indukciji čine (obično trivijalni) dokazi baznog slučaja (na primer, za $n = 0$) i dokazi induktivnog koraka: pod pretpostavkom da tvrđenje važi za n dokazuje se da tvrđenje važi za $n + 1$. Rekurzivne definicije pojmova u kojima se javljaju prirodni brojevi imaju sličan oblik.

[1] Često se citira šala kako se u rečniku rekurzija može najilustrativnije objasniti tako što se pod stavkom *rekurzija* napiše: *vidi rekurzija*.

- Bazni slučaj rekurzivne definicije je slučaj koji može biti rešen bez rekurzivnog poziva;

- U rekurzivnom koraku, za vrednost n, pretpostavljamo da je definicija raspoloživa za vrednost $n-1$.

Princip dokazivanja matematičkom indukcijom je u vezi sa induktivnom definicijom skupa prirodnih brojeva — skup prirodnih brojeva je najmanji skup koji sadrži nulu i zatvoren je za operaciju sledbenika. Iz ovoga proističe da je algoritme za rad sa prirodnim brojevima ponekad moguće definisati tako da u slučaju izlaska iz rekurzije razrešavaju slučaj nule, dok u slučaju nekog broja većeg od nule (sledbenika nekog broja) rekurzivno svode problem na slučaj njegovog prethodnika. Ovakav tip rekurzije, koji direktno odgovara induktivnoj definiciji tipa podataka, naziva se *primitivna rekurzija*.

Na sličan način, moguće je induktivno predstaviti i druge tipove (skupove) podataka, što ih onda čini pogodnim za primenu primitivne rekurzije. Na primer, zapis prirodnog je ili (i) zapis dekadne cifre (bazni slučaj) ili (ii) zapis prirodnog broja na koji je nadovezan zapis dekadne cifre sa desne strane (induktivni korak). Koristeći ovakvo predstavljanje prirodnih brojeva, moguće je definisati primitivno rekurzivne funkcije takve da izlaz iz rekurzije predstavljaju jednocifreni brojevi, dok se slučaj višecifrenog broja razrešava svođenjem na broj sa odsečenom poslednjom cifrom. Ako je broj n predstavljen podatkom celobrojnog tipa, poslednju cifru možemo odrediti izrazom `n%10`, a možemo je ukloniti izrazom `n/10`. Na primer, zbir cifara broja n može se izračunati na sledeći način.

```
int zbir_cifara(unsigned n)
{
  if (n < 10)
    return n;
  return zbir_cifara(n / 10) + n % 10;
}
```

Dualni pristup – razlaganje broja na prvu cifru i na ostale cifre – u ovom slučaju nije pogodan, zbog toga što ovakvo razlaganje nije moguće sprovesti jednostavnim matematičkim operacijama.

Nisku (karaktera) moguće je definisati na sledeći način: (i) prazna niska predstavlja nisku (bazni slučaj) i (ii) dodavanjem karaktera na kraj neke niske dobija se niska. Pri ovom razmatranju, primitivnom rekurzijom moguće je definisati funkcije tako da pri izlasku iz rekurzije obrađuju slučaj prazne niske, dok slučaj neprazne niske dužine n rešavaju tako što rekurzivno razreše njen prefiks dužine $n-1$ i onda rezultat iskombinuju sa poslednjim elementom niske. Dualno razlaganje na prvi element i sufiks niske takođe je moguće, pri čemu je onda prilikom svakog rekurzivnog poziva potrebno, pored dužine niske, prosleđivati i poziciju početka sufiksa.

U nekim slučajevima, potrebno je koristiti i naprednije oblike indukcije, kakva je, na primer, *totalna indukcija*. U tom slučaju, nakon dokazivanja induktivne baze, u okviru induktivnog koraka moguće je pretpostaviti tvrđenje za sve brojeve manje od n i iz te pretpostavke dokazati tvrđenje za broj n. Slično, pored primitivno rekurzivnih funkcija, postoje i funkcije koje su *totalno (generalno) rekurzivne*. Tako je, na primer, prilikom razmatranja nekog broja n, dozvoljeno izvršiti rekurzivni poziv za bilo koji broj manji od njega (pa čak i više rekurzivnih poziva za različite prirodne brojeve manje od njega). Slično, prilikom implementacije algoritma koji radi sa nekim nizom, dozvoljeno je vršiti rekurzivne pozive za sve podnizove polaznog niza kraće od njega. Kao što će kasnije biti pokazano, obrada niza kao unije dva njegova dvostruko kraća podniza često vodi boljoj efikasnosti nego primitivno-rekurzivna obrada niza kao unije jednog elementa i podniza bez tog elementa. U nekim slučajevima upotrebe totalne rekurzije, bazni slučaj mora da pokriva više od jedne vrednosti.

4.2 Primeri primene rekurzije

U programiranju, rekurzija je tehnika u kojoj funkcija poziva samu sebe, direktno ili indirektno. Rekurzivne funkcije su pogodne za širok spektar informatičkih problema, ali pored svojih dobrih strana imaju i loše.

4.2.1 Faktorijel

Funkcija faktorijel (za prirodni broj n, vrednost $n!$ jednaka je proizvodu svih prirodnih brojeva od 1 do n) može se definisati na (primitivno) rekurzivan način:

bazni slučaj: $0! = 1$ (tj. za $n = 0$ važi $n! = 1$)

rekurzivni korak: za $n > 0$ važi: $n! = n \cdot (n-1)!$

Vrednost faktorijela se može izračunati korišćenjem petlje, ali i korišćenjem rekurzije:

```
unsigned faktorijel(unsigned n)
{
  if (n == 0)
    return 1;
  else
    return n*faktorijel(n-1);
}
```

Ovim kodom su realizovane sledeće jednakosti:

$$\begin{aligned} f(0) &= 1 \\ f(n) &= n \cdot f(n-1), \text{ za } n > 0 \end{aligned}$$

Izračunavanje vrednosti ove funkcije, na primer za vrednost 3, može se prikazati narednim nizom jednakosti (jednostavnosti radi funkcija je označena sa f).

$$f(3) = 3 \cdot f(2) = 3 \cdot 2 \cdot f(1) = 3 \cdot 2 \cdot 1 \cdot f(0) = 3 \cdot 2 \cdot 1 \cdot 1 = 6.$$

Izračunavanje vrednosti rekurzivnih funkcija vrši se uz podršku programskog steka. Ukoliko je argument funkcije, na primer, vrednost 5, onda se funkcija `f` poziva najpre za tu vrednost, a onda, rekurzivno, za vrednosti 4, 3, 2, 1, 0. Prilikom svakog poziva funkcije u stek segmentu memorije stvara se novi stek okvir — stek okvir za novu instancu funkcije `f`. U ovim stek okvirima lokalna promenljiva n imaće redom vrednosti 5, 4, 3, 2, 1, 0. Sve instance funkcije `f` koriste isti primerak koda funkcije `f` (koji se nalazi u kôd segmentu). Stek okvir svake instance funkcije `f` „pamti“ dokle je ta instanca funkcije stigla sa izvršavanjem koda (kako bi izvršavanje moglo da bude nastavljeno od te tačke kada ta instanca funkcije ponovo postane aktivna).

Pod pretpostavkom da nema prekoračenja, funkcija `faktorijel` je ispravna (zaista vraća faktorijel svog argumenta), kao što je pokazano u poglavlju 2.3.2. Funkcija `faktorijel` ima složenost $O(n)$, kao što je pokazano u poglavlju 3.3.

4.2.2 Sumiranje niza brojeva

Sumiranje niza brojeva može biti izraženo (primitivno) rekurzivno:

bazni slučaj: $\sum_{i=1}^{0} a_i = 0$ (tj. za $n = 0$ važi $\sum_{i=1}^{n} a_i = 0$)

rekurzivni korak: za $n > 0$ važi: $\sum_{i=1}^{n} a_i = \sum_{i=1}^{n-1} a_i + a_n$

Rekurzivna funkcija koja sumira elemente niza može se definisati na sledeći način:

```
float suma(float a[], unsigned n)
{
  if (n == 0)
    return 0.0f;
  else
    return suma(a, n-1) + a[n-1];
}
```

Lako se dokazuje da je navedena funkcija `suma` ispravna (zaista vraća zbir prvih n elemenata niza `a`) i da ima vremensku složenost $O(n)$. Primetimo da ova složenost ne zavisi od parametra `float a[]`, već samo od parametra `unsigned n`.

4.2.3 Fibonačijev niz

Jedan od primera koji se često koristi radi ilustracije dobrih i loših strana rekurzije je Fibonačijev niz (0,1,1,2,3,5,8,13,...) u kojem važi da je svaki naredni član jednak zbiru prethodna dva.[2] On se može definisati u vidu (totalno) rekurzivne funkcije fib (primetimo da bazni slučaj pokriva dve vrednosti – 0 i 1, a da rekurzivni korak koriste dve prethodne vrednosti niza):

bazni slučaj: $fib(0) = 0$ i $fib(1) = 1$ (tj. za $n = 0$ važi $fib(n) = 0$ i za $n = 1$ važi $fib(n) = 1$)

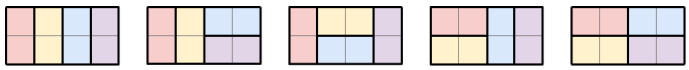
rekurzivni korak: za $n > 1$ važi: $fib(n) = fib(n-1) + fib(n-2)$

Funkcija za izračunavanje n-tog elementa Fibonačijevog niza može se definisati na sledeći način:

```
unsigned fib(unsigned n)
{
  if (n == 0 || n == 1)
    return n;
  else
    return fib(n-1) + fib(n-2);
}
```

Kasnije će biti pokazano da je ovaj pristup generisanju Fibonačijevih objekata, iako veoma jednostavan za implementaciju, veoma neefikasan i biće diskutovani mogući pravci poboljšanja.

Rekurzivna definicija Fibonačijevog niza opšte je poznata i prethodna funkcija napisana je na osnovu te definicije. Međutim, funkcije sličnog tipa često se dobijaju kada se tehnika rekurzije primeni na rešavanje nekih problema prebrojavanja. Kao ilustraciju rekurzivnog pristupa rešavanju problema, razmotrimo broj načina da se pravougaona tabla dimenzije $2 \times k$ poploča pločicama dimenzije 1×2 i 2×1. Sva popločavanja table dimenzije 2×4 prikazana su na narednoj slici (ima ih 5).



Prvo polje na tabli (polje u gornjem levom uglu) mora biti prekriveno dominom. Ona mora biti postavljena ili vertikalno ili horizontalno.

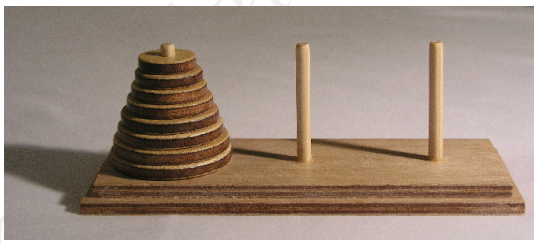
[2] Fibonačijev niz javlja se u mnogim pojavama u prirodi. Na primer, pčele se rađaju iz oplođenih, a trutovi iz neoplođenih jaja, pa trut ima samo jednu majku, ona ima dva roditelja (oca i majku), oni imaju tri roditelja (dve bake i deku), oni imaju pet roditelja (tri bake i dva deke) i tako dalje. Dakle, broj predaka trutova čini Fibonačijev niz. Fibonači je niz uveo kroz problem brojanja parova zečeva koji se pare tako da prvo potomstvo daju dva meseca posle rođenja i nakon toga svaki naredni mesec.

- Ako dominu postavimo vertikalno tada preostaje da se pokrije deo table bez prve kolone. To je problem istog oblika, ali manje dimenzije, pa se može rešiti rekurzivno.
- Dominu možemo postaviti horizontalno samo ako tabla ima bar dve kolone. Pošto i drugo polje u prvoj koloni mora biti pokriveno, ispod prve moramo postaviti još jednu horizontalnu dominu. Tada preostaje da se pokriju sve kolone table, osim prve dve, što je opet problem istog oblika, ali manje dimenzije, koji se rešava rekurzivno.

Izlaz iz ove rekurzije predstavlja tabla dimenzije 2×0, koja se može popločati samo na jedan način (ne stavljajući ni jednu dominu) i tabla dimenzije 2×1, koja se takođe može popločati samo na jedan način (stavljajući jednu vertikalnu dominu). Ako $F(k)$ označava broj popločavanja table $2 \times k$, važi da je $F(0) = F(1) = 1$ i da je $F(k) = F(k-1) + F(k-2)$, što je upravo jednačina koja definiše Fibonačijev niz.

4.2.4 Kule Hanoja

Problem kula Hanoja[3] glasi ovako: date su tri kule i na prvoj od njih 64 diska opadajućih veličina; zadatak je prebaciti sve diskove sa prve na treću kulu (koristeći i drugu) ali tako da nikada nijedan disk ne stoji iznad nekog manjeg.



Ovaj zadatak jedan je od onih koje je lakše rešiti u opštijem obliku: umesto da smatramo da na prvoj kuli ima 64 diska, smatraćemo da ih ima n.

Iterativno rešenje ovog problema veoma je kompleksno, a rekurzivno prilično jednostavno: ukoliko je $n = 0$, nema diskova koji treba da se prebacuju; inače, prebaci (rekurzivno) $n-1$ diskova sa polaznog na pomoćnu kulu (korišćenjem dolazne kule kao pomoćne), prebaci najveći disk sa polazne na dolaznu kulu i, konačno, prebaci (rekurzivno) $n-1$ diskova sa pomoćne na dolaznu kulu (korišćenjem polazne kule kao pomoćne). U nastavku je implementacija ovog rešenja:

[3]Ovaj problem je u formi autentičnog mita opisao francuski matematičar De Parvil u jednom časopisu 1884.

```
void kule(unsigned n, char polazna, char dolazna, char pomocna)
{
  if (n > 0) {
    kule(n-1, polazna, pomocna, dolazna);
    printf("Prebaci disk sa kule %c na kulu %c\n",
                                  polazna, dolazna);
    kule(n-1, pomocna, dolazna, polazna);
  }
}
```

Poziv navedene funkcije `kule(3,'A','C','B')` daje sledeći izlaz:

```
Prebaci disk sa kule A na kulu C
Prebaci disk sa kule A na kulu B
Prebaci disk sa kule C na kulu B
Prebaci disk sa kule A na kulu C
Prebaci disk sa kule B na kulu A
Prebaci disk sa kule B na kulu C
Prebaci disk sa kule A na kulu C
```

U ovom algoritmu ključna ideja je da se rešenje za n diskova rekurzijom svede na rešenje za $n-1$ diskova, ali nije svejedno kojih $n-1$ diskova. Kada se najveći disk prebaci sa polazne na dolaznu kulu, to ne onemogućava prebacivanje drugih diskova (jer je najveći), te se prirodno može iskoristiti rešenje za $n-1$. S druge strane, ne vodi do rešenja pristup da se koristi rekurzivno rešenje za $n-1$ najvećih diskova.

4.2.5 Grejovi kodovi

Binarni brojevi sa n cifara obično seređaju po veličini. Na primer, trocifreni binarni brojevi bi bili poređani u niz 000, 001, 010, 011, 100, 101, 110, 111. Postoji alternativni načinređanja binarnih brojeva koji se, u čast svog izumitelja Frenka Greja, naziva *Grejov kôd*. Grejovi kodovi se koriste za minimalizaciju logičkih funkcija u sklopu metode Karnoovih mapa, a koriste se i za korekciju grešaka u digitalnoj komunikaciji (na primer, u digitalnoj i kablovskoj televiziji). Ključno svojstvo Grejovih kodova je da se svaka dva susedna broja razlikuju tačno na jednom bitu (pri čemu to svojstvo važi i za poslednji i prvi broj u nizu). Trocifreni Grejov kôd može biti 000, 001, 011, 010, 110, 111, 101, 100. Ovo nije jedini trocifreni Grejov kôd, ali se on ipak najčešće razmatra, jer je dobijen vrlo pravilnim, sistematičnim postupkom. Naime, prva četiri broja počinju nulom, dok naredna četiri broja počinju jedinicom. Kada se sa početka prva četiri broja izbriše nula, dobija se dvocifren Grejov kôd 00, 01, 11, 10, dok se brisanje jedinice sa početka naredna četiri broja dobija isti taj Grejov kôd, ali u obratnom poretku 10, 11, 01, 00. I ovaj dvocifreni Grejov kôd može biti dobijen na isti način. Prva dva broja počinju nulom, nakon koje se javlja

jednocifreni kôd 0, 1, a druga dva broja počinju jedinicom iza koje se javlja obratan jednocifreni kôd 1, 0. Čak i za jednocifreni kôd možemo konstantovati isto. Prvi broj počinje nulom nakon koje ide Grejov kôd sa nula cifara (koji je prazan), dok drugi broj počinje jedinicom nakon koje ide isti taj kôd sa nula cifara (koji je prazan) u obratnom redosledu.

Definišimo funkciju koja određuje k-ti po redu zapis Grejovog koda sa n cifara. Taj kôd sadrži 2^n n-tocifrenih binarnih brojeva (podrazumevaćemo da je $0 \leq k < 2^n$). Nju je jednostavno definisati rekurzivno. Ako je n nula, rezultat je prazna niska. U suprotnom, treba izračunati neki element Grejovog koda sa $n-1$ cifara i zatim ga dopuniti sleva nulom ili jedinicom. Treba razlikovati slučaj elemenata u prvoj i u drugoj polovini liste kodova. Pošto ukupno ima 2^n kodova, elementi u prvoj polovini su na pozicijama $0 \leq k < 2^{n-1}$, dok su elementi u drugoj polovini na pozicijama $2^{n-1} \leq k < 2^n$.

- Kada je $0 \leq k < 2^{n-1}$, tada se vraća k-ti element Grejovog koda sa $n-1$ cifara, dopunjen početnom nulom.
- Kada je $2^{n-1} \leq k < 2^n$ tada se vraća $2^n - 1 - k$-ti element Grejovog koda sa $n-1$ cifara, dopunjen početnom jedinicom. Izrazom $2^n - 1 - k$ se pozicija k svodi u raspon $[0, 2^{n-1})$ i ujedno se obrće redosled brojeva. Naime, operacijom $k - 2^{n-1}$ vršimo redukciju intervala $[2^{n-1}, 2^n)$ na interval $[0, 2^{n-1})$. Generalno, prilikom obrtanja redosleda elemenata, svaka pozicija p u intervalu $[0, m)$ se preslikava u poziciju $m - p - 1$ (pozicija 0 se slika u $m-1$, dok se $m-1$ slika u 0). Stoga se prilikom obrtanja intervala $[0, 2^{n-1})$ pozicija $k - 2^{n-1}$ slika u $2^{n-1} - (k - 2^{n-1}) - 1$, no to je jednako $2^n - 1 - k$.

Izračunavanje stepena dvojke najjednostavnije se vrši bitovskim operacijama, pri čemu treba obratiti pažnju na potencijalna prekoračenja. Podsetimo se, šiftovanje ulevo za jednu poziciju je ekvivalentno množenju broja sa 2 (poglavlje 5.7), tako da je šiftovanje broja 1 za n pozicija ulevo ekvivalentno sa n uzastopnih množenja sa 2 i izračunava tačno vrednost 2^n (pod pretpostavkom da ne dođe do prekoračenja).

Elementi Grejovog koda mogu se jednostavno predstaviti u obliku niski karaktera.

```
#include <stdio.h>

#define MAXN 32

void grej_(unsigned n, unsigned k, unsigned i, char kod[])
{
  if (n == 0)
    return;
  if (k < (1u << (n - 1))) {
    kod[i] = '0';
```

```
    grej_(n - 1, k, i + 1, kod);
  } else {
    kod[i] = '1';
    grej_(n-1, (1ul << n) - 1 - k, i + 1, kod);
  }
}

void grej(unsigned n, unsigned k, char kod[])
{
  grej_(n, k, 0, kod);
  kod[n] = '\0';
}

int main()
{
  unsigned n;
  if (scanf("%u", &n) != 1)
    return -1;
  unsigned k;
  if (scanf("%u", &k) != 1)
    return -1;
  char kod[MAXN];
  grej(n, k, kod);
  printf("%s\n", kod);
  return 0;
}
```

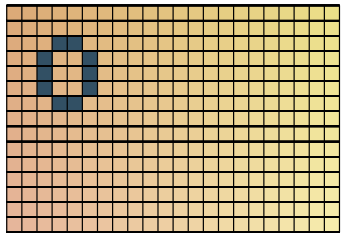
Navedeni program za ulaz `3 2` ispisuje sledeći tekst:

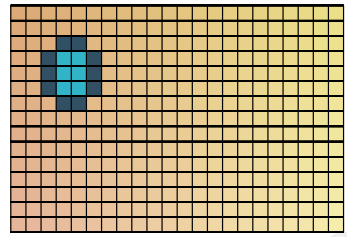
```
011
```

4.2.6 Popunjavanje konture na slici

Često postoji potreba da se sistematično obiđu i obrade elementi nekog skupa koji su međusobno povezani. To, na primer, mogu biti elementi neke matrice, istobojna polja na nekoj slici ili gradovi (koji su povezani ako između njih postoji direktan put) i slično. Obilazak svih elemenata do kojih se može stići od nekog početnog elementa često se sprovodi rekurzivnim algoritmima. Ilustrujmo takvu primenu rekurzije ovim i narednim primerom.

Programi za obradu slika obično imaju alat za popunjavanje kontura („fill“ alatka). Potrebno je izabrati piksel određene boje i čitava oblast slične boje koja je ograničena konturom druge boje ili rubom slike biće obojena zadatom bojom (pri čemu smatramo da su pikseli susedni ako imaju jednu stranicu zajedničku). Naredna slika prikazuje jednu konturu (levo) i rezultat popunjavanja ako je bila izabrana tačka unutar polazne konture (desno).





Razmotrimo pojednostavljenu varijantu problema: smatrajmo da svi pikseli imaju vrednost 0, a da pikseli koji čine konture imaju vrednost 1 i da tom istom bojom treba obojiti i unutrašnjost izabrane konture. U rekurzivnom rešenju opisanom u nastavku (to je, takozvano, *flood fill* rešenje), kreće se od zadatog piksela, on se boji i onda se ista funkcija poziva za četiri susedna piksela. Za tekući piksel se proverava da li je već obojen ili je van granica slike i ta provera omogućava izlazak iz rekurzije. Slika je predstavljena dvodimenzionim nizom `slika` koji sadrži nule i jedinice. Jednostavnosti radi pretpostavićemo da je taj niz statički alociran, a da su poznate dimenzije m i n dela koji treba obraditi, kao i koordinate početnog polja v i k.

```
#include <stdio.h>

#define MAX_DIM 50

void floodFill(int slika[MAX_DIM][MAX_DIM], int m, int n,
               int v, int k)
{
  /* da li je polje (v,k) van slike? */
  if (!(0 <= v && v < m && 0 <= k && k < n))
    return;

  /* da li je polje (v,k) vec obojeno? */
  if (slika[v][k])
    return;

  slika[v][k] = 1;
  floodFill(slika, m, n, v+1, k);
  floodFill(slika, m, n, v-1, k);
  floodFill(slika, m, n, v, k-1);
  floodFill(slika, m, n, v, k+1);
}

int main() {
  int slika[MAX_DIM][MAX_DIM], m, n, x, y;
  char c;
```

```
  /* ucitavamo dimenzije slike */
  if (scanf("%d %d\n", &m, &n) != 2)
    return -1;
  /* ucitavamo polje od kojeg krece popunjavanje */
  if (scanf("%d %d\n", &x, &y) != 2)
    return -1;
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      c = getchar();
      slika[i][j] = (c == '1'); /* '1' oznacava konturu */
    }
    c = getchar(); /* preskacemo '\n' */
  }
  /* popunjavanje konture krece od polja (x,y) */
  floodFill(slika, m, n, x, y);

  printf("\n");
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      putchar('0'+slika[i][j]);
    }
    printf("\n");
  }
  printf("\n");
  return 0;
}
```

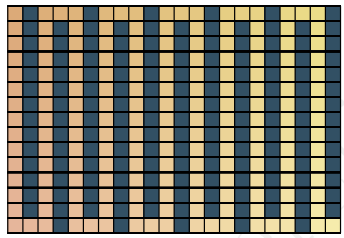
Program za naredne ulazne podatke:

```
5 8
2 2
00100100
01000100
01000100
00100100
00011111
```

daje sledeći izlaz:

```
00111100
01111100
01111100
00111100
00011111
```

Funkcija `floodFill` ima četiri rekurzivna poziva, te deluje da će lako doći do eksplozije broja polja koja se ispituju. Međutim, ne ulazi se u rekurziju za piksele koji su već obrađeni i obojeni, te je broj piksela koji se obrađuju reda $O(mn)$ i tolika je i vremenska složenost funkcije `floodFill`. Stek okvir za funkciju `floodFill` je konstante veličine, ali je pitanje koliko rekurzivnih poziva može biti aktivno u jednom trenutku, tj. koliko najviše može biti stek okvira na programskom steku. Taj broj je, naravno, ograničen odozgo vrednošću mn, a naredna slika ilustruje jedan tip situacija u kojma je broj stek okvira reda $\Theta(mn)$:



Ovaj problem može se rešiti i bez korišćenja rekurzije, ali zato uz korišćenje dodatnih struktura podataka, kao što su stek i red (videti poglavlja 6.4.5 i 6.5.6).

4.3 Složenost rekurzivnih funkcija i rekurentne jednačine

Kod rekurzivnih funkcija, vreme $T(n)$ potrebno za izračunavanje vrednosti funkcije za ulaz veličine n može se izraziti kao zbir vremena izračunavanja za sve rekurzivne pozive i vremena potrebnog za pripremu rekurzivnih poziva i objedinjavanje rezultata. Tako se, obično jednostavno, može zapisati *linearna rekurentna relacija* oblika:

$$T(n) = a_1 T(n-1) + \ldots + a_k T(n-k) + r(n), \quad n \geq k,$$

gde rekurzivna funkcija za ulaz veličine n vrši po nekoliko (a_i su prirodni brojevi) rekurzivnih poziva za ulaze veličina $n-1$, …, $n-k$, dok je $r(n)$ vreme potrebno za pripremu poziva i objedinjavanje rezultata. Ovakvom rekurentnom vezom i početnim članovima niza $T(0)$, $T(1)$, …, $T(k-1)$ u potpunosti je određen niz T.

U nekim slučajevima, iz rekurentne relacije i početnih elemenata može se eksplicitno izračunati nepoznati opšti član niza $T(n)$. U nekim slučajevima eksplicitno rešavanje jednačine nije moguće, ali se može izračunati asimptotsko ponašanje niza $T(n)$.

4.3.1 Homogena rekurentna jednačina prvog reda.

Razmotrimo jednačinu oblika

$$T(n) = aT(n-1),$$

za $n > 0$, pri čemu je data vrednost $T(0) = c$. Jednostavno se pokazuje da je rešenje ove jednačine geometrijski niz $T(n) = ca^n$.

Važna posledica ove jednačine je to da je složenost rekurzivnih funkcija koje vrše više od jednog rekurzivnog poziva dimenzije $n-1$, čak i kada se ostale operacije zanemare, eksponencijalna. Zato takva rešenja treba izbegavati kada god je to moguće.

4.3.2 Homogena rekurentna jednačina drugog reda

Razmotrimo jednačinu oblika

$$T(n) = aT(n-1) + bT(n-2),$$

za $n > 1$, pri čemu su date vrednosti za $T(0) = c_0$ i $T(1) = c_1$.

Ukoliko nisu navedeni početni uslovi, jednačina ima više rešenja. Zaista, ukoliko nizovi $T_1(n)$ i $T_2(n)$ zadovoljavaju jednačinu, tada jednačinu zadovoljava i njihova proizvoljna linearna kombinacija $T(n) = \alpha T_1(n) + \beta T_2(n)$:

$$\begin{aligned} T(n) &= \alpha T_1(n) + \beta T_2(n) \\ &= \alpha(aT_1(n-1) + bT_1(n-2)) + \beta(aT_2(n-1) + bT_2(n-2)) \\ &= a(\alpha T_1(n-1) + \beta T_2(n-1)) + b(\alpha T_1(n-2) + \beta T_2(n-2)) \\ &= aT(n-1) + bT(n-2)\ . \end{aligned}$$

S obzirom na to da i nula niz (niz čiji su svi elementi nule) trivijalno zadovoljava jednačinu, skup rešenja čini vektorski prostor.

Razmotrimo funkcije oblika t^n i pokušajmo da proverimo da li postoji broj t takav da t^n bude rešenje date jednačine. Za takvu vrednost bi važilo:

$$t^n = a \cdot t^{n-1} + b \cdot t^{n-2},$$

odnosno, posle množenja sa t^2 i deljenja sa t^n:

$$t^2 = at + b.$$

Dakle, da bi t^n bilo rešenje jednačine, potrebno je da t bude koren navedene kvadratne jednačine, koja se naziva *karakteristična jednačina* za *homogenu rekurentnu jednačinu drugog reda.*

Ako su t_1 i t_2 različiti koreni ove jednačine (oni mogu biti i kompleksne vrednosti), može se dokazati da opšte rešenje $T(n)$ može biti izraženo kao linearna kombinacija baznih funkcija t_1^n i t_2^n, tj. da je oblika

$$T(n) = \alpha \cdot t_1^n + \beta \cdot t_2^n\ ,$$

tj. da ove dve funkcije čine bazu pomenutog vektorskog prostora rešenja. Ako se želi pronaći ono rešenje koje zadovoljava zadate početne uslove (tj. zadovoljava date vrednosti $T(0) = c_0$ i $T(1) = c_1$), onda se vrednosti koeficijenata α i β mogu dobiti rešavanjem sistema dobijenog za $n = 0$ i $n = 1$, tj. rešavanjem sistema jednačina $c_0 = \alpha + \beta$, $c_1 = \alpha \cdot t_1 + \beta \cdot t_2$.

U slučaju da je t_1 dvostruko rešenje karakteristične jednačine, može se dokazati da opšte rešenje $T(n)$ može biti izraženo kao linearna kombinacija baznih funkcija t_1^n i $n \cdot t_1^n$, tj. da je oblika

$$T(n) = \alpha \cdot t_1^n + \beta \cdot n \cdot t_1^n \ .$$

Koeficijenti α i β koji određuju partikularno rešenje koje zadovoljava početne uslove, takođe se dobijaju rešavanjem sistema za $n = 0$ i $n = 1$.

Iz prethodnog sledi da je složenost rekurzivnih funkcija koje dovode do rekurentnih jednačina drugog reda eksponencijalna, pa je često dovoljno odrediti osnovu te eksponecijalne funkcije (rešavanjem karakteristične jednačine), dok se određivanje konkretnih vrednosti koeficijenata α i β manje značajno. Takve rekurzivne funkcije je dobro izbegavati i rešenja formulisati, ukoliko je moguće, tako da im složenost bude polinomska (dobar primer kako se ovo može uraditi čini Fibonačijev niz). Postoje, međutim, i rekurzivni algoritmi eksponencijalne složenosti koji rešavaju probleme za koje se ne zna da li imaju rešenja polinomske složenosti.

Primer 4.1. Neka za vreme izvršavanja $T(n)$ algoritma A (gde n određuje ulaznu vrednost za algoritam) važi $T(n+2) = 2T(n+1) + 3T(n)$ (za $n \geq 1$) i $T(1) = 5, T(2) = 19$. Složenost algoritma A može se izračunati na sledeći način. Karakteristična jednačina za navedenu homogenu rekurentnu vezu je

$$t^2 = 2t + 3,$$

i njeni koreni su $t_1 = 3$ i $t_2 = -1$. Opšti član niza $T(n)$ može biti izražen u obliku

$$T(n) = \alpha \cdot t_1^n + \beta \cdot t_2^n \ .$$

tj.

$$T(n) = \alpha \cdot 3^n + \beta \cdot (-1)^n \ .$$

Iz $T(1) = 5, T(2) = 19$ dobija se sistem:

$$\begin{aligned} \alpha \cdot 3 + \beta \cdot (-1) &= 5 \\ \alpha \cdot 9 + \beta \cdot 1 &= 19 \end{aligned}$$

čije je rešenje $(\alpha, \beta) = (2, 1)$, pa je $T(n) = 2 \cdot 3^n + (-1)^n$, odakle sledi da je $T(n) = O(3^n)$.

Primetimo da je za računanje asimptotske složenosti bilo dovoljno izračunati korene karakteristične jednačine. Veći od njih je 3 i to je dovoljno da se zaključi da složenost pripada klasi $O(3^n)$.

Primer 4.2. Neka za vreme izvršavanja $T(n)$ algoritma A (gde n određuje ulaznu vrednost za algoritam) važi $T(n+2) = 4T(n+1) - 4T(n)$ (za $n \geq 1$) i $T(1) = 6, T(2) = 20$. Složenost algoritma A može se izračunati na sledeći način. Karakteristična jednačina za navedenu homogenu rekurentnu vezu je

$$t^2 = 4t - 4$$

i njen dvostruki koren je $t_1 = 2$. Opšti član niza $T(n)$ može biti izražen u obliku

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

Iz $T(1) = 6, T(2) = 20$ dobija se sistem

$$\begin{aligned} \alpha \cdot 2 + \beta \cdot 2 &= 6 \\ \alpha \cdot 4 + \beta \cdot 8 &= 20 \end{aligned}$$

čije je rešenje $(\alpha, \beta) = (1, 2)$, pa je $T(n) = 2^n + 2 \cdot n \cdot 2^n$, odakle sledi da je $T(n) = O(n2^n)$.

U ovom primeru se u rekurentnoj jednačini javlja i jedan negativan koeficijent (koeficijent -4 uz vrednost $T(n)$) ali, kako smo videli, to ne utiče na opšti način rešavanja. Iako se jednačine sa negativnim koeficijentima ne mogu javiti direktnim modelovanjem rekurzivnih funkcija (jer se u njima ukupno vreme dobija sabiranjem vremena koje se utroši na svaki rekurzivni poziv), one mogu nastati tokom procesa rešavanja i uvođenjem raznih smena u originalne jednačine.

4.3.3 Homogena rekurentna jednačina reda k

Homogena rekurentna jednačina reda k (gde k može da bude i veće od 2) je jednačina oblika:

$$T(n) = a_1 \cdot T(n-1) + a_2 \cdot T(n-2) + \ldots + a_k \cdot T(n-k),$$

za $n > k-1$, pri čemu su date vrednosti za $T(0) = c_0$, $T(1) = c_1$, …, $T(k-1) = c_{k-1}$.

Tehnike prikazane na homogenoj jednačini drugog reda, lako se uopštavaju na jednačinu proizvoljnog reda k. Karakteristična jednačina navedene jednačine je:

$$t^k = a_1 \cdot t^{k-1} + a_2 \cdot t^{k-2} + \ldots + a_k.$$

Ako su rešenja $t_1, t_2, \ldots, t_k$ sva različita, onda je opšte rešenje polazne jednačine oblika:

$$T(n) = \alpha_1 \cdot t_1^n + \alpha_2 \cdot t_2^n + \ldots + \alpha_k \cdot t_k^n,$$

pri čemu se koeficijenti α_i mogu dobiti iz početnih uslova (kada se u navedeno opšte rešenje za n uvrste vrednosti 0, 1, …, $k-1$).

Ukoliko je neko rešenje t_1 dvostruko, onda u opštem rešenju figurišu bazne funkcije t_1^n i $n \cdot t_1^n$. Ukoliko je neko rešenje t_1 trostruko, onda u opštem rešenju figurišu bazne funkcije t_1^n, $n \cdot t_1^n$, $n^2 \cdot t_1^n$, itd.

4.3.4 Nehomogena rekurentna jednačina prvog reda

Razmotrimo jednačinu oblika

$$T(n) = aT(n-1) + b,$$

za $n > 0$, pri čemu je data vrednost $T(0) = c$. Jedan način rešavanja ovog tipa jednačina je svođenje na homogenu jednačinu drugog reda. Iz $T(1) = aT(0)+b$, sledi da je $T(1) = ac + b$. Iz

$T(n) = aT(n-1) + b$

$T(n+1) = aT(n) + b$,

sledi $T(n+1) - T(n) = (aT(n) + b) - (aT(n-1) + b) = aT(n) - aT(n-1)$ i, dalje, $T(n+1) = (a+1)T(n) - aT(n-1)$, za $n > 0$. Rešenje novodobijene homogene jednačine se može dobiti na gore opisani način (jer su poznate i početne vrednosti $T(0) = c$ i $T(1) = ac + b$).

Primer 4.3. Fibonačijev niz. Za elemente Fibonačijevog niza važi $fib(0) = 0$, $fib(1) = 1$ i $fib(n) = fib(n-1) + fib(n-2)$, za $n > 1$. Karakteristična jednačina je $t^2 = t + 1$ i njeni koreni su $\frac{1+\sqrt{5}}{2}$ i $\frac{1-\sqrt{5}}{2}$, pa je opšte rešenje oblika

$$fib(n) = \alpha \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n + \beta \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n.$$

Koristeći početne uslove, može se izračunati opšti član niza:

$$fib(n) = \frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}} \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n.$$

Videli smo da se funkcija za izračunavanje n-tog elementa Fibonačijevog niza može definisati na sledeći način:

```
unsigned fib(unsigned n)
{
  if (n == 0 || n == 1)
    return n;
  else
    return fib(n-1) + fib(n-2);
}
```

Neka $T(n)$ označava broj instrukcija koje zahteva poziv funkcije `fib` za ulaznu vrednost n. Za $n \leq 1$ važi $T(n) = c_1$, gde je c_1 neka konstanta. Za $n > 1$, važi $T(n) = T(n-1) + T(n-2) + c_2$, gde je c_2 neka konstanta (koja odgovara izvršavanju poređenja, dva oduzimanja, pripremu poziva funkcije za $n-1$ i $n-2$, jedno sabiranje i jedna naredba `return`). Iz $T(n) = T(n-1) + T(n-2) + c_2$ i $T(n+1) = T(n)+T(n-1)+c_2$, sledi $T(n+1) = 2T(n)-T(n-2)$. Karakteristična

jednačina ove jednačine je $t^3 = 2t^2 - 1$ i njeni koreni su 1, $\frac{1+\sqrt{5}}{2}$ i $\frac{1-\sqrt{5}}{2}$, pa je opšte rešenje oblika

$$T(n) = a \cdot 1^n + b \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n + c \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n ,$$

odakle sledi da je $T(n) = O(\left(\frac{1+\sqrt{5}}{2}\right)^n)$.

Primer 4.4. Kule Hanoja. Analizirajmo složenost algoritma za rešavanje Hanojskih kula. Izračunajmo broj prebacivanja diskova $T(n)$ koje opisuje navedena funkcija. Važi $T(0) = 0$ i $T(n) = 2T(n-1)+1$ (i $T(1) = 2T(0)+1 = 1$). Dakle, jednačina koja opisuje ovaj niz je nehomogena rekurentna jednačina prvog reda. Iz $T(n) - 2T(n-1) = 1 = T(n-1) - 2T(n-2)$ (za $n > 1$) sledi $T(n) = 3T(n-1) - 2T(n-2)$. Ova jednačina je homogena jednačina drugog reda i ona može biti rešena na ranije opisan način. Karakteristična jednačina je $t^2 = 3t - 2$ i njeni koreni su 2 i 1. Iz sistema

$$\begin{aligned} \alpha \cdot 1 + \beta \cdot 1 &= 0 \\ \alpha \cdot 2 + \beta \cdot 1 &= 1 \end{aligned}$$

sledi $\alpha = 1$ i $\beta = -1$, pa je

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

Razmotrimo i prostornu složenost $S(n)$ funkcije `kule`. Jedna instanca funkcije `kule` zauzima (na programskom steku) konstantan prostor c. Primetimo da se u okviru funkcije `kule` u jednom trenutku može izvršavati najviše jedan od dva rekurzivna poziva. Zato za prostornu složenost važi:

$$\begin{aligned} S(0) &= c \\ S(n) &= S(n-1) + c \end{aligned}$$

(a ne $S(n) = 2S(n-1) + c$), odakle se dobija da $S(n)$ pripada klasi $O(n)$. Na programskom steku, za ulazni argument n, može se u jednom trenutku naći najviše $n + 1$ stek okvira instanci funkcije `kule`.

4.3.5 Nehomogena rekurentna jednačina reda k

Nehomogena rekurentna jednačina reda k ($k > 0$) oblika:

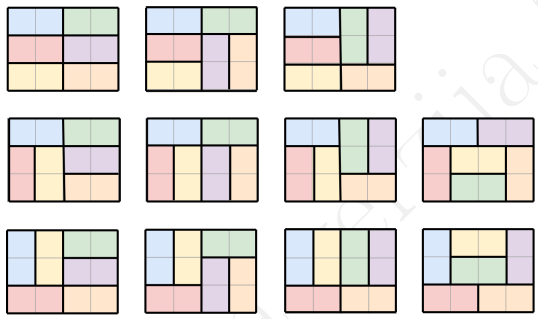
$$T(n) = a_1 \cdot T(n-1) + a_2 \cdot T(n-2) + \ldots + a_k \cdot T(n-k) + c,$$

za $n > k-1$, pri čemu su date vrednosti za $T(0) = c_0$, $T(1) = c_1$, ..., $T(k-1) = c_{k-1}$, može se rešiti svođenjem na homogenu rekurentnu jednačinu reda $k+1$, analogno gore opisanom slučaju za $k = 1$.

4.4 Uzajamna rekurzija

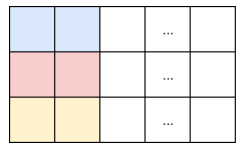
U dosadašnjim primerima, rekurzivne funkcije su pozivale same sebe direktno. Postoji i mogućnost da se funkcije međusobno pozivaju i tako stvaraju *uzajamnu rekurziju*. Uzajamna rekurzija nije samo sintaksička osobina programskog jezika, već je ponekad i veoma važna tehnika konstrukcije algoritama. Prikažimo to na jednom primeru.

Primer 4.5. Neka je potrebno da se izračuna broj različitih popločavanja pravougaone table dimenzije $3 \times k$ pravougaonim pločicama dimenzije 1×2 i 2×1, gde je k paran prirodan broj. Na narednoj slici prikazana su sva takva popločavanja za $k = 4$.

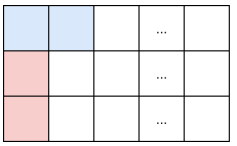


Gornje levo polje table sigurno mora da bude pokriveno dominom. Postoje dve mogućnosti da se to uradi: ta domina je postavljena ili horizontalno ili vertikalno.

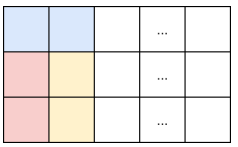
- Razmotrimo prvo slučaj kada je ta domina postavljena horizontalno. I polje ispod nje (polje u drugoj vrsti i prvoj koloni) mora da bude pokriveno dominom. I ona može biti postavljena bilo horizontalno, bilo vertikalno.
  - Ako je i ta domina postavljena horizontalno, onda je ispod nje sigurno potrebno postaviti još jednu horizontalnu dominu. Nakon postavljanja tri horizontalne domine, ostalo je da se poploča pravougaono polje dimenzije $3 \times (k-2)$ ($k-2$ je paran prirodan broj), što je problem istog oblika, ali manje dimenzije i to sugeriše da se ovaj problem može rešiti rekurzivnim pristupom.



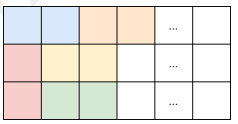
  - Ako je druga domina postavljena vertikalno, preostaje da se poploča pravougaonik dimenzije $3 \times (k-1)$ ($k-1$ je neparan prirodan broj) kojem nedostaje gornje levo teme (ono je već prekriveno).



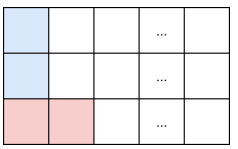
– Postavljanjem još jedne vertikalne domine pored te postavljene vertikalne domine, problem bi se ponovo sveo na pravougaonik dimenzije $3 \times (k-2)$ ($k-2$ je paran prirodan broj).



Međutim, ako polje u drugoj vrsti i drugoj koloni pokrijemo horizontalnom dominom, tada je ispod te domine neohpodno staviti još jednu horizontalnu dominu (jer polja u trećoj vrsti u drugoj i trećoj koloni ne mogu više biti pokrivene vertikalnim dominama). Takođe, u prvoj vrsti mora biti postavljena još jedna horizontalna domina (jer polje u prvoj vrsti i trećoj koloni više ne može biti pokriveno vertikalnom dominom). Tada ponovo dobijamo zadatak da pokrijemo pravougaonik kojem nedostaje jedno teme (i to pravougaonik dimenzije $3 \times (k-3)$, pri čemu je $k-3$ neparan prirodan broj).



- Primetimo i da se postavljanjem prve domine vertikalno, a zatim postavljanjem horizontalne domine ispod nje ponovo dobija zadatak u kojem je potrebno da se poploča pravougaonik bez jednog temena (i to pravougaonik dimenzije $3 \times (k-1)$, pri čemu je $k-3$ neparan prirodan broj).



Dalja ovakva analiza preti da postane previše komplikovana, ali i dosadašnja analiza sugeriše da se sve vreme pojavljuju dva oblika problema, koji se uzajamno svode jedan na drugi. Jedan je popločavanje pravougaonih tabli dimenzije $3 \times k$, za parne vrednosti k, a drugi je popločavanje pravougaonih tabli dimenzije $3 \times k$, kojima nedostaje jedan pravougaonik za neparne vrednosti k. Pretpostavićemo, kao induktivnu hipotezu, da umemo da popločavamo tj. da

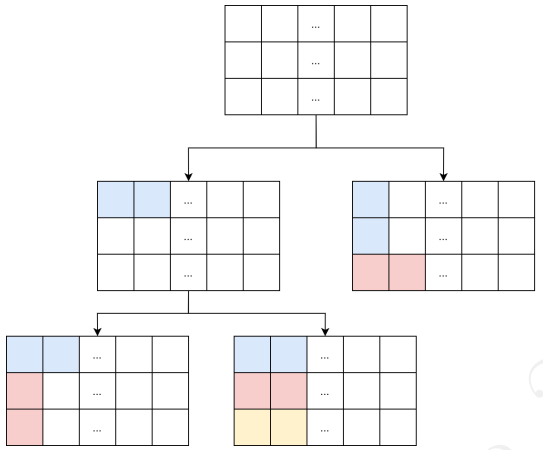
izračunamo broj mogućih načina popločavanja tabli oba gore opisana oblika, za dimenzije strogo manje od tekućih (u rekurzivnoj implementaciji, ti potproblemi će biti rešeni rekurzivnim pozivima). Imajući ovo na umu, započnimo rešavanje problema.

Neka je p_k broj načina da se prekrije čitav pravougaonik dimenzije $3 \times k$ i neka je n_k broj načina da se prekrije pravougaonik dimenzije $3 \times k$ bez donjeg levog temena (to je ujedno i broj načina da se prekrije pravougaonik te dimenzije bez gornjeg levog temena). Pokušaćemo da uspostavimo veze između ovih vrednosti.

Razmotrimo kako da popločamo čitav pravougaonik dimenzije $3 \times k$, tj. kako da izračunamo vrednost p_k. Ako je k neparan broj, popločavanje je nemoguće (takva tabla sadrži neparan broj polja, a pošto svaka domina zauzima tačno dva polja, domine ukupno zauzimaju paran broj polja). Dakle, za neparne vrednosti k važi da je $p_k = 0$. Razmotrimo slučaj kada je k paran broj. Za $k = 0$ smatramo da se prazna tabla može prekriti na jedinstven način (nepostavljanjem ni jedne domine), pa je $p_0 = 1$. Razmotrimo sada slučaj da je k paran broj veći od 0.

- Pretpostavimo da je domina u gornjem levom uglu postavljena horizontalno.
  - Ako i dominu koja pokriva polje u drugoj vrsti i prvoj koloni postavimo horizontalno, tada nemamo drugog izbora osim da polje u trećoj vrsti i prvoj koloni takođe prekrijemo horizontalnom dominom što nas dovodi do problema dimenzije $3 \times (k-2)$ koji umemo da rešimo na osnovu induktivne hipoteze (rekurzivnim pozivom).
  - Ako dominu koja pokriva polje u drugoj vrsti i prvoj koloni postavimo vertikalno tada preostaje da se poploča pravougaonik dimenzije $3 \times (k-1)$, iz koga je izbačeno jedno teme, što je problem koji umemo da rešimo na osnovu induktivne hipoteze (rekurzivnim pozivom).
- Pretpostavimo da je domina u gornjem levom uglu postavljena vertikalno. Pošto je potrebno prekriti i polje u trećoj vrsti i prvoj koloni, potrebno je preko tog polja postaviti horizontalnu dominu. Na taj način preostaje da se poploča pravougaonik dimenzije $3 \times (k-1)$ kome nedostaje jedno teme, što je problem koji umemo da rešimo na osnovu induktivne hipoteze (rekurzivnim pozivom).

Prethodna analiza prikazana je na narednoj slici i ona daje da za parne vrednosti k važi $p_k = 2n_{k-1} + p_{k-2}$:

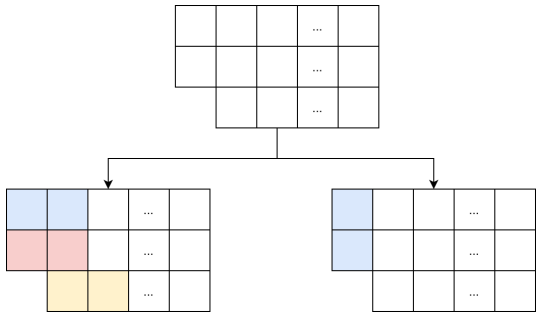


Razmotrimo sada kako da izračunamo vrednost n_k, tj. kako da popločamo pravougaonik dimenzije $3 \times k$ kojem nedostaje jedno teme (bez gubitka na opštosti može se pretpostaviti da je to donje levo teme, jer ako nije, tablu možemo simetrično preslikati), Ako je k paran broj, popločavanje dominama je nemoguće (takva tabla sadrži neparan broj polja. Dakle, za parne vrednosti k važi da je $n_k = 0$. Ako je $k = 1$, kada zaključujemo da se tabla dimenzije 3×1 bez donjeg temena može prekriti na jedinstven način (postavljanjem jedne vertikalne domine), pa je $n_1 = 1$. Razmotrimo sada slučaj da je k neparan broj veći od 1.

- Pretpostavimo da je domina u gornjem levom uglu postavljena horizontalno. Pošto polje u drugoj vrsti i prvoj koloni mora biti pokriveno i druga vrsta mora biti započeta horizontalnom dominom. Međutim, pošto i polje u trećoj vrsti i drugoj koloni mora biti popločano, i treća vrsta mora biti započeta horizontalnom dominom. Nakon postavljanja te tri horizontalne domine preostaje da se poploča pravougaonik dimenzije $3 \times (k - 2)$, kojem nedostaje donje levo teme, što je problem koji umemo da rešimo na osnovu induktivne hipoteze (rekurzivnim pozivom).

- Pretpostavimo da je domina u gornjem levom uglu postavljena vertikalno. Tada je preostalo da se poploča ceo pravougaonik dimenzije $3 \times (k - 1)$, što je problem koji umemo da rešimo na osnovu induktivne hipoteze (rekurzivnim pozivom).

Prethodna analiza prikazana je na narednoj slici i ona daje da za neparne vrednosti k važi da je $n_k = p_{k-1} + n_{k-2}$.



Prikažimo kako se na osnovu prethodnih veza može izračunati broj popločavanja table dimenzije, na primer, 3×4. To je vrednost p_4. Važi:

$$\begin{aligned} p_4 &= 2n_3 + p_2 = 2(p_2 + n_1) + p_2 = 3p_2 + 2n_1 = 3(2n_1 + p_0) + 2n_1 \\ &= 8n_1 + 3p_0 = 8 \cdot 1 + 3 \cdot 1 = 11. \end{aligned}$$

Na osnovu ovih veza moguće je definisati uzajamno rekurzivne funkcije koje izračunavaju vrednosti p_k tj. n_k.

Poželjno je da prevodilac zna definiciju funkcije pre svakog njenog poziva. Međutim, ako se dve funkcije uzajamno pozivaju, tada to nije moguće (jer u bilo kom redosledu definisanja pre poziva druge funkcije u sklopu prve funkcije definicija te druge funkcije još nije viđena). Pošto je ispravno prevođenje moguće ako se pre poziva zna samo deklaracija (prototip funkcije), rešenje može biti tako što se pre definicija funkcija navedu njihove deklaracije (dovoljno bi bilo navesti i jednu deklaraciju, ali simetrije radi ne smeta da se navedu obe).

```
/* zbog uzajamne rekurzije unapred deklarisemo funkcije */
long long parno(int);
long long neparno(int);

/* broj nacina da se dominama dimenzije 2x1 poploca
   pravougaona tabla dimenzije 3xk za neparno k i
   kojoj nedostaje donje levo polje */
long long neparno(int k)
{
  if (k == 1)
    return 1;
  return parno(k-1) + neparno(k-2);
}

/* broj nacina da se dominama dimenzije 2x1 poploca
   pravougaona tabla dimenzije 3xk za parno k  */
long long parno(int k)
{
```

```
  if (k == 0)
    return 1;
  return 2 * neparno(k-1) + parno(k-2);
}
```

Prethodne funkcije veoma su neefikasne, jer u njima dolazi do preklapanja tj. ponavljanja identičnih rekurzivnih poziva. Umesto njih, bolje je koristiti sledeće iterativno rešenje (ovo rešenje je u duhu takozvanog dinamičkog programiranja, vidi poglavlje 4.6). Razmotrimo rekurentne veze koje su implementirane ovim rekurzivnim funkcijama:

$$\begin{array}{lll} p_0 = 1, & p_k = 2n_{k-1} + p_{k-2}, \text{ za parno } k \geq 2, & p_k = 0, \text{ za neparno } k \\ n_1 = 1, & n_k = p_{k-1} + n_{k-2}, \text{ za neparno } k \geq 3, & n_k = 0, \text{ za parno } k \end{array}$$

Ove veze omogućavaju da se elementi nizova p i n izračunaju redom, odozdo naviše, od manjih ka većim vrednostima.

k	0	1	2	3	4	5	...
p_k	1	0	3	0	11	0	...
n_k	0	1	0	4	0	15	...
a_k	1	1	3	4	11	15	...

Ako niz a_k sadrži samo nenula elemente nizova p_k i n_k, tada njegove elemente možemo izračunati krenuvši od $a_0 = a_1 = 1$, a zatim primenjujući formule $a_k = 2a_{k-1} + a_{k-2}$ za parne vrednosti k i $a_k = a_{k-1} + a_{k-2}$ za neparne vrednosti k (primetimo da smo na ovaj način uzajamnu rekurziju zapravo sveli na običnu rekurziju). Nema potrebe da u memoriji istovremeno čuvamo sve elemente niza a, već samo dva poslednje izračunata. Ako promenljiva `p` čuva prethodni, a `pp` pretprethodni element, tada na osnovu njih možemo izračunati tekući element u promenljivoj `t`, a zatim se pripremiti za narednu iteraciju ažurirajući vrednosti `pp` i `p`. Složenost algoritma zasnovanog na navedenoj ideji je očigledno $O(k)$ i on može biti implementiran na sledeći način:

```
/* broj nacina da se dominama dimenzije 2x1 poploca
   pravougaona tabla dimenzije 3xk za parno k */
long long parno(int k)
{
  long long pp, p;
  pp = p = 1;
  for (int i = 2; i <= k; i++) {
    long long t;
    if (i % 2 == 0)
      t = 2 * p + pp;
    else
      t = p + pp;
```

```
    pp = p;
    p = t;
  }
  return p;
}
```

Vrednosti p_k i n_k mogu se, za svako k dobiti još efikasnije, iz eksplicitne formule koja može biti dobijena tehnikama rešavanja rekurentnih jednačina. Uvedimo dva pomoćna niza na sledeći način:

$P(i) = p_{2i}$, za $i = 0, 1, 2, \ldots$

$N(i) = n_{2i+1}$, za $i = 0, 1, 2, \ldots$

Iz veze $p_k = 2n_{k-1} + p_{k-2}$, za $k = 2i$ dobijamo $p_{2i} = 2n_{2i-1} + p_{2(i-1)}$, za $i = 1, 2, \ldots$ i

$$P(i) = 2N(i-1) + P(i-1), \text{za } i = 1, 2, \ldots \tag{4.1}$$

Slično, iz $n_k = p_{k-1} + n_{k-2}$, za $k = 2i + 1$ dobijamo $n_{2i+1} = p_{2i} + n_{2i-1}$, za $i = 1, 2, 3, \ldots$ i

$$N(i-1) = P(i-1) + N(i-2), \text{za } i = 2, 3, \ldots \tag{4.2}$$

Sada ćemo rešiti sistem rekurentnih jednačina po P i N. Iz jednakosti (4.2) dobijamo:

$P(i-1) = N(i-1) - N(i-2)$, za $i = 2, 3, 4, \ldots$

tj. (kada se indeksi pomere za 1)

$P(i) = N(i) - N(i-1)$, za $i = 1, 2, 3, \ldots$

Koristeći ovu vezu, iz jednakosti (4.1) dobijamo:

$N(i) - N(i-1) = 2N(i-1) + (N(i-1) - N(i-2))$, za $i = 2, 3, 4, \ldots$

tj.

$N(i) - 4N(i-1) + N(i-2) = 0$, za $i = 2, 3, 4, \ldots$

Karakteristična jednačina je $t^2 - 4t + 1 = 0$ i njeni koreni su $t_1 = 2 - \sqrt{3}$ i $t_2 = 2 + \sqrt{3}$, te je opšti član niza N jednak:

$N(i) = \alpha \left(2 - \sqrt{3}\right)^i + \beta \left(2 + \sqrt{3}\right)^i$

Kako je $N(0) = 1$ i $N(1) = 4$, važi:

$1 = \alpha + \beta$

$4 = \alpha \left(2 - \sqrt{3}\right) + \beta \left(2 + \sqrt{3}\right)$

odakle, rešavajući sistem po α i β, dobijamo

$\alpha = \frac{1}{2} - \frac{1}{\sqrt{3}}$

$\beta = \frac{1}{2} + \frac{1}{\sqrt{3}}$

Konačno, važi

$N(i) = (\frac{1}{2} - \frac{1}{\sqrt{3}}) \left(2 - \sqrt{3}\right)^i + (\frac{1}{2} + \frac{1}{\sqrt{3}}) \left(2 + \sqrt{3}\right)^i$

i

$n_{2i+1} = (\frac{1}{2} - \frac{1}{\sqrt{3}}) \left(2 - \sqrt{3}\right)^i + (\frac{1}{2} + \frac{1}{\sqrt{3}}) \left(2 + \sqrt{3}\right)^i$, za $i = 0, 1, 2, \ldots$

Analogno se dobijaju i vrednosti p_{2i}. Ovaj rezultat govori da je složenost funkcija iz prvog rešenja reda $O((2 + \sqrt{3})^k)$.

Navedeni primer je, očigledno, zanimljiva „mozgalica“ bez velikih praktičnih primena. Međutim, kroz proces rešavanja naučili smo nekoliko važnih tehnika i ideja, pre svega tehniku uzajamne rekurzije. Naime, može se desiti, kao u ovom primeru, da pokušaj da se problem svede na jedan ili više potproblema *istog* oblika, a manje dimenzije (što je klasična tehnika rešavanja problema rekurzijom) ne uspeva. Tada treba probati da se primeti da li se tokom svođenja javlja nekoliko *različitih* a sličnih potproblema koji se svode jedan na drugi. Ukoliko je tako, onda je umesto klasične rekurzije moguće upotrebiti uzajamnu rekurziju.

Primer 4.6. Mnoge konstrukcije u matematici i računarstvu definišu se i obrađuju korišćenjem uzajamne rekurzije. Razmotrimo, na primer, aritmetičke izraze koji sadrže prirodne brojeve i operatore `+` i `*`. Svaki izraz je niz sabiraka razdvojenih operatorom `+` (to uključuje i mogućnost da postoji samo jedan sabirak), svaki sabirak je niz činilaca razdvojenih operatorom `*` (to uključuje i mogućnost da postoji samo jedan činilac), dok je svaki činilac ili ceo broj ili izraz u zagradama. Dakle, izraz smo definisali preko sabiraka, sabirke preko činilaca, a činioce preko izraza, što predstavlja svojevrsnu uzajamnu rekurziju.

Za obradu izraza u računarstvu često se koristi tehnika poznata pod nazivom *rekurzivni spust*, koja se zasniva na uzajamnoj rekurziji. Funkcije koje čitaju aritmetički izraz i izračunavaju njegovu vrednosti mogu biti definisane na sledeći način (korišćenjem uzajamne rekurzije). Pretpostavićemo da funkcija `procitaj_sledeci_simbol` čita sledeći simbol sa ulaza (to može biti cifra, operator, neka zagrada ili kraj ulaza) i u promenljivoj `sledeci_simbol` beleži o kojoj vrsti simbola se radi, a da prilikom čitanja broja u promenljivu `vrednost_broja` smesti i njegovu brojnu vrednost. Kada se na ulazu pojavi bilo koji simbol koji ne može biti deo izraza, vraća se oznaka da je prepoznat kraj izraza, učitavanje se prekida i ispisuje se vrednost do tada učitanog izraza. Funkcija `main` poziva funkciju `izraz()` i ispisuje vrednost unetog izraza.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

typedef enum {PLUS = 0, PUTA, BROJ, OTVORENA_ZAGRADA,
              ZATVORENA_ZAGRADA, KRAJ} Simbol;
Simbol sledeci_simbol;
int vrednost_broja;

void prijavi_gresku();
void procitaj_sledeci_simbol();
int izraz();
int sabirak();
int cinilac();
```

```
void prijavi_gresku()
{
  fprintf(stderr, "Greska\n");
  exit(1);
}

void procitaj_sledeci_simbol()
{
  int c;
  /* preskacemo beline na pocetku */
  while (isspace(c = getchar()))
    ;

  if (isdigit(c)) {
    sledeci_simbol = BROJ;
    vrednost_broja = c - '0';
    while (isdigit(c = getchar()))
      vrednost_broja = 10 * vrednost_broja + (c - '0');
    ungetc(c, stdin);
  } else {
    switch(c) {
    case '+':
      sledeci_simbol = PLUS;
      break;
    case '*':
      sledeci_simbol = PUTA;
      break;
    case '(':
      sledeci_simbol = OTVORENA_ZAGRADA;
      break;
    case ')':
      sledeci_simbol = ZATVORENA_ZAGRADA;
      break;
    default:
      sledeci_simbol = KRAJ;
      break;
    }
  }
}

int izraz()
{
  int vrednost = sabirak();
  while (sledeci_simbol == PLUS) {
```

```
    procitaj_sledeci_simbol();
    vrednost += sabirak();
  }
  return vrednost;
}

int sabirak()
{
  int vrednost = cinilac();
  while (sledeci_simbol == PUTA) {
    procitaj_sledeci_simbol();
    vrednost *= cinilac();
  }
  return vrednost;
}

int cinilac()
{
  if (sledeci_simbol == BROJ) {
    procitaj_sledeci_simbol();
    return vrednost_broja;
  } else if (sledeci_simbol == OTVORENA_ZAGRADA) {
    procitaj_sledeci_simbol();
    int vrednost = izraz();
    if (sledeci_simbol != ZATVORENA_ZAGRADA)
      prijavi_gresku();
    procitaj_sledeci_simbol();
    return vrednost;
  } else
    prijavi_gresku();
}

int main()
{
  procitaj_sledeci_simbol();
  printf("%d\n", izraz());
  return 1;
}
```

Program za naredni ulaz:

```
(1+2)*(3+45).
```

daje sledeći izlaz:

```
144
```

Program za naredni ulaz:

```
(1+2.
```

daje sledeći izlaz:

```
Greska
```

4.5 Pristup „podeli i vladaj“ i master teorema

Primitivna rekurzija nad prirodnim brojevima podrazumeva da se veličina ulaza u rekurzivnom pozivu smanji za 1 tj. da se problem svede na potproblem čija je veličina ulaza za jedan manja nego za tekuću instancu funkcije. U slučaju nizova to obično znači da se u rekurzivnom pozivu obrađuje niz bez prvog ili bez poslednjeg elementa. No, u nekim problemima, kvalitetno rešenje može se dobiti svođenjem na jedan ili više istovetnih zadataka, ali višestruko manjih ulaza. Na primer, u rekurzivnom pozivu može da se obrađuje jedan ili dva potproblema čiji je ulaz dvostruko manji od polaznog i takva rešenja su često efikasnija nego kada se veličina ulaza smanji samo za jedan. Algoritmi tog tipa se nazivaju „podeli i vladaj“ (eng. divide and conquer). Ilustrujmo ovu tehniku na jednom jednostavnom primeru, a dodatni primeri biće dati u narednim poglavljima.

Primer 4.7. U rekurzivnoj funkcije za brzo stepenovanje (videti primer 2.3), problem se svodi, kada je to moguće (tj. kada je vrednost k parna) ne na slučaj $k-1$, već na slučaj $k/2$):

```
float stepen_brzo(float x, unsigned k)
{
  if (k == 0)
    return 1.0f;
  else if (k % 2 == 0)
    return stepen_brzo(x * x, k / 2);
  else
    return x * stepen_brzo(x, k - 1);
}
```

Izračunajmo složenost ove funkcije u zavisnosti od ulazne vrednosti k (jer je ona ključna u ovom primeru i na složenost ne utiče vrednost parametra x). Neka $T(k)$ označava broj instrukcija koje zahteva poziv funkcije `stepen_brzo` za ulaznu vrednost k (za $k \geq 0$). Za $k = 0$ važi $T(k) = c_1$, gde je c_1 neka konstanta (koja odgovara izvršavanju jednog poređenja i jedne naredbe `return`). Za $k > 0$, ako je k paran, važi $T(k) = T(k/2) + c_2$, a ako je neparan, važi $T(k) = T(k-1)+c_3$, gde su c_2 i c_3 neke konstante. Ako je k neparan broj, onda je $k-1$ paran,

te će u sledećoj iteraciji funkcija biti pozvana za argument $(k-1)/2$, pa je tada: $T(k) = T(k-1)+c_3 = T((k-1)/2)+c_2+c_3$. Dakle, vrednost argumenta k se, u svakom slučaju, nakon dva rekurzivna poziva smanjuje barem dvostruko. Bazni slučaj je kada je k jednako 0, pa je broj rekurzivnih poziva $O(\log k)$. Može se pokazati da je i prostorna složenost navedenog algoritma reda $O(\log k)$.

U analizi vremenske složenosti algoritama tipa „podeli i vladaj“ često se koristi naredna teorema, koja govori o asimptotskom ponašanju rešenja nehomogene rekurentne jednačine oblika[4]:

$$T(n) = aT(n/b) + cn^k.$$

Ovakva rekurentna jednačina opisuje složenost algoritama tipa „podeli i vladaj“ u kojima se rešavanje problema svodi na rešavanje nekoliko problema iste vrste ali manje veličine ulaza. Onda u navedenoj teoremi, konstanta a odgovara broju takvih potproblema, b odgovara faktoru smanjenja veličine problema, a cn^k je vreme potrebno da se polazni problem podeli na potprobleme i da se rešenja potproblema objedine u rešenje polaznog problema.

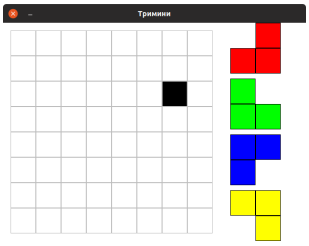
Teorema 4.1 (Master teorema o složenosti)**.** *Rešenje rekurentne relacije*

$$T(n) = aT(n/b) + cn^k \ ,$$

gde su a i b celobrojne konstante takve da važi $a \geq 1$ i $b > 1$, i c i k su pozitivne realne konstante je

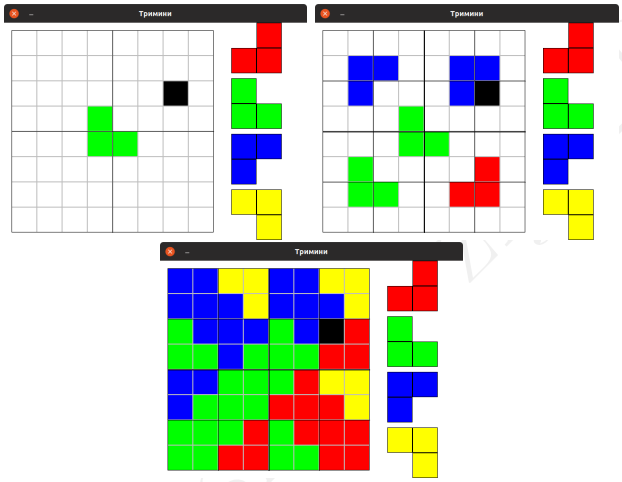
$$T(n) = \begin{cases} \Theta(n^{\log_b a}) \ , & \text{ako je } a > b^k \\ \Theta(n^k \log n) \ , & \text{ako je } a = b^k \\ \Theta(n^k) \ , & \text{ako je } a < b^k \end{cases}$$

Primer 4.8. Ilustrujmo moć tehnike podeli-pa-vladaj i primenu master teoreme o složenosti na jednoj interesantnoj mozgalici. Neka je data tabla dimenzije $n \times n$, gde je n stepen broja 2 i iz koje je izbačeno jedno polje. Potrebno je tablu pokriti triminima – oblicima koji se sastoje od tri kvadrata, prikazanim na narednoj slici.



[4]Tvrđenje teoreme važi i ako umesto člana cn^k u jednačini stoji bilo koja funkcija reda $O(n^k)$.

Nasumičnim slaganjem trimina, bez neke strategije, teško je rešiti zadatak (obično na samom kraju ostaje nekoliko polja koja se ne mogu popuniti). Postavljanjem jednog trimina u centar table, problem se svodi na četiri potproblema iste forme, ali dvostruko manje dimenzije. Da bi se to dogodilo, taj trimino treba okrenuti tako da mu nedostajući kvadrat bude baš na četvrtini table na kojoj se nalazi polje koji je izbačeno. Izlaz iz rekurzije je kada se tabla cela popuni tj. kada joj je dimenzija 1×1. Ilustrujmo primenu ovog postupka na jednoj tabli dimenzije 8×8.



Složenost ovog rekurzivnog postupka se može opisati jednačinom $T(n) = 4T(n/2) + c$, pa su koeficijenti master teoreme $a = 4$, $b = 2$ i $k = 0$. Pošto je $a > b^k$, rešenje je $\Theta(n^{\log_b a}) = \Theta(n^{\log_2 4}) = \Theta(n^2)$. Pošto je broj trimina koje treba postaviti jednak $(n^2 - 1)/3$, ovo rešenje ima optimalnu asimptotsku složenost (jer je svakako potrebno posebno postaviti svaki trimino, pa je broj operacija ne može asimptotski biti manji od n^2).

Opisani postupak može se implementirati tako da se tabla predstavlja matricom karaktera, a svaki postavljeni trimino posebnim karakterom.

Master teorema ima tri različite grane i one pokrivaju tri skupa slučajeva ponašanja algoritama. Ta teorema ovde neće biti dokazana u opštem slučaju, ali kako bi se stekla intuicija o tome zašto ona važi, prikažimo dokaze tri važna specijalna slučaja. Ta tri specijalna slučaja javljaju se u analizi mnogih realnih algoritama, a ujedno pokrivaju i sve tri grane koje se javljaju u master teoremi.

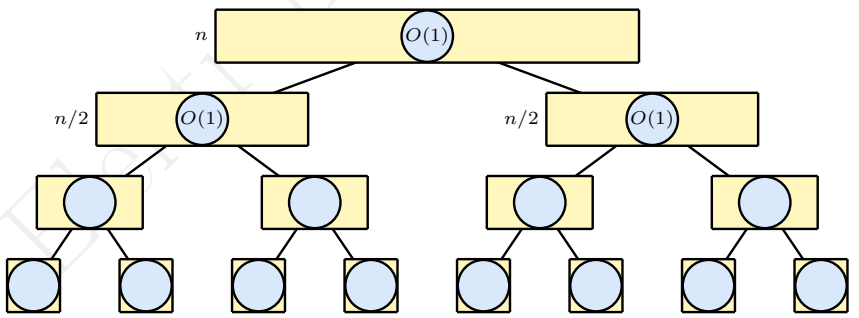
Jednačina $T(n) = 2 \cdot T(n/2) + O(1)$, $T(1) = O(1)$: Na ovakvu jednačinu odnosi se prva grana master teoreme: $a = 2$, $b = 2$, $k = 0$, pa je $a > b^k$. U ovom slučaju dobija se stablo rekurzivnih poziva koje sadrži $O(n)$ čvorova, a u svakom čvoru će se obavljati posao koji zahteva $O(1)$ operacija. Odmotavanjem

rekurentne jednačine dobijamo:

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

Ako je $n = 2^k$ dobijamo da je $n/2^k = 1$, pa pošto je na osnovu formule za zbir geometrijskog niza $2^{k-1} + \ldots + 2 + 1 = 2^k - 1$, vrednost $T(n)$ pripada klasi $\Theta(n)$. I kada n nije stepen dvojke, dobija se isto asimptotsko ponašanje (što se može dokazati ograničavanjem odozgo i odozdo stepenima dvojke).

Na slici 4.1 ilustrovana je ova jednačina. Svaki čvor prikazanog stabla ilustruje jednu instancu rekurzivne funkcije. Dimenzija pravougaonika odgovara veličini ulaza: na prvom nivou obrađuje se ulaz veličine n, na drugom nivou obrađuju se dva ulaza veličine $\frac{n}{2}$, na trećem nivou četiri ulaza veličine $\frac{n}{4}$, itd. S druge strane, dimenzija krugova prikazuje vreme utrošeno na operacije koje se izvrše u sklopu svakog rekurzivnog poziva (ne računajući dalje rekurzivne pozive). Pošto je u ovom slučaju to vreme $O(1)$, svi krugovi su male, jedinične veličine. Ukupnom utrošenom vremenu odgovara ukupna površina svih krugova. Pošto u svakom potpunom binarnom stablu listova ima za jedan više nego unutrašnjih čvorova, a na dijagramu postoji n listova, ukupno postoji $2n - 1$ krugova jedinične površine. I slika jasno ukazuje na to da ukupno utrošeno vreme pripada klasi $O(n)$.



Slika 4.1: Stablo poziva u slučaju $T(n) = 2T(n/2) + O(1)$, $T(1) = O(1)$ za $n = 8$

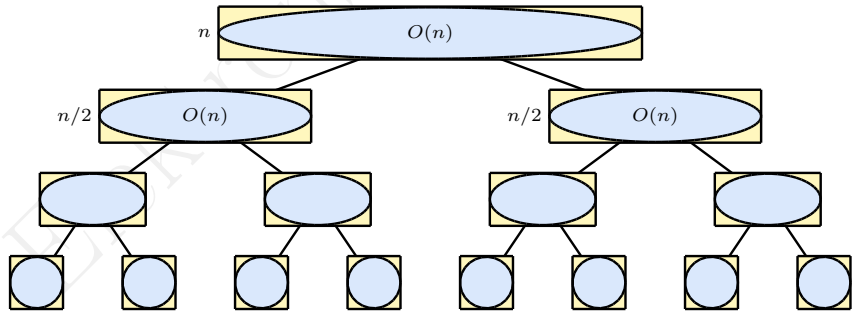
Jednačina $T(n) = 2 \cdot T(n/2) + c \cdot n$, $T(1) = O(1)$: Na ovakvu jednačinu odnosi se druga grana master teoreme: $a = 2$, $b = 2$, $k = 1$, pa je $a = b^k$. U ovom slučaju su broj čvorova i posao koji se obavlja na neki način uravnoteženi.

Odmotavanjem rekurentne jednačine dobijamo:

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

Ako je $n = 2^k$, posle $k = \log_2 n$ koraka, vrednost $n/2^k$ će dostići 1 pa će zbir biti reda veličine $n \cdot O(1) + \log_2 n \cdot c \cdot n = \Theta(n \log n)$. Isto važi i kada n nije stepen dvojke.

Na slici 4.2 ilustrovana je ova jednačina. Ponovo svaki čvor prikazanog stabla ilustruje jednu instancu rekurzivne funkcije, a dimenzije pravougaonika odgovaraju veličini ulaza. S druge strane, dimenzija elipsi prikazuje vreme utrošeno na operacije koje se izvrše u sklopu svakog rekurzivnog poziva (ne računajući dalje rekurzivne pozive). Pošto je u ovom slučaju to vreme $O(n)$, sve elipse nacrtane su tako da ispunjavaju pravougaonike. Ukupno utrošeno vreme može se predstaviti ukupnom površinom svih elipsa. Ukupna površina elipsi na svakom nivou je $O(n)$, pa pošto postoji $O(\log n)$ nivoa, utrošeno vreme pripada klasi $O(n \log n)$ (što i crtež jasno sugeriše).



Slika 4.2: Stablo poziva u slučaju $T(n) = 2T(n/2) + O(n)$, $T(1) = O(1)$ za $n = 8$

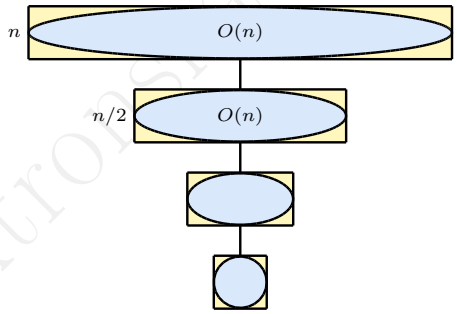
Jednačina $T(n) = T(n/2)+cn$, $T(1) = O(1)$: Na ovakvu jednačinu odnosi se treća grana master teoreme: $a = 1$, $b = 2$, $k = 1$, pa je $a < b^k$. U ovom slučaju posao koji se obavlja u čvorovima dominira nad brojem čvorova. Odmotavanjem

rekurentne jednačine dobijamo:

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

Ako je $n = 2^k$, tada je prvi član jednak $O(1)$ i, pošto na osnovu formule za zbir geometrijskog niza, važi, $1/2^{k-1}+\ldots+1/2+1 = (1-(1/2)^k)/(1-(1/2)) = 2 - 2/n$, zbir je jednak $O(1) + cn(2 - 2/n) = \Theta(n)$.

Na slici 4.2 ilustrovana je ova jednačina. Ponovo svaki čvor prikazanog stabla odgovara jednoj instanci rekurzivne funkcije, a dimenzije pravougaonika prikazuje veličinu ulaza. S druge strane, dimenzija elipsi prikazuje vreme utrošeno na operacije koje se izvrše u sklopu svakog rekurzivnog poziva (ne računajući dalje rekurzivne pozive). Sa dijagrama je jasno da kada bi se svi rekurzivni pozivi od drugog do poslednjeg nivoa složili jedan do drugog, ukupna dimenzija elipsi bila bi skoro jednaka dimenziji elipse na prvom nivou rekurzije. Zbog toga je ukupno utrošeno vreme manje od vremena koje bi se predstavilo pomoću dve elipse dimenzije n. Odatle dalje sledi da ukupno utrošeno vreme pripada klasi $O(n)$.



Slika 4.3: Stablo poziva u slučaju $T(n) = T(n/2)+O(n)$, $T(1) = O(1)$ za $n = 8$

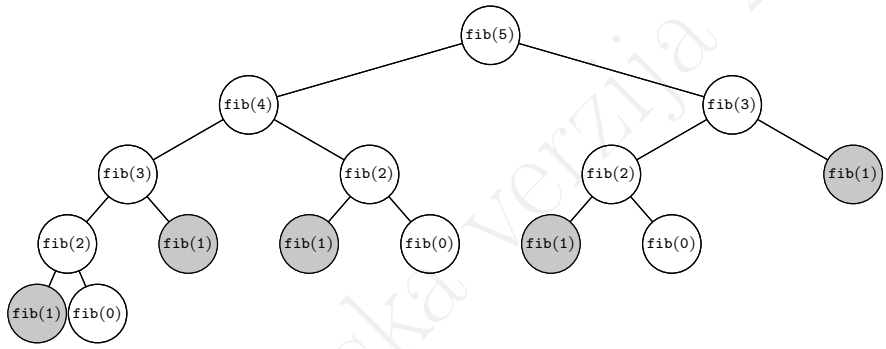
4.6 Dobre i loše strane rekurzije

Dobre strane rekurzije su (obično) čitljiv i kratak kod, jednostavan za razumevanje, analizu, dokazivanje korektnosti i održavanje. Ipak, rekurzivna rešenja imaju i mana.

Cena poziva. Prilikom svakog rekurzivnog poziva kreira se novi stek okvir i kopiraju se argumenti funkcije. Kada rekurzivnih poziva ima mnogo, to može

biti veoma zahtevno u smislu memorije (a donekle i u smislu vremena). Kako je veličina stek segmenta i na savremenim sistemima relativno mala,[5] rekurzivne funkcije mogu dovesti do prekoračenja programskog steka i prekida rada programa. Zato je u nekim situacijama poželjno rekurzivno rešenje zameniti iterativnim. Jedna opšta preporuka je da bi dubina rekurzije trebalo da raste znatno sporije od dimenzije ulaza (jer dimenzija ulaza koja se može obraditi je relativno mala čak i u slučaju kada dubina rekurzije linearno zavisi od dimenzije ulaza).

Suvišna izračunavanja. U nekim slučajevima prilikom razlaganja problema na manje potprobleme dolazi do preklapanja potproblema i do višestrukih rekurzivnih poziva za iste potprobleme.



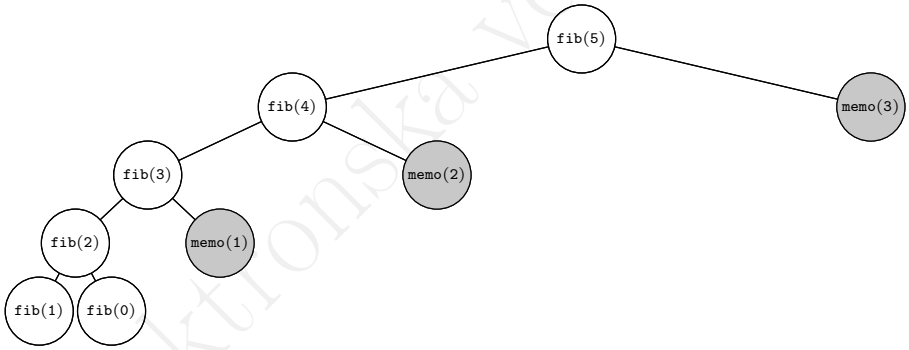
Slika 4.4: Ilustracija ponovljenih izračunavanja prilikom izvršavanja rekurzivne funkcije

Razmotrimo, na primer, izvršavanje navedene funkcije koja izračunava elemente Fibonačijevog niza za $n = 5$. U okviru tog izvršavanja, funkcija `fib` je tri puta pozvana za $n = 0$, pet puta za $n = 1$, i tako dalje. Naravno, na primer, poziv `fib(1)` je svaki put izvršavan iznova i nije korišćena vrednost koju je vratio prethodni takav poziv (slika 4.4). Zbog ovoga, izvršava se mnogo suvišnih izračunavanja i količina takvih izračunavanja u ovom primeru raste. Kako bi se izbegla suvišna izračunavanja moguće je koristiti tehniku *memoizacije*, koja podrazumeva da se u posebnoj strukturi podataka čuvaju svi rezultati već završenih rekurzivnih poziva. Pri svakom ulasku u funkciju konsultuje se ova struktura i, ako je rezultat već poznat, vraća se prethodno izračunat rezultat.

[5] Iako današnji računari imaju nekoliko gigabajta radne memorije, programi na raspolaganju obično imaju svega nekoliko megabajta za programski stek. Predefinisana veličina programskog steka može se promeniti zadavanjem odgovarajuće opcije kompilatoru, u nekoj meri, u zavisnosti od konkretnog sistema.

```
unsigned memo[MAX_FIB];

unsigned fib(unsigned n)
{
  if (memo[n]) return memo[n];
  if (n == 0 || n == 1)
    return memo[n] = n;
  else
     return memo[n] = fib(n-1) + fib(n-2);
}
```

U ovoj varijanti funkcije, poziv za svaku vrednost parametra n izvršava se tačno dva puta (osim za vrednosti 0, n i $n-1$, za koje se poziv izvršava samo jednom), pa je složenost izračunavanja linearna tj. $O(n)$. Na primer, tokom izračunavanja vrednosti `fib(n-1)` biće izračunata i memoizovana vrednost `fib(n-2)`, pa će poziv `fib(n-2)` samo pročitati i vratiti ranije izračunatu vrednost. Stablo rekurzivnih poziva memoizovane funkcije za $n = 5$ prikazano je na slici 4.5.



Slika 4.5: Ilustracija izračunavanja n-og elementa Fibonačijevog niza primenom memoizacije (pozivi u kojima se samo čita ranije izračunata vrednost označeni su sivom bojom).

Globalni niz i statičko ograničenje možemo izbeći korišćenjem pomoćne funkcije i dinamičke alokacije memorije. Pošto vremenska složenost izračunavanja vrednosti u slučaju izlaza iz rekurzije pripada klasi $O(1)$, složenost se ne menja ni ako se preskoči memoizacija za $n = 0$ i $n = 1$.

```
unsigned fib_(unsigned n, unsigned memo[])
{
  if (n == 0 || n == 1)
```

```
    return n;
  if (memo[n])
    return memo[n];
  else
    return memo[n] = fib_(n-1, memo) + fib_(n-2, memo);
}

unsigned fib(unsigned n)
{
  unsigned f;
  unsigned *memo;
  memo = calloc(n + 1, sizeof(unsigned));
  /* pretpostavljamo da je alokacija memorije uspesna */
  f = fib_(n, memo);
  free(memo);
  return f;
}
```

Drugi pristup rešavanja problema suvišnih izračunavanja naziva se *dinamičko programiranje*[6]. Prilikom rekurzivnog rešavanja vrši se poziv funkcije koja treba da izračuna rešenje polaznog problema, a zatim se taj problem svodi na jednostavnije potprobleme koji se zatim rešavaju novim pozivima iste funkcije. Potproblemi se rešavaju samo kada je to potrebno, tj. samo kada funkcija izvrši rekurzivni poziv. Kod dinamičkog programiranja, obično se unapred vrši rešavanje svih potproblema manje dimenzije, bez provere da li će njihovo rešavanje zaista biti neophodno (u praksi se često pokazuje da jeste). Na osnovu rešenih potproblema manje dimenzije, kreira se rešenje problema veće dimenzije, sve dok se ne kreira i rešenje glavnog problema. Implementacije rešenja dinamičkim programiranjem obično ne uključuju rekurzivne funkcije (mada je veza između rešenja problema veće i potproblema manje dimenzije i dalje suštinski rekurentna). Na primer, gore navedena funkcija `fib` može se zameniti iterativnom funkcijom koja od početnih elemenata niza postepeno kreira dalje elemente niza. Iako je potrebno izračunati vrednost funkcije `fib` samo za parametar n, izračunavaju se vrednosti funkcije `fib` za sve parametre manje od ili jednake n.

```
unsigned fib(unsigned n)
{
  unsigned i, ret;
  if (n == 0 || n == 1)
    return n;
```

[6] U nekim tekstovima memoizacija i dinamičko programiranje razmatraju se kao dva oblika jedne iste tehnike (jer se u oba slučaja uvodi pomoćni niz u kojem se pamte rezultati rekurzivnih poziva). Tada se memoizacija naziva dinamičko programiranje naniže, a (klasično) dinamičko programiranje naziva dinamičko programiranje naviše.

```
  unsigned* f = malloc((n + 1) * sizeof(unsigned));
  /* pretpostavljamo da je alokacija memorije uspesna */
  f[0] = 0; f[1] = 1;
  for (i = 2; i <= n; i++)
    f[i] = f[i-1] + f[i-2];
  ret = f[n];
  free(f);
  return ret;
}
```

Primetimo da za izračunavanje n-tog elementa niza nije neophodno pamtiti sve elemente niza do indeksa n već samo dva prethodna, pa se funkcija može implementirati i jednostavnije:

```
unsigned fib(unsigned n)
{
  unsigned i, fpp, fp, f;
  if (n == 0 || n == 1)
    return n;
  fpp = 0;
  fp = 1;
  for (i = 2; i <= n; i++) {
    f = fpp + fp;
    fpp = fp;
    fp = f;
  }
  return f;
}
```

4.7 Eliminisanje rekurzije

Svaku rekurzivnu funkciju je moguće implementirati na drugi način tako da ne koristi rekurziju. Ne postoji jednostavan opšti postupak za generisanje takvih alternativnih implementacija. Opšti postupak bi uključivao implementaciju strukture podataka u koju bi eksplicitno bili smeštani podaci koji se inače smeštaju na programski stek. Pošto je programski stek prilično ograničen, ovim se mogu sprečiti neke greške prekoračenja steka, ali takvi programi i dalje koriste veliku količinu memorije.

Sistematičan postupak eliminacije rekurzije postoji za neke specijalne slučajeve. Rekurzija se obično može eliminisati iz onih funkcija u kojima se tokom svakog izvršavanja rekurzivne funkcije izvrši najviše jedan rekurzivni poziv (u slučajevima kada jedan rekurzivni poziv proizvede više novih rekurzivnih poziva, eliminacija rekurzije je obično neizvodiva bez korišćenja pomoćnih struktura podataka koje emuliraju programski stek). Naročito je zanimljiv slučaj *repne rekurzije*, jer se u tom slučaju rekurzija može jednostavno eliminisati na veoma

sistematičan način[7]. Rekurzivni poziv je *repno rekurzivni* ukoliko je vrednost rekurzivnog poziva upravo i konačan rezultat funkcije, tj. nakon rekurzivnog poziva ne izvršava se nikakva dodatna naredba (uključujući ni bilo kakva ispisivanja). Na primer, u funkciji

```
float stepen_brzo(float x, unsigned k)
{
  if (k == 0)
    return 1.0f;
  else if (k % 2 == 0)
    return stepen_brzo(x * x, k / 2);
  else
    return x * stepen_brzo(x, k - 1);
}
```

prvi rekurzivni poziv je repno-rekurzivan, dok drugi nije (zato što je po izlasku iz rekurzije neophodno još pomnožiti rezultat sa x). Za rekurzivnu funkciju kažemo da je *repno-rekurzivna* ako joj je svaki rekurzivni poziv repni.

Izvršavanje repno-rekurzivne funkcije omogućava određenu optimizaciju. Pošto nakon povratka iz repnog rekurzivnog poziva neće biti više potrebni podaci koji se nalaze u tekućem stek okviru, nema potrebe za rekurzivni poziv alocirati novi stek okvir, već je moguće podatke koji odgovaraju rekurzivnom pozivu sačuvati u tekućem stek okviru. Na taj način se celokupno izvršavanje repno-rekurzivne funkcije realizuje korišćenjem samo jednog stek okvira. Mnogi kompilatori automatski vrše ovu optimizaciju. S druge strane, moguće je da programer samostalno izvrši odgovarajuću optimizaciju, tako što će eliminisati rekurziju i zameniti je petljom (u kojoj će se na mestu rekurzivnog poziva, na kraju tekuće iteracije, ažurirati vrednosti promenljivih koje odgovaraju ulaznim parametrima funkcije).

```
void r(int x)
{
  if (p(x))
    a(x);
  else {
    b(x);
    r(f(x));
  }
}
```

gde su `p`, `a`, `b` i `f` proizvoljne funkcije.

Ključni korak je da se pre rekurzivnog koraka vrednosti parametara funkcije (u ovom slučaju parametra `int x`) postave na vrednosti parametara rekurziv-

[7] Neki kompilatori (pre svega za funkcionalne programske jezike) automatski detektuju repno rekurzivne pozive i eliminišu ih.

nog poziva, a zatim da se kontrola toka izvršavanja nekako prebaci na početak funkcije. Ovo je najjednostavnije (ali ne previše elegantno) uraditi korišćenjem bezuslovnog skoka.

```
void r(int x)
{
pocetak:
  if (p(x))
    a(x);
  else {
    b(x);
    x = f(x);
    goto pocetak;
  }
}
```

Daljom analizom moguće je ukloniti bezuslovni skok i dobiti sledeći iterativni kôd.

```
void r(int x)
{
  while (!p(x)) {
    b(x);
    x = f(x);
  }
  a(x);
}
```

Primer 4.9. Demonstrirajmo tehniku uklanjanja repne rekurzije i na primeru Euklidovog algoritma.

```
unsigned nzd(unsigned a, unsigned b)
{
  if (b == 0)
    return a;
  else
    return nzd(b, a % b);
}
```

Pošto je poziv repno-rekurzivan, potrebno je pripremiti nove vrednosti parametara `a` i `b` i preneti kontrolu izvršavanja na početak funkcije.

```
unsigned nzd(unsigned a, unsigned b)
{
pocetak:
```

```
  if (b == 0)
    return a;
  else {
    unsigned tmp = a % b; a = b; b = tmp;
    goto pocetak;
  }
}
```

Daljom analizom, jednostavno se uklanja `goto` naredba i dobija identičan kôd onom koji smo ranije prikazali.

```
unsigned nzd(unsigned a, unsigned b)
{
  while (b != 0) {
    unsigned tmp = a % b; a = b; b = tmp;
  }
  return a;
}
```

Izračunavanje Grejovih kodova (poglavlje 4.2.5) takođe je zasnovano na repnoj rekurziji, pa se i ta funkcija može jednostavno definisati nerekurzivno:

```
void grej(unsigned n, unsigned k, char kod[])
{
  int i;
  i = 0;
  while (n > 0) {
    if (k < 1u << (n-1))
      kod[i] = '0';
    else {
      kod[i] = '1';
      k = (1ul << n) - 1 - k;
    }
    n--;
    i++;
  }
  kod[i] = '\0';
}
```

S obzirom na svojstva repne rekurzije, ponekad deluje poželjno rekurzivne funkcije formulisati tako da koriste iskjučivo repnu rekurziju. Na primer, pokušajmo da implementiramo repno-rekurzivnu funkciju za izračunavanje n-tog Fibonačijevog broja. U narednoj implementaciji, u svakom pozivu rekurzivne funkcije `fib_` njoj se, uz broj n, prosleđuju i dve uzastopne vrednosti Fibonačijevog niza F_k (u promenljivoj `fpp`) i F_{k+1} (u promenljivoj `fp`). Funkcija

onda na osnovu njih izračunava sledeću vrednost F_{k+2} i zatim narednom rekurzivnom pozivu prosleđuje vrednosti $n-1$, F_{k+1} i F_{k+2}. Pošto se u početnom rekurzivnom pozivu prosleđuju početna vrednost n (nazovimo je n_0) i vrednosti $F_0 = 0$ i $F_1 = 1$, sve vreme važi invarijanta da je $k + n = n_0$. Izlaz iz rekurzije je slučaj kada je $n = 0$, pa se tada tražena vrednost nalazi u promenljivoj $F_k = F_{n_0}$.

```
unsigned fib_(unsigned n, unsigned fp, unsigned fpp)
{
  if (n == 0)
    return fpp;
  return fib_(n-1, fp+fpp, fp);
}

unsigned fib(unsigned n)
{
  return fib_(n, 1, 0);
}
```

Izvršavanje broja F_5 primenom ove funkcije se može opisati sledećim nizom jednakosti:

$$\begin{aligned} \texttt{fib}(5) &= \texttt{fib\_}(5,1,0) = \texttt{fib\_}(4,1,1) = \texttt{fib\_}(3,2,1) \\ &= \texttt{fib\_}(2,3,2) = \texttt{fib\_}(1,5,3) = \texttt{fib\_}(0,8,5) = 5 \end{aligned}$$

Eliminacijom ove repne rekurzije dobijamo veoma elegantnu iterativnu implementaciju.

```
unsigned fbb(unsigned n)
{
  unsigned fp = 1, fpp = 0;
  while (n > 0) {
    int f = fp + fpp;
    fpp = fp; fp = f;
    n--;
  }
  return fpp;
}
```

Ova implementacija veoma je slična onoj koja je ranije izvedena tehnikom dinamičkog programiranja (strana 117).

Primetimo da se tokom izvršavanja repno-rekurzivnih funkcija parametri tokom rekurzivnih poziva postepeno menjaju od nekih početnih vrednosti (u primeru Fibonačijevih brojeva to su vrednosti $F_0 = 0$ i $F_1 = 1$) pa sve do krajnjih, traženih vrednosti (u primeru Fibonačijevih brojeva to su vrednosti F_n i F_{n+1}). Takva izmena promenljivih suštinski je iterativna (promenljive se na

potpuno isti način menjaju i u iterativnoj verziji, sa petljama) i odgovarajuće repno-rekurzivne funkcije često se nazivaju *iterativne funkcije*. Pošto je imperativnim jezicima, kakav je C, takva promena promenljivih karakteristična je za petlje, u njima retko kada pišu repno-rekurzivne funkcije (kada programer osmisli rešenje u kojem se promenljive menjaju na opisani način, on po običaju takvo rešenje odmah formuliše u vidu petlji).

I rekurzija koja nije repno-rekurzivna često se može ukloniti i zameniti jednostavnom iteracijom, ali to nije jednostavno uraditi na sistematičan način, kao u slučaju repne rekuzije.

Primer 4.10. Pokažimo kako se može eliminisati rekurzija iz funkcije za brzo stepenovanje navedene na početku ovog poglavlja (videti i primere 2.3 i 4.7). Postupak izračunavanja vrednosti 2^{10} tom funkcijom može se opisati sledećim nizom jednakosti:

$$2^{10} = 4^5 = 4 \cdot 4^4 = 4 \cdot 16^2 = 4 \cdot 256^1 = 4 \cdot 256 \cdot 256^0 = 1024 \cdot 1 = 1024$$

U svakom koraku proizvod $s \cdot x^k$ ima konstantnu vrednost, pri čemu je s dodatna promenljiva koja u prva dva koraka ima vrednost 1. Napišimo i kako se promenljive s, x i k menjaju tokom rekurzivnih poziva.

s	x	k
1	2	10
1	4	5
4	4	4
4	16	2
4	256	1
1024	256	0

Možemo uočiti dve vrste koraka, u zavisnosti od toga da li je k paran ili neparan broj. Ako je k paran broj, tada se vrednost x kvadrira, a k se deli sa 2. Ako je k neparan broj, tada se vrednost s množi vrednošću x, dok se vrednost k smanjuje za 1. Postupak se završava kada je $k = 0$ i konačan rezultat je tekuća vrednost promenljive s. Dakle, iterativni algoritam koji izvršava brzo stepenovanje može se implementirati na sledeći način:

```
unsigned stepen(unsigned x, unsigned k)
{
  unsigned s = 1;
  while (k > 0) {
    if (k % 2 == 0) {
      x = x * x;
      k = k / 2;
    } else {
      s = s * x;
      k--;
```

```
    }
  }
  return s;
}
```

Pitanja i zadaci za vežbu

Pitanje 4.1. *Za šta se koristi matematička indukcija, a za šta rekurzija?*

Pitanje 4.2. *Koje delove mora da ima svaka rekurzivna definicija?*

Pitanje 4.3. *Da li se za svaki rekurzivni poziv*
(a) *na programskom steku stvara novi stek okvir za tu funkciju?*
(b) *u kôd segmentu stvara nova kopija koda te funkcije?*
(c) *na hipu rezerviše novi prostor za tu funkciju?*
(d) *u segmentu podataka rezerviše novi prostor za njene promenljive?*

Pitanje 4.4. *Da li za bilo koju rekurzivnu funkciju postoji funkcija koja je njoj ekvivalentna a ne koristi rekurziju? Koju je pomoćnu strukturu podataka potrebno koristiti, da bi to bilo moguće? Obrazložiti odgovor.*

Pitanje 4.5. *Šta su dobre a šta loše strane rekurzije?*

Pitanje 4.6. *Izabrati jedan od dva ponuđena odgovora u zagradi (iterativno ili rekurzivno):*
(a) *Ukoliko se razmatra brzina izvršavanja, da li su poželjnije iterativne ili rekurzivne implementacije algoritma?*
(b) *Ukoliko se razmatra razumljivost, da li su poželjnije iterativne ili rekurzivne implementacije algoritma?*
(c) *Ukoliko se razmatra zauzeće memorije obično su da li su poželjnije iterativne ili rekurzivne implementacije algoritma?*

Pitanje 4.7. *Funkcija* `f` *je definisana sa:*

```
unsigned f(unsigned x)
{
  if (x == 0) return 1; else return x + 2*f(x - 1);
}
```

Šta je rezultat poziva `f(4)`*?*

Pitanje 4.8. *Funkcija* `f` *je definisana sa:*

```
int f(int i)
{
  if (i > 0)
    return f(i/10)+i%10;
  else
    return 0;
}
```

Šta je rezultat poziva `f(2021)`*?*

Pitanje 4.9. *Funkcija* `f` *je definisana sa:*

```
int f(int n)
{
  if (n == 0) return 0;
  if (n == 1) return 1;
  if (n % 2 == 0) return f(n / 2);
  else return f(n / 2) + 1;
}
```

Šta je rezultat poziva `f(472)`*?*

Pitanje 4.10. *Koju vrednost za dato n izračunava naredna funkcija i koja je njena prostorna složenost?*

```
unsigned long long f(unsigned n)
{
  return n == 0 ? 1 : n*f(n-1);
}
```

Pitanje 4.11. *Niz je definisan na sledeći način: $F(1) = 1$, $F(2) = 2$, $F(n+2) = 2F(n) + 3F(n+1)$, za $n > 0$. Napisati rekurzivnu funkciju koja računa vrednost n-tog elementa niza. Šta su nedostaci te funkcije? Definisati rekurzivnu i iterativnu funkciju kojom se ti nedostaci ispravljaju.*

Zadatak 4.1. *Definisati rekurzivnu funkciju koja za uneto n sa standardnog ulaza računa $f(n)$ ako je $f(n)$ definisana jednakostima $f(1) = 1$, $f(2) = 2$, $f(3) = 3$, $f(n+3) = f(n+2) + f(n+1) + f(n)$, za $n > 0$. Šta su nedostaci te funkcije? Definisati rekurzivnu i iterativnu funkciju kojom se ti nedostaci ispravljaju.*

Pitanje 4.12. *Odrediti n-ti član niza $T(n) = x \cdot T(n-1)$, $T(0) = y$.*

Pitanje 4.13. *Ako je $T(n+1) = 2 \cdot T(n) + 1$ i $T(0) = 1$ kojoj klasi pripada $T(n)$?*

Pitanje 4.14. *Algoritam A za ulaznu vrednost n poziva sebe samog za ulaznu vrednost $n-1$ i koristi još n dodatnih operacija. Izračunati složenost algoritma A.*

Pitanje 4.15. *Ako važi $T(n) = 4T(n-1) - 4T(n-2)$, koju formu ima opšti član niza T?*

Pitanje 4.16. *Ako je $T(1) = 1$, $T(2) = 7$ i $T(n+2) = -T(n+1) + 2T(n)$, koliko je $T(20)$?*

Pitanje 4.17. *Odrediti n-ti član niza* $T(n)$ *za koji važi* $T(1) = 0$, $T(2) = 5$, $T(n+2) = 5T(n+1) - 6T(n)$. *Kojoj klasi pripada* $T(n)$.

Pitanje 4.18. *Odrediti opšti član niza* $T(n)$ *za koji važi* $T(1) = 0$, $T(2) = 5$, $T(3) = 14$, $T(n+3) = 3T(n+2) - 3T(n+1) + T(n)$.

Pitanje 4.19. *Ako za vreme izvršavanja* $T(n)$ *algoritma A (gde n određuje ulaznu vrednost za algoritam) važi* $T(n+2) = 4T(n+1) - 3T(n)$ *(za* $n \geq 1$*) i* $T(1) = 1, T(2) = 2$, *onda je složenost algoritma A:*
(a) $O(n)$;
(b) $O(3^n)$;
(c) $O(4^n)$;
(d) $O(n^4)$.

Pitanje 4.20. *Funkcija* `f` *definisana je sa:*

```
void f(unsigned n)
{
  if (n > 0) { f(n-1); putchar('.'); f(n-1); }
}
```

Koliko tačkica ispisuje poziv `f(n)`*?*

Pitanje 4.21. *Funkcija* `f` *definisana je sa:*

```
void f(int n)
{
  if (n<1)
    printf("*");
  else {
    f(n-1);
    printf("----\n");
    f(n-1);
  }
}
```

Koliko zvezdica ispisuje poziv `f(n)`*?*

Pitanje 4.22. *Funkcija* `f` *definisana je sa:*

```
void f(int n)
{
  if (n<2)
    printf("* ");
  else {
    f(n-2);
    f(n-1);
    f(n-2);
```

```
  }
}
```

Koliko zvezdica ispisuje poziv `f(n)`*?*

Pitanje 4.23. *Algoritam A izvršava se za vrednost n ($n > 1$) pozivanjem algoritma B za vrednost $n - 1$, pri čemu se za svođenje problema troši jedna vremenska jedinica. Algoritam B izvršava se za vrednost n ($n > 1$) pozivanjem algoritma A za vrednost $n - 1$, pri čemu se za svođenje problema troše dve vremenske jedinice. Za ulaznu vrednost $n = 1$, algoritam A troši jednu, a algoritam B dve vremenske jedinice. Izračunati vreme izvršavanja algoritma A za ulaznu vrednost n.*

Pitanje 4.24. *Ako je $T(n) = aT(n/b)+cn^k$, gde su a i b celobrojne konstante ($a \geq 1, b \geq 1$) i c i k pozitivne konstante, kojeg je reda $T(n)$?*

Pitanje 4.25. *Kojoj klasi pripada rešenje rekurentne relacije:*
(a) $T(n) = 2T(n/2) + cn$*?*
(b) $T(n) = T(n/2) + cn$*?*
(c) $T(n) = 2T(n/2) + c$*?*
(d) $T(n) = T(n/2)+c$*?* (e) $T(n) = 4T(n/2)+cn^2$*?* (f) $T(n) = 8T(n/2)+cn^2$*?*

Pitanje 4.26. *Kako se zove rekurzivni poziv nakon kojeg nema daljih akcija?*

Pitanje 4.27. *Funkcija* `f` *definisana je na sledeći način:*

```
void f(unsigned x)
{
  if (x == 0) return;
  else if (x == 1) f(x-1);
  else if (x == 2) { printf("2"); f(x-2); }
  else { f(x - 3); printf("3"); }
}
```

Zaokružiti sve repno-rekurzivne pozive.

Pitanje 4.28. *Kako se rekurzija može eliminisati iz bilo koje repno-rekurzivne funkcije?*

Pitanje 4.29. *Definisati repno-rekurzivnu funkciju koja izračunava $n!$. Eliminisati rekurziju iz te funkcije.*

Pitanje 4.30. *Napisati nerekurzivnu implementaciju funkcije:*

```
void f(int x)
{
  if (x <= 0)
    a(x);
  else {
```

```
    b(x); f(x - 1);
  }
}
```

Pitanje 4.31. *Napisati nerekurzivnu implementaciju funkcije:*

```
void f(unsigned x)
{
  if (x < 2) a(); else { g(x + 1); h(x + 2); f(x - 2); }
}
```

Pitanje 4.32. *Napisati nerekurzivnu implementaciju funkcije:*

```
int f(int x, int a)
{
  if (x == 0)
    return a;
  return f(x-1, x*a);
}
```

Pitanje 4.33. *Napisati iterativnu verziju funkcije:*

```
void nalazenje_prostog_broja(int n)
{
  if (!prost(n)) {
    printf("Broj %d je slozen",n);
    nalazenje_prostog_broja(n+1);
  }
  else
    printf("Broj %d je prost",n);
}
```

Pitanje 4.34. *Napisati iterativnu verziju sledeće funkcije:*

```
void test(int n,int k)
{
  if (k>1) {
    test(n,k-1);
    printf("\n %d",n+k);
  }
  else
    printf("\n %d",n*n);
}
```

Zadatak 4.2. *Definisati rekurzivnu funkciju i izračunati joj složenost, koja:*
– ispisuje brojeve od 0 *do zadatog broja;*
– ispisuje brojeve od zadatog broja do 0*;*

– *izračunava sumu prvih n prirodnih brojeva, gde je n zadati broj;*
– *odredjuje maksimum datog niza brojeva;*
– *učitava n brojeva i ispisuje ih u obratnom redosledu;*
– *proverava da li u datom nizu brojeva postoji negativan broj;*
– *proverava da li su svi elementi datog niza brojeva parni;*
– *određuje zbir svih elemenata datog niza brojeva;*
– *određuje zbir svih parnih elemenata datog niza brojeva;*
– *ispisuje sve cifre datog celog broja sleva nadesno;*
– *ispisuje sve cifre datog celog broja zdesna nalevo;*
– *izračunava zbir cifara datog celog broja;*
– *izračunava broj cifara datog celog broja;*
– *izračunava istovremeno i zbir i broj cifara datog celog broja;*
– *izračunava broj parnih cifara datog celog broja;*
– *izračunava najveću cifru datog broja;*
– *izbacuje sve parne cifre iz zadatog celog broja;*
– *uklanja sva pojavljivanja date cifre iz datog broja;*
– *izbacuje svaku drugu cifru iz zadatog celog broja;*
– *posle svake neparne cifre datog broja dodaje cifru* 0*;*
– *izračunava vrednost* $a_1 + 2 \cdot a_2 + \ldots + k \cdot a_k$*, gde su* a_1 *do* a_k *cifre u zapisu zadatog celog broja sleva nalevo;*
– *obrće cifre datog celog broja;*
– *određuje binarni (oktalni, heksadekadni) zapis datog celog broja;*
– *obrće niz brojeva;*
– *obrće nisku;*
– *ispituje da li su elementi nekog niza brojeva poređani palindromski (isto sleva nadesno i zdesna nalevo);*
– *izračunava vrednost binomnog koeficienta* $\binom{n}{k}$*;*
– *izračunava skalarni proizvod dva data vektora (predstavljena nizovima dužine n).*

Zadatak 4.3. *Napisati rekurzivnu funkciju koja za zadato k i n crta „stepenice“. Svaka stepenica je širine k, a ima n stepenika. Na primer* $k = 4$*,* $n = 3$*:*

```
****
    ****
        ****
```

Izračunati vremensku složenost algoritma.

Zadatak 4.4. *Napisati rekurzivnu funkciju koja za dato n iscrtava trougao kakav je na slici prikazan za* $n = 3$*:*

(a)
```
+
++
+++
```

(b)
```
  +
 +++
+++++
```

(c)
```
 +
+++
+++++
```

(d)
```
+
++
+
```

Zadatak 4.5. *Data je matrica koja sadrži samo nule i jedinice. Definišimo ostrvo kao povezanu oblast jedinica tj. oblast koja se sastoji samo od jedinica takvu da se od bilo kojeg polja na ostrvu može stići do bilo kojeg drugog polja, krećući se u 8 dopuštenih smerova. Definisati rekurzivnu funkciju koja određuje broj ostrva u toj matrici. Na primer, u narednoj matrici postoje 3 ostrva.*

```
0011100001
0011110011
0001110100
1000000000
1100000000
```

Zadatak 4.6. *Definisati rekurzivnu funkciju koja razmenjuje prvih k i poslednjih $n-k$ elemenata niza dužine n (gde je $k \leq n$). Na primer, ako je niz $1,2,3,4,5,6,7$ i $k=3$, rezultat je niz $4,5,6,7,1,2,3$. Odrediti složenost funkcije. Ukloniti rekurziju iz definisane funkcije.*

Zadatak 4.7. *Definisati rekurzivnu funkciju koja na osnovu infiksnog i prefiksnog obilaska binarnog stabla određuje njegov postfiksni obilazak (obilasci stabla opisani su u poglavlju 6.6).*

Zadatak 4.8. *Morzeov niz se definiše na sledeći način. Prvi element je 1, drugi se dobija logičkom negacijom prvog NOT(1) = 0, treći i četvrti logičkom negacijom prethodna dva NOT(10) = 01, naredna četiri logičkom negacijom prva četiri NOT(1001) = 0110 itd. Niz je, dakle, 1001011001101001.... Definisati rekurzivnu funkciju koja određuje n-ti član ovog niza. Eliminisati rekurziju iz dobijene funkcije.*

Zadatak 4.9. *Definisati rekurzivnu funkciju koja za dati prirodan broj n određuje zbir svih brojeva koji se mogu dobiti izbacivanjem nekih cifara broja n. Na primer, za broj 123 rezultat je $123+12+13+23+1+2+3+0$. Uputstvo: razložiti broj na njegov prefiks i poslednji cifru i rekurzivno rešiti problem za prefiks. Eliminisati rekurziju iz definisane funkcije.*

Zadatak 4.10. *Definisati rekurzivnu funkciju koja određuje koliko prirodnih brojeva iz intervala $[0,n]$ ne sadrže cifru 3 u svom dekadnom zapisu. Uputstvo: razložiti interval na podintervale oblika $[0,99...99]$, ..., $[100...000,199...99]$, ... $[n_m00...00,n]$, gde je n_m početna cifra broja n. Eliminisati rekurziju iz definisane funkcije.*

Zadatak 4.11. *Definisati rekurzivnu funkciju koja prikazuje sve načine da se tabla dimenzije $2 \times k$ poploča dominama dimezije 1×2 i 2×1. Na primer, za $k=4$ funkcija treba da ispiše sledeća popločavanja (jedno ispod drugog).*

```
||||    ||--    |--|    --||    ----
||||    ||--    |--|    --||    ----
```

Glava 5

Osnovni algoritmi

U ovoj glavi biće predstavljeni neki od osnovnih algoritama koji se odnose na poređenje, poredak, pretragu, sortiranje i izračunavanje.

5.1 Poređenje i poredak

U mnogim problemima potrebno je uporediti dva objekta. U nekim situacijama potrebno je proveriti da li su dva objekta jednaka, a u nekim da li je jedan manji (ili veći) od drugog. Za poređenje vrednosti osnovnih, brojevnih tipova na raspolaganju su operatori `==`, `!=`, `<`, `>`, `<=`, `>=` sa odgovarajućim, uobičajenim matematičkim značenjem. Poređenje vrednosti složenih tipova svodi se na poređenje vrednosti osnovnih tipova.

5.1.1 Relacija jednakosti

Relacija jednakosti je relacija ekvivalencije: ona je refleksivna, simetrična i tranzitivna. Relacioni operator jednakosti (`==`), nad osnovnim, brojevnim tipovima (na primer, `int`) zadovoljava ove uslove.[1] Ovaj operator može se koristiti za proveru jednakosti dve vrednosti osnovnih tipova, ali za druge tipove to nije moguće. Na primer, provera jednakosti niski svodi se na poređenje pojedinačnih

[1] Za brojeve u pokretnom zarezu (ako se sledi standard IEEE 754, što standard jezika C ne propisuje), ovo važi samo za skup vrednosti bez pozitivne i negativne vrednosti „not-a-number“ (`NaN`). Naime, izraz `NaN==NaN` nije tačan. Dodatno, treba naglasiti da se svojstva relacije ekvivalencije odnose samo na vrednosti koje pripadaju istim tipovima. Na primer, nakon naredbe `float x = 0.1;`, promenljiva `x` ima (na nekom sistemu) vrednost `0.100000001490116119384765625`, a nakon naredbe `double x = 0.1;`, promenljiva `x` ima (na istom tom sistemu) vrednost `0.1000000000000000055511151231257827021181583404541015625`. Treba, dakle, imati na umu i da su rezultati operacija (čak i jednostavnih dodela) nad brojevima u pokretnom zarezu često dobijeni zaokruživanjem. Zbog toga, vrednosti dva izraza mogu biti različite i kada su vrednosti odgovarajućih izraza nad realnim brojevima jednake. Drugim rečima, treba uvek imati na umu da su matematička pravila za brojeve u pokretnom zarezu drugačija od matematičkih pravila koja važe za realne brojeve.

karaktera, kao u narednoj funkciji (koja vraća `1` ako su zadate niske jednake, a 0 inače).

```
int jednake_niske(const char *a, const char *b)
{
  while (*a == *b) {
    if (*a == '\0') return 1;
    a++;
    b++;
  }
  return 0;
}
```

U navedenoj funkciji, kvalifikator `const` osigurava da sadržaj na koji ukazuju pokazivači `a` i `b` ne može i neće biti promenjen u okviru funkcije `jednake_niske`. Korist od ovog kvalifikatora je dvostruka – on sprečava nehotične greške u implementaciji funkcije u kojoj se koristi, ali i govori korisniku te funkcije (čak iako on ne poznaje njenu definiciju već samo deklaraciju) da može biti siguran da sadržaj na koji ukazuje pokazivač neće biti menjan. Zbog ovih koristi, dobra je praksa koristiti kvalifikator `const` u deklaraciji parametara kad god je to moguće i prirodno (kao u navedenom slučaju – provera jednakosti niski ne treba da menja te niske).[2]

Jednakost niski moguće je proveriti i bibliotečkom funkcijom `strcmp` (deklarisanom u zaglavlju `string.h`) koja vraća `0` ako i samo ako su dve zadate niske jednake (o njoj će biti reči i u daljem tekstu).

Za dve vrednosti tipa neke strukture, provera jednakosti svodi se na proveru jednakosti svih članova pojedinačno ili možda na neki drugi način. Na primer, dva razlomka nisu jednaka samo ako su im i imenilac i brojilac jednaki, nego i u nekim drugim slučajevima. Ako je struktura `razlomak` zadata na sledeći način

```
typedef struct razlomak {
  int brojilac;
  int imenilac;
} razlomak;
```

onda se jednakost dva razlomka (definisanih vrednosti) može ispitati narednom funkcijom:

[2] Deklaracija `const char *a` može da se zapiše i u obliku `char const *a`. Naizgled sličan tip, `char * const`, bitno je drugačiji – to je je tip konstantnog pokazivača na `char *`, što znači da vrednost pokazivača ne može da se menja, ali karakter na koji ukazuje može. Za parametar funkcije moguće je koristiti i konstantni pokazivač na konstantni karakter `const char * const` (mada je umesto toga prirodnije i jednostavnije kao parametar koristiti tip `char`).

```
int jednaki_razlomci(razlomak a, razlomak b)
{
  return
    (a.imenilac * b.brojilac == b.imenilac * a.brojilac);
}
```

ili, kako bi se umesto čitavih struktura u funkciju prenosile samo njihove adrese:

```
int jednaki_razlomci(const razlomak *a, const razlomak *b)
{
  return
    (a->imenilac * b->brojilac == b->imenilac * a->brojilac);
}
```

Primetimo da u prethodnoj funkciji postoji opasnost od nastanka prekoračenja, pa je treba koristiti veoma obazrivo.

5.1.2 Relacije poretka

Relacija poretka je relacija koja je refleksivna, antisimetrična i tranzitivna. Takva je, na primer, relacija $\leq$ nad skupom prirodnih brojeva. Slično, relacioni operatori `<=` i `>=` nad osnovnim, brojevnim tipovima određuju relacije poretka. *Relacija strogog poretka* je relacija koja je antirefleksivna, antisimetrična i tranzitivna. Takva je, na primer, relacija $<$ nad skupom prirodnih brojeva. Slično, relacioni operatori `<` i `>` nad osnovnim, brojevnim tipovima određuju relacije strogog poretka.[3] Vrednosti osnovnih, brojevnih tipova mogu, međutim, da se porede i na neki drugi način – na primer, celi brojevi mogu se porediti prema zbiru svojih cifara. Takođe, za druge tipove potrebno je implementirati funkcije koje vrše poređenje a one se obično zasnivaju na poređenju za jednostavnije tipove. Na primer, naredna funkcija poredi dvoslovne oznake država po standardu ISO 3166[4]. Ona vraća vrednost -1 ako je prvi kôd manji, 0 ako su zadati kodovi jednaki i 1 ako je drugi kôd manji:

```
int poredi_kodove_drzava(const char *a, const char *b)
{
  if (a[0] < b[0])
    return -1;
  if (a[0] > b[0])
    return 1;
```

[3] Ako se sledi standard IEEE 754 za brojeve u pokretnom zarezu, ovo važi samo za skup vrednosti bez pozitivne i negativne vrednosti „not-a-number“ (`NaN`).

[4] Svrha standarda ISO 3166 je definisanje međunarodno priznatih dvoslovnih kodova za države ili neke njihove delove. Kodovi su sačinjeni od po dva slova engleskog alfabeta. Koriste se za oznaku nacionalnih internet domena, od strane poštanski organizacija, i dr. Dvoslovna oznaka za Srbiju je „rs“, za Portugaliju „pt“, za Rusiju „ru“, itd.

```
  if (a[1] < b[1])
    return -1;
  if (a[1] > b[1])
    return 1;
  return 0;
}
```

Parovi karaktera se, dakle, mogu porediti tako što se najpre porede prvi karakteri u parovima, a zatim, ako je potrebno, drugi karakteri. Slično se mogu porediti i datumi opisani narednom strukturom:

```
typedef struct datum {
  unsigned dan;
  unsigned mesec;
  unsigned godina;
} datum;
```

Prvo se porede godine – ako su godine različite, redosled dva datuma može se odrediti na osnovu njihovog odnosa. Ako su godine jednake, onda se prelazi na poređenje meseci. Na kraju, ako su i meseci jednaki, prelazi se na poređenje dana. Naredna funkcija implementira ovaj algoritam i vraća -1 ako je prvi datum pre drugog, 1 ako je drugi datum pre prvog i 0 ako su jednaki.

```
int poredi_datume(const datum *d1, const datum *d2)
{
  if (d1->godina < d2->godina)
    return -1;
  if (d1->godina > d2->godina)
    return 1;
  if (d1->mesec < d2->mesec)
    return -1;
  if (d1->mesec > d2->mesec)
    return 1;
  if (d1->dan < d2->dan)
    return -1;
  if (d1->dan > d2->dan)
    return 1;
  return 0;
}
```

Može se napisati i jedinstven logički izraz kojim se proverava da li je prvi datum ispred drugog:

```
int datum_pre(const datum *d1, const datum *d2)
{
```

```
  return
    d1->godina < d2->godina ||
    (d1->godina == d2->godina && d1->mesec < d2->mesec) ||
    (d1->godina == d2->godina && d1->mesec == d2->mesec &&
     d1->dan < d2->dan);
}
```

Generalno, torke sa fiksnim jednakim brojem elemenata mogu se poreditи tako što se najpre porede njihovi prvi elementi, zatim, ako je potrebno, njihovi drugi elementi i tako dalje, sve dok se ne naiđe na neki različit par elemenata, na osnovu kojeg se određuje poredak. Dakle, relacija poređenja pojedinačnih elemenata može se proširiti na relaciju poređenja n-torki elemenata tj. ako je na skupu X (na primer, na skupu karaktera) definisana relacija poretka $\prec$, onda se može definisati i relacija poretka $\prec^l$ na skupu X^n (na primer, na niskama karaktera dužine n). Štaviše, relacija poretka nad pojedinačnim elementima skupa X može se proširiti i na skup $X^* = \bigcup_{n=0}^{+\infty} X^n$, tj. može se proširiti i na skup svih torki svih dužina. Poređenje se ponovo vrši redom i čim se naiđe na prvu poziciju na kojoj se u dve torke nalazi različit element, na osnovu njega određuje se poredak torki. Kada su torke različite dužine, tada se može desiti da se dođe do kraja jedne od njih. Ako se istovremeno došlo i do kraja druge, tada su torke jednake, a ako nije, tada je kraća torka prefiks one duže. Tada se smatra da je kraća torka manja (ide pre one duže torke). Ovako definisana relacija poretka naziva se *leksikografski poredak* (jer se koristi i u poretku odrednica u leksikonima i rečnicima).

Za niske (potencijalno različitih dužina) može se definisati leksikografski poredak koji je zasnovan na poređenju karaktera i odgovarajuća funkcija koja vraća vrednost -1 ako je prva niska manja, 0 ako su zadate niske jednake i 1 ako je druga niska manja:

```
int poredi_niske(const char *a, const char *b)
{
  while (*a == *b) {
    if (*a == '\0') return 0;
    a++;
    b++;
  }
  if (*a < *b)
    return -1;
  else
    return 1;
}
```

U navedenoj funkciji, niske se porede karakter po karakter i ako na jednoj poziciji postoji razlika, konstatuje se da li je manja prva ili druga niska. Ako se došlo do kraja niske i nije pronađena razlika ni na jednoj poziciji, onda su

niske jednake. Funkcija opisanog ponašanja (nije nužno definisana na navedeni način) raspoloživa je i u okviru standardne biblioteke, a deklarisana u datoteci zaglavlja `string.h`:

```
int strcmp(const char *str1, const char *str2)
```

Niske mogu da se porede i na neki drugi način, na primer, samo po dužini:

```
int poredi_niske(const char *a, const char *b)
{
  return (strlen(a)-strlen(b));
}
```

Dva razlomka (čiji su imenioci pozitivni) mogu da se porede sledećom funkcijom (pod pretpostavkom da smo sigurni da prilikom množenja neće doći do prekoračenja):

```
int poredi_razlomke(const razlomak *a, const razlomak *b)
{
  return (a->brojilac*b->imenilac - b->brojilac*a->imenilac);
}
```

5.1.3 Sortiranost niza

Ako je zadata relacija poretka (ili strogog poretka), onda se može proveriti da li je niz *uređen* (ili *sortiran*) u skladu sa tom relacijom. Na primer, naredna funkcija proverava da li je niz tipa `int` uređen u skladu sa relacijom `<=` (parametar `n` daje broj elemenata niza i takav je obično tipa `size_t`):[5]

```
int sortiran(const int a[], size_t n)
{
  int i;
  for (i = 0; i < n - 1; i++)
    if (!(a[i] <= a[i + 1]))
      return 0;
  return 1;
}
```

Navedena funkcija vraća 1 ako i samo ako je niz `a` uređen u skladu sa relacijom `<=` i tada kažemo da je on uređen ili sortiran *neopadajuće*. Slično, ako je niz uređen u skladu sa relacijom `<` kažemo da je uređen *rastuće*, ako je niz

[5] Naravno, uslov `!(a[i] <= a[i + 1])` može da se zameni jednostavnijim `a[i] > a[i + 1]`, ali je ovde naveden zbog jasnijeg izlaganja i uopštenja opisanih u nastavku.

uređen u skladu sa relacijom >= kažemo da je uređen *nerastuće*, i ako je niz uređen u skladu sa relacijom > kažemo da je uređen *opadajuće*.

Niz nekog brojevnog tipa, dakle, može biti uređen na različite načine. Niz tipa `int` može, na primer, biti uređen i neopadajuće po zbiru svojih cifara. Sledeća funkcija proverava da li je niz uređen na takav način (podrazumeva se da funkcija `int zbir_cifara(int)` vraća zbir cifara svog parametra):

```
int sortiran(const int a[], size_t n)
{
  int i;
  for (i = 0; i < n-1; i++)
    if (!(zbir_cifara(a[i]) <= zbir_cifara(a[i + 1])))
      return 0;
  return 1;
}
```

Primetimo da su prethodne dve funkcije veoma slične i razlikuju se samo u liniji u kojoj se proverava da li su dva susedna elementa u ispravnom poretku. Te provere mogu se izdvojiti u funkcije, te se pokazivač na njih može koristiti kao dodatni argument funkcije `sortiran`.

```
int u_poretku1(int x, int y)
{
  return (x <= y);
}

int u_poretku2(int x, int y)
{
  return (x >= y);
}

int u_poretku3(int x, int y)
{
  return (zbir_cifara(x) <= zbir_cifara(y));
}

int sortiran(const int a[], size_t n, int (*p)(int, int))
{
  int i;
  for (i = 0; i < n-1; i++)
    if (!p(a[i], a[i+1]))
      return 0;
  return 1;
}
```

Koristeći navedeni kôd, pozivima poput `sortiran(a, 10, u_poretku1)` i `sortiran(a, 10, u_poretku2)` može se utvrditi da li je niz `a` od 10 elemenata sortiran na jedan ili na drugi način. Štaviše, forma funkcije `sortiran` može da se uopšti ne samo za bilo koju vrstu poređenja, nego i na bilo koji tip niza. Mogla bi, dakle, da postoji jedinstvena, *generička* funkcija koja proverava da li je zadati niz proizvoljnog tipa uređen na neki zadati način. Takva funkcija, međutim, ne bi imala parametar `int a[]` (tj. `int *a`) jer treba da radi ne samo za tip `int`, već i za bilo koji drugi. Zato će parametar koji prima niz da bude tipa `void *`. Pošto takav parametar nosi samo informaciju o adresi početka niza, a ne i o lokacijama pojedinačnih elemenata, biće potreban i parametar koji predstavlja veličinu jednog elementa niza (u bajtovima). Kao i u osnovnom slučaju, biće potreban i parametar koji daje broj elemenata niza, kao i pokazivač na funkciju koja definiše poređenje između dva elementa. Međutim, u opštem slučaju, ni funkcija poređenja neće imati prototip poput `int u_poretku1(int x, int y)`, jer neće biti proveravani samo nizovi tipa `int`. Umesto da su parametri funkcije za poređenje tipa `int`, oni mogu da budu pokazivači na `int`, a u opštem slučaju pokazivači na `void`. Dodatno, kako elementi niza ne smeju biti promenjeni tokom ovih funkcija, parametri će biti tipa `const void *`, pa funkcije poređenja mogu imati opšti prototip `int u_poretku(const void *x, const void *y)`. Opšta, generička funkcija za proveravanje da li je niz sortiran može, onda, biti definisana na sledeći način (i ima linearnu vremensku složenost po dužini zadatog niza):

```
int sortiran(const void *a, size_t n, size_t velicina,
             int (*p)(const void *x, const void *y))
{
  int i;
  for (i = 0; i < n-1; i++)
    if (!p(a+i*velicina,a+(i+1)*velicina))
      return 0;
  return 1;
}
```

U pozivu `p(a+i*velicina,a+(i+1)*velicina)`, funkciji `p` šalju se adrese `i`-tog i `i+1`-tog elementa niza i ona vrši njihovo poređenje. Ova funkcija `p` ima generički prototip (da bi se uklopila u generičku formu funkcije `sortiran`), ali je njena definicija nužno prilagođena jednom konkretnom tipu. Na primer, za tip `int`, ona može biti definisana na sledeći način.

```
int u_poretku_int(const void *x, const void *y)
{
  return (*(int *)x <= *(int *)y);
}
```

U navedenom kodu, podrazumeva se da pokazivači `x` i `y` pokazuju na neke

elemente niza koji je tipa `int`. Ovi pokazivači se, međutim, ne mogu dereferencirati jer je njihov tip `const void *`. Potrebno ih je najpre kastovati u tip `int *`, a tek onda dereferencirati (i obraditi vrednosti tipa `int` na koje ukazuju).

Analogno se može definisati funkcija za poređenje niski koja vraća `1` ako je prva niska manja od ili jednaka drugoj:

```
int u_poretku_niske(const void *pa, const void *pb)
{
  return (strcmp(*(char **)pa, *(char **)pb) <= 0);
}
```

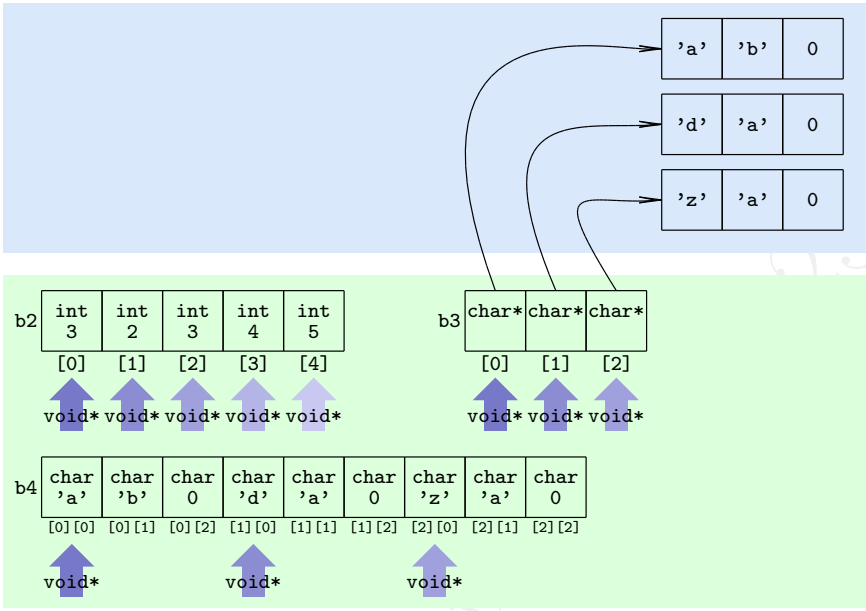
Navedene funkcije mogu se koristiti kao u sledećem primeru:

```
int main()
{
  int b1[] = { 1, 2, 3, 4, 5 };
  int b2[] = { 3, 2, 3, 4, 5 };
  char *b3[] = { "ab", "da", "za" };

  if (sortiran(b1, sizeof(b1)/sizeof(b1[0]), sizeof(b1[0]),
               u_poretku_int))
    printf("Niz b1 je sortiran.\n");
  else
    printf("Niz b1 nije sortiran.\n");
  if (sortiran(b2, sizeof(b2)/sizeof(b2[0]), sizeof(b2[0]),
               u_poretku_int))
    printf("Niz b2 je sortiran.\n");
  else
    printf("Niz b2 nije sortiran.\n");
  if (sortiran(b3, sizeof(b3)/sizeof(b3[0]), sizeof(b3[0]),
               u_poretku_niske))
    printf("Niz b3 je sortiran.\n");
  else
    printf("Niz b3 nije sortiran.\n");
  return 0;
}
```

Navedena funkcija `main` ispisuje sledeći tekst:

```
Niz b1 je sortiran.
Niz b2 nije sortiran.
Niz b3 je sortiran.
```

Razmotrimo dodatno navedeni primer. Kada se proverava da li je neki niz vrednosti tipa `int` sortiran, u funkciji `sortiran` dva pokazivača tipa `void *`



Slika 5.1: Ilustracija provere da li je niz sortiran

ukazuju na dve vrednosti tipa `int` koje treba uporediti. U okviru funkcije koja vrši poređenje, ti pokazivači se kastuju u pokazivače tipa `int *`, time je omogućeno njihovo dereferenciranje, i onda se vrši poređenje vrednosti tipa `int` (koje su dobijene dereferenciranjem). U okviru funkcije `sortiran`, pozicije pokazivača pomeraju se za širinu vrednosti `int` (videti sliku 5.1). Kada se proverava da li je neki niz vrednosti tipa `char *` sortiran, situacija je analogna, s tim što se ne porede elementi niza, već niske na koju ti elementi ukazuju: u funkciji `sortiran` dva pokazivača tipa `void *` ukazuju na dve vrednosti tipa `char *` koji ukazuju na niske koje treba uporediti. U okviru funkcije koja vrši poređenje, ti pokazivači se kastuju u pokazivače tipa `char **`, time je omogućeno njihovo dereferenciranje, i onda se vrši poređenje niski koje su identifikovane pokazivačima tipa `char *` (koji su dobijene dereferenciranjem). U okviru funkcije `sortiran`, pozicije pokazivača pomeraju se za širinu vrednosti `char *` (videti sliku 5.1). Generalno, ako se proverava da li niz tipa `T` jeste sortiran, pokazivači tipa `void *` se kastuju u pokazivače tipa `T *`, time je omogućeno njihovo dereferenciranje, i onda se vrši poređenje vrednosti tipa `T` (koje su dobijene dereferenciranjem) ili, eventualno, vrednosti na koje one ukazuju. U ovom, opštem slučaju, u okviru funkcije `sortiran`, pozicije pokazivača pomeraju se za širinu vrednosti `T`.

Iako je ovaj mehanizam u osnovi isti, razlikuje se slučaj kada je potrebno proveriti da li je sortiran skup niski datih u vidu dvodimenzionalnog niza karaktera. U tom slučaju, naime, ne vrši se poređenje pojedinačnih elemenata niza (niti vrednosti na koje oni ukazuju), već poređenje niski koje ti elementi formiraju. Pretpostavimo da je dat i niz `b4` deklarisan i inicijalizovan na sledeći način:

```
char b4[][3] = {"ab", "da", "za" };
```

Funkcija koja poredi ovakve niske može da bude definisana na sledeći način:

```
int u_poretku_niske2(const void *pa, const void *pb)
{
  return (strcmp((char *)pa, (char *)pb) <= 0);
}
```

a funkcija `sortiran` mogla bi biti pozvana na sledeći način:

```
sortiran(b4, sizeof(b4)/sizeof(b4[0]), sizeof(b4[0]),
         u_poretku_niske2);
```

U ovom scenariju, u funkciji `sortiran` dva pokazivača tipa `void *` ukazuju na dve vrednosti tipa `char`. U okviru funkcije koja vrši poređenje, ti pokazivači se kastuju u pokazivače tipa `char *` i oni su, kao takvi, već spremni da budu argumenti funkcije `strcmp`. U okviru funkcije `sortiran`, pozicije pokazivača pomeraju se za po `sizeof(b4[0])` bajtova, koliko je veličina jedne niske (to je u ovom konkretnom slučaju tri bajta, videti sliku 5.1).

Navedena generička funkcija `sortiran` veoma je jednostavna i stoga nema veliku upotrebnu vrednost, ali ilustruje mehanizam koji se koristi i u mnogo važnijim funkcijama opisanim u poglavljima 5.2.2 i 5.3.7.

Pitanja i zadaci za vežbu

Pitanje 5.1. *Koja svojstva zadovoljavaju relacije poretka? Da li relacioni operatori* `==` *i* `!=` *određuju relacije poretka?*

Pitanje 5.2. *Da li se poredak prirodnih brojava po njihovoj vrednosti razlikuje od leksikografskog poretka nad ciframa u dekadnom zapisu prirodnih brojeva? Obrazložiti.*

Pitanje 5.3. *U kojem slučaju važi da je niz sortiran u skladu sa relacijom* `<=` *i istovremeno u skladu sa relacijom* `>=`*?*

Pitanje 5.4. *U kojem vremenu se može proveriti da li su svi elementi nekog niza brojeva međusobno jednaki (podrazumevati da se poređenje jednakosti dva elementa niza izvršava u konstantnom vremenu)?*

U kojem vremenu se može proveriti da li je niz brojeva sortiran u skladu sa zadatom relacijom (podrazumevati da se poređenje dva elementa niza izvršava u konstantnom vremenu)?

Zadatak 5.1. *Struktura* `student` *definisana je na sledeći način:*

```
typedef struct _student {
  char ime[20];
  char prezime[20];
  int broj_indeksa;
} student;
```

Definisati funkciju koja za dva objekta ovog tipa proverava da li su jednaka. Definisati funkciju koja proverava da li je jedan objekat manji od drugog objekta ovog tipa, pri čemu se poređenje vrši najpre po prezimenu, pa po imenu, pa po broju indeksa. Napisati funkcija koja koristi prethodnu funkciju i proverava da li je niz tipa `student` *sortiran.*

5.2 Pretraživanje

Pod *pretraživanjem* (*pretragom*) za dati niz elemenata podrazumevamo određivanje indeksa elementa niza koji je jednak datoj vrednosti ili ispunjava neko drugo zadato svojstvo (na primer, najveći je element niza).

5.2.1 Linearno pretraživanje

Linearno (ili *sekvencijalno*) pretraživanje serije elemenata (najčešće, ali ne nužno niza) je pretraživanje zasnovano na ispitivanju redom svih elemenata ili ispitivanju redom elemenata sve dok se ne naiđe na traženi element (zadatu vrednost ili element koji ima neko specifično svojstvo, na primer — dan kada je temperatura bila iznad 40 stepeni).

U osnovnoj varijanti linearnog pretraživanja, u kojoj tražimo element u datoj seriji elemenata, tip elementa mora biti takav da je moguće ispitati jednakost dva elementa (na primer, ako ispitujemo da li se među učitanim brojevima nalazi neki traženi ili ako tražimo poziciju prvog pojavljivanja niske u datom nizu niski). Nekada su podaci koji se pretražuju takvi da su jednoznačno određeni vrednošću takozvanog *ključa* i tada je dovoljno moći ispitati jednakost ključeva (na primer, ako imamo niz struktura koje čuvaju podatke o studentima, pretragu studenta obično vršimo na osnovu broja indeksa i dovoljno je uporediti da li je broj indeksa tekućeg studenta jednak traženom broju indeksa. U naprednijim varijantama linearnog pretraživanja traži se element ili pozicija elementa koji ima neko zadato svojstvo i tada je potrebno da za svaki element možemo eksplicitno proveriti da li zadovoljava to svojstvo.

Pod pretpostavkom da se provera svakog elementa vrši u konstantnoj složenosti, linearno pretraživanje je linearne vremenske složenosti po dužini serije

elemenata koja se pretražuje. Ukoliko se u seriji od n elemenata traži element koji je jednak zadatoj vrednosti, u prosečnom slučaju (ako su elementi serije nasumično raspoređeni), ispituje se $n/2$, u najboljem 1, a u najgorem slučaju n elemenata serije.

Primer 5.1. Naredna funkcija vraća indeks prvog pojavljivanja zadatog celog broja `x` u zadatom nizu `a` dužine `n` ili vrednost -1, ako se taj broj ne pojavljuje u `a`:

```
int linearna_pretraga(int a[], int n, int x)
{
  int i;
  for (i = 0; i < n; i++)
    if (a[i] == x)
      return i;
  return -1;
}
```

Složenost ove funkcije je $O(\mathtt{n})$ a njena ispravnost se dokazuje jednostavno (pod pretpostavkom da nenegativan broj `n` predstavlja broj elemenata niza `a`).

Česta greška prilikom implementacije linearne pretrage je prerano vraćanje negativne vrednosti, kao što je ilustrovano narednim kodom. Ukoliko početni element niza nije jednak `x`, druga naredba `return` prouzrokuje prekid rada funkcije, pa se uvek izvršava tačno jedna iteracija petlje.

```
int linearna_pretraga(int a[], int n, int x)
{
  int i;
  for (i = 0; i < n; i++)
    if (a[i] == x)
      return i;
    else
      return -1;
}
```

Primer 5.2. Linearna pretraga može biti realizovana korišćenjem rekurzije (mada se to retko radi, zbog loše prostorne složenosti). Naredna implementacija poziva pomoćnu funkciju `linearna_pretraga_` koja ima dodatni parametar `i` i koja rešava opštiji zadatak – pronalazi u nizu element `x` počev od indeksa `i`. Primetimo da je funkcija `linearna_pretraga_` repno rekurzivna.

```
int linearna_pretraga_(int a[], int i, int n, int x)
{
  if (i == n)
    return -1;
```

```
  if (a[i] == x)
    return i;
  return linearna_pretraga_(a, i+1, n, x);
}

int linearna_pretraga(int a[], int n, int x)
{
  return linearna_pretraga_(a, 0, n, x);
}
```

Ukoliko se traži poslednje pojavljivanje elementa `x` u nizu, kôd je jednostavniji.

```
int linearna_pretraga(int a[], int n, int x)
{
  if (n == 0)
    return -1;
  else if (a[n - 1] == x)
    return n-1;
  else return linearna_pretraga(a, n-1, x);
}
```

I u ovom drugom slučaju, rekurzivni poziv je repno rekurzivni. Eliminacijom repne rekurzije dobija se naredni kôd:

```
int linearna_pretraga(int a[], int n, int x)
{
  while (n > 0) {
    if (a[n - 1] == x)
      return n - 1;
    n--;
  }
  return -1;
}
```

Primer 5.3. Naredna funkcija vraća indeks poslednjeg pojavljivanja zadatog karaktera `c` u zadatoj niski `s` ili vrednost -1, ako se `c` ne pojavljuje u `s`:

```
int string_last_char(char s[], char c)
{
  int i;
  for (i = strlen(s)-1; i >= 0; i--)
    if (s[i] == c)
      return i;
  return -1;
```

```
}
```

Složenost ove funkcije linearna je po dužini niske `s`, a njena korektnost dokazuje se jednostavno.

Primer 5.4. Naredna funkcija vraća indeks najvećeg elementa među prvih `n` elemenata niza `a` (pri čemu je `n` veće od ili jednako 1):

```
int max(int a[], int n)
{
  int i, index_max;
  index_max = 0;
  for(i = 1; i < n; i++)
    if (a[i] > a[index_max])
      index_max = i;
  return index_max;
}
```

Složenost ove funkcije je $O(n)$ a njena korektnost dokazuje se jednostavno.

Primer 5.5. Linearno pretraživanje može se koristiti i u situacijama kada se ne traži samo jedan element niza sa nekim svojstvom, nego više njih. Sledeći program sadrži funkciju koja vraća indekse *dva najmanja* (ne nužno različita) elementa niza. Ona samo jednom prolazi kroz zadati niz i njena složenost je $O(n)$.

```
#include <stdio.h>

int min2(int a[], int n, int *index_min1, int *index_min2)
{
  int i;
  if (n < 2)
    return -1;
  if (a[0] <= a[1]) {
    *index_min1 = 0; *index_min2 = 1;
  }
  else {
    *index_min1 = 1; *index_min2 = 0;
  }
  for (i = 2; i < n; i++) {
    if (a[i] < a[*index_min1]) {
      *index_min2 = *index_min1;
      *index_min1 = i;
    } else if (a[i] < a[*index_min2])
      *index_min2 = i;
  }
```

```
  return 0;
}

int main()
{
  int a[] = {12, 13, 2, 10, 34, 1};
  int n = sizeof(a)/sizeof(a[0]);
  int i, j;
  if (min2(a, n, &i, &j) == 0)
    printf("Najmanja dva elementa niza su %d i %d.\n",
                                          a[i], a[j]);
  else
    printf("Neispravan ulaz.\n");
  return 0;
}
```

5.2.2 Binarno pretraživanje

Binarno pretraživanje je algoritam pretrage uređene (sortirane) serije elemenata (najčešće, ali ne nužno niza). U osnovnoj varijanti proverava se da li uređeni niz sadrži neku konkretnu vrednost, ali se algoritam može primeniti i na širu klasu problema. Na primer, ako se zna da su elementi uređeni tako da prvo idu oni koji ne zadovoljavaju neko svojstvo P, a zatim oni koji zadovoljavaju svojstvo P, moguće je efikasno pronaći prvi element koji to svojstvo zadovoljava (ili poslednji element koji ne zadovoljava to svojstvo). Tako, na primer, u sortiranom nizu možemo pronaći prvi element koji je veći ili jednak od neke date vrednosti.

U svakom koraku, sve dok se ne pronađe tražena vrednost, serija elemenata deli se na dva dela i pretraga se nastavlja samo u jednom njenom delu — odbacuje se deo koji sigurno ne sadrži traženu vrednost. Ako sortirani niz u kojem tražimo zadati element ima n (konačno mnogo) elemenata, pretraživanje se vrši na sledeći način: pronalazi se srednji element (element na središnjoj poziciji) dela niza koji se pretražuje, proverava se da li je on jednak zadatoj vrednosti i ako jeste – vraća se njegov indeks, a ako nije – pretraživanje se nastavlja nad delom niza u kojem su svi manji elementi (ako je srednji element veći od zadate vrednosti) ili u delu niza u kojem su svi veći elemenati (ako je srednji element manji od zadate vrednosti). Binarno pretraživanje je u tom, diskretnom slučaju, logaritamske vremenske složenosti — $O(\log n)$, gde je n dužina niza koji se pretražuje. Binarno pretraživanje je primer pristupa *podeli i vladaj* (engl. *divide and conquer*). Pošto se jedna polovina elemenata eliminiše, ponekad se ovaj pristup naziva i *smanji i vladaj* (engl. *decrease and conquer*).

Primer 5.6. Binarno pretraživanje može se koristiti u igri pogađanja zamišljenog prirodnog broja iz zadatog intervala. Jedan igrač treba da zamisli jedan broj iz tog intervala, a drugi igrač treba da pogodi taj broj, na osnovu

što manjeg broja pitanja na koje prvi igrač odgovara samo sa *da* ili *ne*. Ako pretpostavimo da interval čine brojevi od 1 do 16 i ako je prvi igrač zamislio broj 11, onda igra može da se odvija na sledeći način:

Da li je zamišljeni broj veći od 8*? da*
Da li je zamišljeni broj veći od 12*? ne*
Da li je zamišljeni broj veći od 10*? da*
Da li je zamišljeni broj veći od 11*? ne*

Na osnovu dobijenih odgovora, drugi igrač može da zaključi da je zamišljeni broj 11. Generalno, broj potrebnih pitanja je reda $O(\log k)$, gde je k širina polaznog intervala. Naime, posle prvog pitanja širina sa k opada na $\frac{k}{2}$, posle sledećeg na $\frac{k}{4}$ itd. Posle m pitanja širina intervala opada na $\frac{k}{2^m}$. Pošto se pretraga vrši sve dok interval ne postane jednočlan ili prazan, važi da je $\frac{k}{2^m} \leq 1$, tj. da se odgovor dobija kada m dostigne vrednost približno jednaku $\log_2 k$.

Primer 5.7. Ukoliko u prethodnoj igri nije zadata gornja granica intervala, najpre treba odrediti jedan broj koji je veći od zamišljenog broja i onda primeniti binarno pretraživanje. Ako pretpostavimo da je prvi igrač zamislio broj 11, onda igra može da se odvija na sledeći način:

Da li je zamišljeni broj veći od 1*? da*
Da li je zamišljeni broj veći od 2*? da*
Da li je zamišljeni broj veći od 4*? da*
Da li je zamišljeni broj veći od 8*? da*
Da li je zamišljeni broj veći od 16*? ne*

Na osnovu dobijenih odgovora, drugi igrač može da zaključi da je zamišljeni broj u intervalu od 9 do 16 i da primeni binarno pretraživanje na taj interval. Broj pitanja potrebnih za određivanje intervala pretrage je $O(\log k)$, gde je k zamišljeni broj a ukupna složenost pogađanja ponovo je $O(\log k)$.

Binarno pretraživanje daleko je efikasnije nego linearno, ali zahteva da su podaci koji se pretražuju uređeni. To je i jedan od glavnih razloga da se u rečnicima, enciklopedijama, telefonskim imenicima i slično odrednice sortiraju. Ovakve knjige obično se pretražuju postupkom koji odgovara varijantama binarne pretrage[6]. Odnos složenosti postaje još očigledniji ukoliko se zamisli koliko bi komplikovano bilo sekvencijalno pretraživanje reči u nesortiranom rečniku.

U osnovnoj varijanti u kojoj se traži dati element u datom nizu brojeva potrebno je da niz bude uređen (sortiran) u odnosu na neki poredak i da svaki element niza može da se uporedi sa traženim u odnosu na taj poredak. U varijanti niza u kojoj se traži prvi element koji zadovoljava neko svojstvo, dovoljno je da za svaki element niza umemo da ispitamo da li zadovoljava to svojstvo (pri čemu je potrebno osigurati da su elementi serije takvi da prvo idu svi elementi koji nemaju to svojstvo, pa onda svi oni koji imaju to svojstvo).

[6] Postupak se naziva *interpolaciona pretraga* i podrazumeva da se knjiga ne otvara uvek na sredini, već se tačka otvaranja određuje otprilike na osnovu položaja slova u abecedi (na primer, ako se traži reč na slovo B, knjiga se otvara mnogo bliže početku, a ako se traži reč na slovo U, knjiga se otvara mnogo bliže kraju).

Binarno pretraživanje može se implementirati iterativno ili rekurzivno.

Naredna implementacija binarnog pretraživanja poziva pomoćnu, rekurzivnu funkciju `binarna_pretraga_` koja rešava nešto opštiji zadatak – vraća indeks elementa niza `a` između indeksa `l` i indeksa `d` (uključujući i njih) koji je jednak zadatoj vrednosti `x` ako takav postoji, a vraća `-1` inače.

```
int binarna_pretraga_(int a[], int l, int d, int x)
{
  int s;
  if (l > d)
    return -1;
  s = l + (d - l)/2;
  if (x == a[s])
    return s;
  if (x < a[s])
    return binarna_pretraga_(a, l, s-1, x);
  else /* if (x > a[s]) */
    return binarna_pretraga_(a, s+1, d, x);
}

int binarna_pretraga(int a[], int n, int x)
{
  return binarna_pretraga_(a, 0, n-1, x);
}
```

Primetimo da se, umesto izraza `l + (d - l)/2`, za određivanje središnjeg indeksa može koristiti i kraći izraz `(l + d)/2`. Ipak, upotreba prvog izraza je preporučena kako bi se smanjila mogućnost nastanka prekoračenja. Ovo je jedan od mnogih primera gde izrazi koji su matematički jednaki, imaju različita svojstva u aritmetici fiksne širine.

Složenost navedene funkcije je $O(\log \mathtt{n})$, a njena korektnost dokazuje se jednostavno (indukcijom), pri čemu se pretpostavlja da je niz `a` sortiran.

Oba rekurzivna poziva u navedenoj implementaciji su repno-rekurzivna, tako da se mogu jednostavno eliminisati. Time se dobija iterativna funkcija koja vraća indeks elementa niza `a` koji je jednak zadatoj vrednosti `x` ako takva postoji i `-1` inače.

```
int binarna_pretraga(int a[], int n, int x)
{
  int l, d, s;
  l = 0; d = n-1;
  while (l <= d) {
    s = l + (d - l)/2;
    if (x == a[s])
```

```
      return s;
    if (x < a[s])
      d = s - 1;
    else /* if (x > a[s]) */
      l = s + 1;
  }
  return -1;
}
```

Sistemska implementacija binarnog pretraživanja

U standardnoj biblioteci jezika C postoji podrška za binarno pretraživanje u vidu funkcije `bsearch`, deklarisane u okviru zaglavlja `stdlib.h`. Implementacija ove funkcije je *generička*, jer se može koristiti za pretraživanje niza bilo kog tipa i bilo koje dužine i koristeći bilo koji uslov poređenja elemenata (u istom duhu kao generička funkcija za proveravanje da li je niz sortiran u skladu sa zadatim poretkom, opisana u poglavlju 5.1.3). Prototip funkcije `bsearch` je:

```
void *bsearch(const void *key, const void *base,
              size_t number, size_t size,
              int (*compare)(const void *e1, const void *e2));
```

Argumenti funkcije imaju sledeću ulogu:

`key` je pokazivač na vrednost koja se traži; on je tipa `void *` kako bi mogao da prihvati pokazivač na bilo koji konkretan tip;

`base` je pokazivač na početak niza koji se pretražuje; on je tipa `void *` kako bi mogao da prihvati pokazivač na bilo koji konkretan tip;

`number` je broj elemenata niza koji će biti ispitani;

`size` je veličina jednog elementa u bajtovima; ona je potrebna kako bi funkcija mogla da izračuna adresu svakog pojedinačnog elementa niza;

`compare` je pokazivač na funkciju (često definisanu od strane korisnika) koja vrši poređenje dve vrednosti zadatog tipa analogno funkcijama za poređenje iz poglavlja 5.1; da bi funkcija imala isti prototip za sve tipove, njeni argumenti su tipa `void *` (zapravo — `const void *` jer vrednost na koju ukazuje ovaj pokazivač ne treba da bude menjana u okviru funkcije `compare`). Funkcija vraća jednu od sledećih vrednosti:

- < 0, ako je u izabranom poretku vrednost na koju ukazuje prvi argument manja od vrednosti na koju ukazuje drugi argument;
- 0, ako su u izabranom poretku vrednosti na koju ukazuju prvi i drugi argument jednake;

- > 0, ako je u izabranom poretku vrednost na koju ukazuje prvi argument veća od vrednosti na koju ukazuje drugi argument.

Pretpostavlja se da je zadati niz sortiran u skladu sa relacijom poretka kojoj odgovara funkcija `compare`. Nad podacima istog tipa mogu se definisati različita uređenja. Na primer, celi brojevi mogu se sortirati neopadajuće ili nerastuće. Shodno tome, funkcija `bsearch` može se koristiti za pretraživanje nizova celih brojeva koji su sortirani bilo neopadajuće ili nerastuće.

Funkcija `bsearch` vraća pokazivač na element koji je pronađen ili `NULL`, ako on nije pronađen. Ukoliko u nizu ima više elemenata koji su jednaki traženoj vrednosti, biće vraćen pokazivač na jednog od njih (standard ne propisuje na koji).

Zbog opštosti funkcije `bsearch` i prosleđivanja pokazivača na `void` kao argumenata, nema provere tipova čime se povećavaju šanse za pogrešnu upotrebu ove funkcije.

Primer 5.8. Naredni program, koristeći sistemsku implementaciju binarne pretrage, traži zadati element u datom nizu celih brojeva.

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

int poredi_dva_cela_broja(const void *px, const void *py)
{
  return ( *(int *)px - *(int *)py );
}

int main () {
  int a[] = { 10, 20, 25, 40, 90, 100 };
  int *p;
  int trazena_vrednost = 40;
  p = (int *)bsearch(&trazena_vrednost, a,
                     sizeof(a)/sizeof(int), sizeof(int),
                     poredi_dva_cela_broja);
  if (p != NULL)
    printf("Trazena vrednost se nalazi u nizu.\n");
  else
    printf("Trazena vrednost se ne nalazi u nizu.\n");
  return 0;
}
```

Funkcija poređenja brojeva mogla je da bude implementirana i na sledeći način:

```
int poredi_dva_cela_broja(const void *px, const void *py)
{
```

```
  int x = *(int *)px, y = *(int *)py;
  if (x < y) return -1;
  else
    if (x > y) return 1;
  else /* if (x == y) */
    return 0;
}
```

Na ovaj način uklanja se mogućnost nastanka prekoračenja izračunavanja razlike dva cela broja.

Binarno traženje prelomne tačke

Osnovna varijanta binarne pretrage, čija je implementacija prikazana, pronalazi datu vrednost u nizu. Kao što je već nagovešteno, binarna pretraga može se upotrebiti i za rešavanje nešto opštijih problema. Pretpostavimo da je niz uređen tako da svi njegovi početni elementi zadovoljavaju neki uslov P, a da posle njih idu elementi koji ne zadovoljavaju taj uslov i da je potrebno pronaći mesto u nizu gde se ta promena dešava (tj. potrebno je pronaći poslednji element koji zadovoljava uslov P ili prvi element koji ga ne zadovoljava). Na primer, može biti poznato da se u nizu nalaze prvo parni, a zatim neparni elementi i potrebno je pronaći koliko postoji svakih. Slično, može se razmatrati sortiran niz i za uslov P uzeti uslov da je element niza strogo manji od neke date vrednosti x (u sortiranom nizu, prvo su svi elementi koji su strogo manji od x, a iza njih su elementi koji su veći od ili jednaki x). Kao ilustraciju varijante binarne pretrage u kojoj se pronalazi takva prelomna tačka, u nastavku je prikazana funkcija koja pronalazi poziciju prvog elementa u sortiranom nizu koji je veći od ili jednak datom elementu x (ili vraća dužinu niza ako takav element ne postoji).

Uvedimo promenljive l i d i osigurajmo, kao invarijantu, da sve vreme tokom pretrage važi da su elementi niza a na pozicijama iz intervala $[0, l)$ manji od vrednosti x (zadovoljavaju uslov P), a da su elementi na pozicijama iz intervala $[d, n)$ veći od ili jednaki x (ne zadovoljavaju uslov P). Elementima na pozicijama iz intervala $[l, d)$ status još nije poznat. Ovaj uslov biće na početku ispunjen, ako se promenljiva l inicijalizuje na nulu, a promenljiva d na vrednost n (tada je interval $[d, n)$, tj. interval $[n, n)$ prazan).

Neka s predstavlja sredinu intervala $[l, d)$. Ako je element niza a na poziciji s manji od vrednosti x (zadovoljava uslov P) takvi su i svi elementi iz intervala $[l, s]$. Zato se vrednost l može postaviti na $s + 1$ i invarijanta će ostati da važi (zaista, ako je $l' = s + 1$, svi elementi iz intervala pozicija $[0, l')$ će biti ili iz intervala pozicija $[0, l)$ za koje se zna da su manji od x (zadovoljavaju uslov P) ili su iz intervala pozicija $[l, s]$ za koje je upravo utvrđeno da su manji od x tj. da zadovoljavaju uslov P).

Ako je element niza na poziciji s veći od ili jednak x (ne zadovoljava uslov P), tada su i svi elementi intervala $[s, d)$ veći od ili jednaki x (i ne zadovoljavaju

uslov P). Zato se vrednost d može postaviti na s i invarijanta će ostati da važi (zaista, ako je $d' = s$, tada su svi elementi iz intervala pozicija $[d', n)$ ili iz intervala $[s, d)$ za koje je upravo utvrđeno da će biti veći od ili jednaki x ili iz intervala $[d, n)$ za koje se od ranije zna da su veći od ili jednaki x).

Pretraga se vrši sve dok postoje elementi nepoznatog statusa, tj. sve dok je interval $[l, d)$ neprazan, odnosno dok je $l < d$. U trenutku kada važi $l = d$, na osnovu invarijante sledi:

- ako je $d = n$, svi elementi niza manji su od x, pa je d koje je jednako n zaista tražena vrednost;
- ako je $d < n$ (i $l = d$), prvi element koji je veći od ili jednak x (tj. prvi element koji ne zadovoljava uslov P) nalazi se na poziciji d (jer su svi elementi u intervalu $[0, l)$ tj. u intervalu $[0, d)$ strogo manji od x, dok su elementi na pozicijama $[d, n)$ veći od ili jednaki x), pa je d zaista tražena vrednost.

```
int prvi_veci_ili_jednak(int a[], int n, int x)
{
  int l = 0, d = n;
  while (l < d) {
    int s = l + (d - l) / 2;
    if (a[s] < x) {
      l = s + 1;
    } else
      d = s;
  }
  return d;
}
```

Napomenimo da smo funkciju mogli zasnovati i na invarijanti da se u intervalu $[0, l)$ nalaze elementi koji su manji od x (zadovoljavaju svojstvo P), da se u intervalu (d, n) nalaze elementi koji su veći od ili jednaki x (ne zadovoljavaju svojstvo P), a da se u intervalu $[l, d]$ nalaze elementi čiji status još nije poznat. Tada bi vrednost l bila inicijalizovana na 0, a d na $n - 1$. Pretraga se vrši dok je $l \leq d$. Nakon pronalaženja sredine s intervala $[l, d]$, ako je $a[s] < x$, tada se l može postaviti na $s + 1$ (jer su svi elementi na pozicijama levo od s zaključno sa njom manji od x), a u suprotnom se d može postaviti na $s - 1$ (jer su svi elementi od pozicije s pa naviše veći od ili jednaki x)

Na sličan način može se odrediti i pozicija prvog elementa koji je strogo veći od zadatog broja, poslednjeg elementa koji je manji od ili jednak datom broju ili poslednjeg elementa koji strogo manji od datog broja.

Problem traženja prelomne tačke opštiji je od problema traženja zadate vrednosti u sortiranom nizu, jer se u ovom drugom problemu podrazumeva postojanje relacije poretka na osnovu koje su vrednosti u nizu sortirane, dok se u

prvom problemu podrazumevamo samo da se u nizu prvo javljaju elementi koji zadovoljavaju neko svojstvo P, pa onda oni koji ga ne zadovoljavaju. Varijanta u kojoj se pronalazi prelomna tačka može se jednostavno iskoristi i za proveru da li niz sadrži dati broj. Kada se nađe pozicija prvog elementa koji je veći od ili jednak traženom, jednostavno se može proveriti da li se na toj poziciji nalazi upravo taj element (ako postoji u nizu, on mora biti na toj poziciji).

```
int binarna_pretraga(int a[], int n, int x)
{
  int p = prvi_veci_ili_jednak(a, n, x);
  if (p < n && a[p] == x)
    return p;
  return -1;
}
```

Korišćenjem funkcije `prvi_veci_ili_jednak` može se, na primer, odrediti i broj elemenata koji su veći od ili jednaki datom elementu u nekom sortiranom nizu.

```
int broj_vecih_ili_jednakih(int a[], int n, int x)
{
  int p = prvi_veci_ili_jednak(a, n, x);
  return n - p;
}
```

Određivanje nula funkcije

Jedna od numeričkih metoda za određivanje nula neprekidne realne funkcije zasniva se na metodi polovljenja intervala koji odgovara binarnoj pretrazi nad neprekidnim domenom. Pretpostavka metode je da je funkcija neprekidna i da su poznate vrednosti a i b u kojima funkcija ima različit znak. U svakom koraku tekući interval se polovi i postupak se nastavlja nad jednom ili drugom polovinom u zavisnosti od toga da li je u središtu intervala funkcija pozitivna ili negativna. Metod uvek konvergira (kada su ispunjeni uslovi primene), tj. za bilo koju zadatu tačnost pronalazi odgovarajuće rešenje. U svakom koraku postupka, dužina intervala smanjuje se dvostruko i ako je dužina početnog intervala jednaka d, onda nakon n iteracija dužina tekućeg intervala jednaka $d/2^n$. Zato, ukoliko je nula funkcije tražena sa greškom manjom od ε, potrebno je napraviti $\log_2(d/\varepsilon)$ iteracija. Postoje i brži postupci za određivanje nule funkcije (na primer, metoda sečice ili Njutnova metoda), ali je metoda polovljenja primenljivija na širu klasu funkcija i numerički može biti stabilnija.

Primer 5.9. Naredna funkcija pronalazi, za zadatu tačnost, nulu zadate funkcije `f`, na intervalu [`l`, `d`], pretpostavljajući da su vrednosti funkcije `f` u `l` i `d` različitog znaka.

```
float polovljenje(float (*f)(float), float l, float d,
                  float epsilon)
{
  float fl = (*f)(l), fd = (*f)(d);
  for (;;) {
    float s = (l+d)/2;
    float fs = (*f)(s);
    if (fs == 0.0 || d-l < epsilon)
      return s;
    if (fl*fs <= 0.0) {
      d = s; fd = fs;
    }
    else {
      l = s; fl = fs;
    }
  }
}
```

U navedenoj funkciji uslov da su vrednosti `fl` i `fs` različitog znaka proverava se proverom tačnosti uslova `fl*fs <= 0.0`. Ovako formulisan uslov jeste koncizan, ali može da dovede do prekoračenja, pa treba biti obazriv prilikom njegovog korišćenja.

Kriterijum zaustavljanja je takav da se postupak prekida kada se interval na x-osi dovoljno suzi. Česta greška je da se postupak zaustavi kada je vrednost `fs` mala, međutim, kod funkcija koje sporo rastu, ta vrednost može biti mala iako je argument funkcije veoma udaljen od njene nule.

5.2.3 Tehnika dva pokazivača

Sortiranost niza omogućava (pa i sugeriše) primenu binarnog pretraživanja. Međutim, postoje još neke tehnike pretrage koje koriste činjenicu da je niz sortiran ili se koriste prilikom sortiranja nizova. Jedna grupa takvih tehnika, u kojima dva pokazivača prolaze kroz niz (obično sa dva njegova kraja) ili prolaze kroz dva niza naziva se *tehnika dva pokazivača (engl. two pointer)*.

Primer 5.10. Razmotrimo problem određivanja broja parova različitih elemenata strogo rastuće sortiranog niza čiji je zbir jednak datom broju s. Naivna varijanta je da proverimo sve parove elemenata, što dovodi do algoritma složenosti $O(n^2)$. Bolja varijanta je da se za svaki element niza a_i binarnom pretragom (poglavlje 5.2.2) proveri da li se na pozicijama od $i+1$ do kraja niza nalazi element $s - a_i$ i ako postoji, da se uveća brojač parova. Time se dobija algoritam složenosti $O(n \log n)$. Međutim, postoji bolji algoritam i od toga.

Definišimo rekurzivnu funkciju koja izračunava koliko parova čiji je zbir s ima u delu niza između pozicija l i d, tj. u delu niza čije su pozicije u intervalu

$[l, d]$. Traženi broj parova jednak je vrednosti te funkcije za interval $[0, n-1]$, tj. za poziv $l = 0$, $d = n - 1$. Postoje sledeći slučajevi:

- Ako je $l \geq d$ tada je interval jednočlan ili prazan i u njemu ne može da postoji nijedan par (a_i, a_j) takav da je $i < j$, pa funkcija treba da vrati nulu.
- Ako je $a_l + a_d > s$, tada je broj parova jednak broju parova u intervalu $[l, d-1]$. Zaista, za $l \leq l' < d$, nijedan od parova $(a_{l'}, a_d)$ ne može imati zbir s, jer je niz sortiran i važi da je $a_{l'} + a_d \geq a_l + a_d > s$. Zato se iz razmatranja mogu eliminisati svi parovi tog oblika.
- Ako je $a_l + a_d < s$, tada je broj parova jednak broju parova u intervalu $[l+1, d]$. Zaista, za $l < d' \leq d$, nijedan od parova $(a_l, a_{d'})$ ne može imati zbir s, jer je niz sortiran i važi da je $a_l + a_{d'} \leq a_l + a_d < s$. Zato se iz razmatranja mogu eliminisati svi parovi tog oblika.
- Na kraju, ako je $a_l + a_d = s$, tada je broj parova za jedan veći od broja parova u intervalu $[l+1, d-1]$. U tom slučaju iz razmatranja se mogu eliminisati sve parovi oblika $(a_{l'}, a_d)$ za $l < l' < d$, i svi parovi oblika $(a_l, a_{d'})$ za $l < d' < d$ i par (a_l, a_d). Zbir ovog poslednjeg jeste s, dok nijedan od ostalih eliminisanih parova ne može da ima zbir s. Naime, pošto je niz sortiran strogo rastuće, važi $a_{l'} > a_l$ i važi da je $a_{l'} + a_d > a_l + a_d = s$. Slično je i $a_{d'} < a_d$ i važi da je $a_l + a_{d'} < a_l + a_d = s$.

Prethodno razmatranje može biti implementirano i bez rekurzije (eliminacijom repne rekurzije).

```
int broj_parova = 0;
int l = 0, d = n - 1;
while (l < d) {
  if (a[l] + a[d] > s)
    d--;
  else if (a[l] + a[d] < s)
    l++;
  else {
    broj_parova++;
    l++; d--;
  }
}
```

Broj elemenata intervala $[l, d]$ tj. $d - l + 1$ kreće od vrednosti n, celobrojan je i pozitivan, i u svakom koraku se smanjuje, pa je složenost ovog algoritma linearna tj. $O(n)$.

Na sličan način mogu se odrediti i parovi čija je razlika jednaka datom broju.

Primer 5.11. Ako je dat niz pozitivnih celih brojeva, odrediti koliko postoji nepraznih segmenata tog niza (podnizova uzastopnih elemenata) čiji elementi imaju zbir jednak datom broju s.

Naivno rešenje je ponovo da se ispitaju svi segmenti. Ako se za svaki segment iznova izračunava zbir, dobija se algoritam složenosti čak $O(n^3)$, koji je praktično neupotrebljiv. Ako se segmenti obilaze tako da se za fiksirani levi kraj l, desni kraj uvećava od l do $n-1$, onda se vrednost sume segmenata može izračunavati inkrementalno, dodajući a_d na vrednost zbira prethodnog segmenta, što daje algoritam složenosti $O(n^2)$, koji je i dalje veoma neefikasan.

Važna tehnika primenljiva u mnogim zadacima je da se suma segmenta $\sum_{i=l}^{d} a_i$ izrazi kao $\sum_{i=0}^{d} a_i - \sum_{i=0}^{l-1} a_i$[7]. Pretpostavimo da umesto niza a znamo njegove parcijalne sume tj. da znamo niz p_k definisan sa $p_0 = 0$ i $p_k = \sum_{i=0}^{k-1} a_i$. Ako elemente niza p računamo inkrementalno, tokom učitavanja elemenata niza a, složenost tog izračunavanja je samo $O(n)$. Tada je suma segmenta na pozicijama $[l, d]$ jednaka $p_{d+1} - p_l$. Pošto su po uslovu zadatka brojevi u nizu a pozitivni, niz p je sortiran rastuće i ovim je zadatak sveden na zadatak da se u sortiranom nizu p pronađe broj elemenata čija je razlika jednaka zadatom broju, što je problem pomenut u prethodnom primeru (može se rešiti binarnom pretragom u ukupnoj složenosti $O(n \log n)$ ili tehnikom dva pokazivača u ukupnoj složenosti $O(n)$).

Niz p nije neophodno čuvati u memoriji i tehnika dva pokazivača može da se primeni i na elemente originalnog niza.

```
/* broj segmenata trazenog zbira */
int broj = 0;
/* granice segmenta */
int l = 0, d = 0;
/* zbir segmenta */
int zbir = a[0];
while (1) {
  /* na ovom mestu vazi da je zbir = sum(a[l], ..., a[d])
     i da za svako l <= d' < d vazi da je
     sum(a[l], ..., a[d']) < s */

  if (zbir < s) {
    /* prelazimo na interval [l, d+1] */
    d++;
    /* ako takav interval ne postoji zavrsili smo pretragu */
    if (d >= n)
       break;
    /* na osnovu zbira intervala [l, d]
       izracunavamo zbir intervala [l, d+1] */
```

[7] Primetimo da ovo u osnovnoj ideji odgovara Njutn-Lajbnicovoj formuli iz matematičke analize.

```
      zbir += a[d];
    } else {
      /* ako je zbir jednak trazenom,
         vazi da je sum(a[l], ..., a[d]) = s
         pa prijavljujemo interval */
      if (zbir == s)
         broj++;
      /* na osnovu zbira intervala [l, d]
         izracunavamo zbir intervala [l+1, d] */
      zbir -= a[l];
      l++;
    }
  }
```

Primer 5.12. Dva već sortirana niza mogu se objediniti u treći sortirani niz samo jednim prolaskom kroz nizove (tj. u linearnom vremenu $O(m+n)$ gde su m i n dimenzije polaznih nizova):

```
void merge(int a[], int m, int b[], int n, int c[])
{
  int i, j, k;
  i = 0, j = 0, k = 0;
  while (i < m && j < n)
    c[k++] = a[i] < b[j] ? a[i++] : b[j++];
  while (i < m) c[k++] = a[i++];
  while (j < n) c[k++] = b[j++];
}
```

U prikaznoj implementaciji, paralelno se prolazi kroz niz `a` dimenzije `m` i niz `b` dimenzije `n`. Promenljiva `i` čuva tekuću poziciju u nizu `a`, dok promenljiva `j` čuva tekuću poziciju u nizu `b`. Tekući elementi se porede i manji se upisuje u niz `c` (na tekuću poziciju `k`), pri čemu se napreduje samo u nizu iz kojeg je taj manji element uzet. Postupak se nastavlja dok se ne stigne do kraja jednog od nizova. Kada se jedan niz isprazni, preostali elementi iz drugog niza se nadovezuju na kraj niza `c`.

Ova funkcija koristi se u algoritmu za sortiranje *mergesort* (poglavlje 5.3.5).

Primer 5.13. Sledeći zadatak važan je u sortiranju, u okviru znamenitog algoritma *quicksort* (poglavlje 5.3.7). Zadatak je elemente niza reorganizovati u odnosu na početni element niza x tako da u nizu najpre idu elementi manji od ili jednaki x, zatim x, a onda elementi veći od ili jednaki x. Ovaj postupak zove se *particionisanje*. Jedan način implementiranja koraka particionisanja je da se početni niz obilazi sa dva kraja i da se, kada se na levom kraju naiđe na element koji je veći, a na desnoj na element koji je manji od pivota, izvrši njihova razmena.

```
int particionisanje(int a[], int l, int d)
{
  int l_start = l;

  while (l < d) {
    while (a[l] <= a[l_start] && l < d)
      l++;
    while (a[d] >= a[l_start] && l < d)
      d--;
    if (l < d)
      razmeni(a, l, d);
  }

  if (a[l] >= a[l_start])
    l--;
  razmeni(a, l_start, l);
  return l;
}
```

Invarijanta spoljašnje petlje je da je `l` manje od ili jednako `d`, da je (multi)skup elemenata niza `a` nepromenjen (što je očigledno jer se algoritam zasniva isključivo na razmenama), da se na poziciji `l_start` nalazi pivot i da je svaki element niza `a[l_start, l-1]` jednak ili manji od pivota (gde je `l_start` početna vrednost promenljive `l`), a svaki element niza `a[d+1, d_start]` (gde je `d_start` početna vrednost promenljive `d`) jednak ili veći od pivota. Po završetku petlje je `l` jednako `d`. Element na toj poziciji još nije ispitan i njegovim ispitivanjem se osigurava da `l` bude takav da je svaki element niza `a[l_start, l]` jednak ili manji od pivota, a svaki element niza `a[l+1, d_start]` jednak ili veći od pivota. Odavde, nakon zamene elemenata na pozicijama `l_start` i `l`, postiže se postuslov particionisanja.

S obzirom na to da se u svakom koraku petlji smanjuje pozitivna celobrojna vrednost `d - l`, a sve petlje se zaustavljaju u slučaju kada je `d - l` jednako nula, algoritam se zaustavlja.

Naredna implementacija optimizuje prethodnu ideju, tako što izbegava korišćenje zamena.

```
int particionisanje(int a[], int l, int d)
{
  int pivot = a[l];
  while (l < d) {
    while (a[d] >= pivot && l < d)
      d--;
    if (l != d)
      a[l++] = a[d];
```

```
    while (a[l] <= pivot && l < d)
      l++;
    if (l != d)
      a[d--] = a[l];
  }
  a[l] = pivot;
  return l;
}
```

Invarijanta spoljašnje petlje je da je `l` manje od ili jednako `d`, da je (multi)skup elemenata niza `a` van pozicije `l` jednak (multi)skupu elemenata polaznog niza bez jednog pojavljivanja pivota, da su elementi `a[l_start, l-1]` manji od pivota (gde je `l_start` početna vrednost promenljive `l`), a elementi `a[d+1, d_start]` (gde je `d_start` početna vrednost promenljive `d`) veći od ili jednaki pivotu. Na sredini spoljne petlje, invarijanta se donekle naruši — svi uslovi ostaju da važe, osim što je (multi)skup elemenata `a` van pozicije `d` (umesto van pozicije `l`) jednak (multi) skupu elemenata polaznog niza bez jednog pojavljivanja pivota. Kada se petlja završi, `l` je jednako `d` i upisivanjem (sačuvane) vrednosti pivota na ovo mesto se postiže postuslov funkcije particionisanja.

Zaustavljanje je analogno prethodnom slučaju.

Pitanja i zadaci za vežbu

Pitanje 5.5. *Da bi se primenilo linearno pretraživanje niza (*izabrati sve ispravne odgovore*):*
(a) *neophodno je da elementi niza budu sortirani.*
(b) *poželjno je da elementi niza budu sortirani.*
(c) *elementi niza ne smeju da budu sortirani.*
(d) *nije potrebno da elementi niza budu sortirani.*
(e) *niz mora da bude niz brojeva.*
(f) *nema preduslova.*

Pitanje 5.6. *Koja je, po analizi najgoreg slučaja, složenost algoritma za linearno pretraživanje niza?*

Pitanje 5.7. *Koja je složenost linearne pretrage niza od k celih brojeva čije vrednosti su između m i n?*

Pitanje 5.8. *Ako su elementi niza sortirani, onda binarno pretraživanje tražene vrednosti u nizu radi*
(a) *najbrže.*
(b) *najsporije.*
(c) *ispravno.*

Pitanje 5.9. *Da bi binarno pretraživanje tražene vrednosti u nizu bilo primenljivo,*
(a) *neophodno je da elementi niza budu sortirani.*
(b) *poželjno je da elementi niza budu sortirani.*
(c) *elementi niza ne smeju da budu sortirani.*
(d) *nije potrebno da elementi niza budu sortirani.*

Pitanje 5.10. *Da li je binarno pretraživanje vrednosti u nizu moguće primeniti ako su elementi niza sortirani opadajuće?*

Pitanje 5.11. *Pod pretpostavkom da je elemente moguće uporediti u konstantnom vremenu, koja je, po analizi najgoreg slučaja, složenost algoritma za binarno pretraživanje vrednosti u nizu?*

Pitanje 5.12. *Koja je složenost binarne pretrage vrednosti u nizu od k celih brojeva čije vrednosti su između m i n?*

Pitanje 5.13. *Ako je niz od 1024 elementa sortiran, onda binarno pretraživanje može, i u najgorem slučaju, da proveri da li se neka vrednost nalazi u nizu u:*
(a) *oko 10 koraka;* (b) *oko 128 koraka;* (c) *oko 512 koraka;* (d) *oko 1024 koraka.*

Pitanje 5.14. *Nesortirani niz ima n elemenata. Da li se više isplati primenjivati linearnu ili binarnu pretragu, ako je potrebno proveravati da li neka zadata vrednost postoji u nizu:*
(a) $O(1)$ *puta?*
(b) $O(n)$ *puta?*
(c) $O(n^2)$ *puta?*
Šta je potrebno uraditi pre nego što se primeni binarna pretraga?

Pitanje 5.15. *Nesortirani niz ima n^2 elemenata. Da li se više isplati primenjivati linearnu ili binarnu pretragu, ako je potrebno proveravati da li neka zadata vrednost postoji u nizu:*
(a) $O(1)$ *puta?*
(b) $O(n)$ *puta?*
(c) $O(n^2)$ *puta?*
Šta je potrebno uraditi pre nego što se primeni binarna pretraga?

Pitanje 5.16. *Definisati funkciju* `poredi` *tako da može da se koristi kao poslednji element bibliotečke funkcije* `bsearch` *za pretragu niza brojeva tipa* `int` *uređenih nerastuće.*

Pitanje 5.17. *Koji uslov je dovoljan da bi metodom polovljenja intervala mogla da se aproksimira nula funkcije f, ako funkcija f ima različit znak na krajevima intervala?*

Pitanje 5.18. *Metod polovljenja intervala koji traži nulu neprekidne funkcije f na zatvorenom intervalu $[a, b]$, ima smisla primenjivati ako je funkcija f neprekidna i važi*
(a) $f(a)f(b) < 0$; (b) $f(a)f(b) > 0$; (c) $f(a) + f(b) > 0$; (d) $f(a) + f(b) < 0$.

Pitanje 5.19. *Šta je mogući problem ako se provera da li funkcija f ima različit znak na krajevima intervala $[a, b]$ vrši izračunavanjem proizvoda $f(a) \cdot f(b)$?*

Pitanje 5.20. *Kada treba zaustaviti numeričko određivanje nule funkcije metodom polovljenja intervala?*

Zadatak 5.2. *Koristeći binarnu pretragu, napisati funkciju*

```
float find_zero(float a, float b, float eps)
```

koja računa nulu funkcije $f(x) = 5sin(x)ln(x)$ na intervalu `(a, b)` *sa tačnošću* `eps`*. Brojevi* `a`*,* `b` *i* `eps` *unose se sa standardnog ulaza.*

Zadatak 5.3. *Definisati funkciju sa prototipom*

```
double treci_koren(double x, double a, double b, double eps);
```

koja, metodom polovljenja intervala, računa treći koren zadatog broja x ($x \geq 1$), tj. za dato x određuje broj k za koji važi $k^3 = x$, sa tačnošću eps i sa početnom pretpostavkom da se broj k nalazi u datom intervalu $[a, b]$.

Zadatak 5.4. *Kao argumenti komandne linije dati su celi brojevi a, b, c, x, y. Binarnom pretragom naći na intervalu $[x, y]$ nulu funkcije $ax^3 + bx + c$ sa tačnošću* `0.0001`*.*

Pitanje 5.21. *Napisati prototip funkcije koja određuje nulu zadate funkcije na intervalu* `[a,b]` *(jedan od argumenata treba da bude pokazivač na funkciju).*

Zadatak 5.5. *Definisati funkciju koja binarnom pretragom u sortiranom nizu brojeva određuje poziciju poslednjeg elementa koji ima vrednost strogo manju od date vrednosti x. Ako takav element ne postoji, funkcija treba da vrati -1.*

Zadatak 5.6. *U nizu dužine n nalaze se prvo parni, a zatim neparni brojevi. Napisati funkciju koja u složenosti $O(\log n)$ određuje poziciju prvog neparnog elementa (funkcija treba da vrati n ako takav element ne postoji).*

Zadatak 5.7. *Definisati funkciju koja u uređenom nizu niski efikasno pronalazi indeks zadatke niske (a vraća -1 ako takva niska nije pronađena).*

Zadatak 5.8. *Drvoseča treba da naseče određenu količinu drveta i ima testeru koju može da podešava da seče na bilo kojoj celobrojnoj visini (u metrima). Pošto testera seče samo drvo iznad visine na koju je postavljena, što je testera viša, naseći će se manje drveta. Pošto drvoseča brine o okolini, on ne želi da naseče više drveta nego što mu je potrebno. Napiši program koji na osnovu niza visina svih stabala i količine potrebnog drveta određuje najvišu moguću celobrojnu visinu testere, tako da drvoseča dobije dovoljno drveta (pretpostaviti da uvek postoji dovoljno drveta). Efikasnosti radi koristiti binarnu pretragu.*

Zadatak 5.9. *U duhu algoritma objedinjavanja dva sortirana niza, definisati funkciju koja ispisuje zajedničke elemente dva sortirana niza (presek dva niza). Definisati i funkciju koja određuje uniju elemenata dva sortirana niza.*

Zadatak 5.10. *Definisati funkciju koja u složenosti $O(n)$ u sortiranom nizu celih brojeva određuje broj parova elemenata koji imaju datu razliku.*

Zadatak 5.11. *Postoji n lokacija na x-osi na koje je moguće postaviti predajnik. Na raspolaganju su dva predajnika sa dometom d. Potrebno je postaviti ih tako da budu što više razmaknuti kako bi pokrivenost bila što veća, ali i da budu na razdaljini najviše d kako bi mogli međusobno da komuniciraju. Napisati program koji u složenosti $O(n)$ određuje maksimalnu razdaljinu između predajnika.*

Zadatak 5.12. *Definisati funkciju koja za dva niza celih brojeva pronalazi par elemenata takav da je prvi element iz prvog niza, drugi iz drugog i da je razlika između ta dva elementa najmanja moguća. Složenost funkcije treba da bude $O(m+n)$, gde su m i n dužine nizova.*

Zadatak 5.13. *Dat je sortiran niz cena tastatura i sortiran niz cena miševa. Kupac kupuje jednu tastaturu i jedan miš. Pošto mu je budžet ograničen, a želi da ga istroši što je više moguće, napisati program koji određuje najskuplji par (tastatura, miš) koji se može kupiti u okviru datog budžeta. Zadatak rešiti prvo korišćenjem binarne pretrage (u složenosti $O(m \log n)$), a zatim i korišćenjem tehnike dva pokazivača (u složenosti $O(m+n)$), gde je m dužina kraćeg, a n dužina dužeg niza cena.*

Zadatak 5.14. *Definisati funkciju koja određuje koliko u nizu pozitivnih celih brojeva postoji segmenata uzastopnih elemenata čiji je zbir manji od datog broja z. Složenost funkcije treba da bude $O(n \log n)$.*

5.3 Sortiranje

Sortiranje je jedan od fundamentalnih zadataka u računarstvu. Sortiranje podrazumeva uređivanje niza u odnosu na neku relaciju poretka (na primer, uređivanje niza brojeva po veličini — rastuće, opadajuće ili nerastuće, uređivanje niza niski leksikografski ili po dužini, uređivanje niza struktura na

osnovu vrednosti nekog polja i slično). Mnogi problemi nad nizovima mogu se jednostavnije i efikasnije rešiti u slučaju da je niz sortiran (na primer, pretraživanje se može vršiti binarnom pretragom).

Većina programskih jezika, uključujući i programski jezik C, u svojim bibliotekama imaju funkcije za sortiranje nizova. U realnom programskom kodu uvek je preporuka vršiti sortiranje korišćenjem tih funkcija, jer su one efikasno implementirane i detaljno testirane. S druge strane, izučavanje algoritama sortiranja može pomoći u savladanju nekih važnih algoritamskih tehnika i stoga je nezaobilazno u učenju programiranja.

Postoji više različitih algoritama za sortiranje nizova. Neki algoritmi su jednostavni i intuitivni ali obično veoma spori, dok su neki malo složeniji, ali veoma efikasni. Spore algoritme sortiranja nikada ne treba koristiti u realnim programima. Oni se uglavnom izučavaju iz istorijskih razloga, ali i kao uvod u efikasnije algoritme sortranja, jer su neki efikasni algoritmi sortiranja zasnovani na istim idejama na kojima počivaju i ovi elementarni. Najznačajniji algoritmi za sortiranje (neki samo u istorijskom kontekstu) su:

- Bubble sort
- Selection sort
- Insertion sort
- Shell sort
- Merge sort
- Quick sort
- Heap sort

Neki od algoritama za sortiranje rade *u mestu* (engl. in-place), tj. sortiraju zadate elemente bez korišćenja dodatnog niza. Drugi algoritmi zahtevaju korišćenje pomoćnog niza ili nekih drugih dodatnih struktura podataka.

Prilikom sortiranja nizova najbrojnije, pa time i najznačajnije, su operacije poređenja dva elementa niza i operacije razmene dva elementa niza. Zbog toga se prilikom izračunavanja složenosti algoritama obično u obzir uzima samo broj primena ovih operacija. Naime, pošto broj poređenja i uvećavanja brojača u petljama obično odgovara broju poređenja elemenata niza i po pravilu ga ne prevazilazi asimptotski, obično se u analizi složenosti taj broj zanemaruje.

Algoritmi za sortiranje obično pripadaju jednoj od dve klase vremenske složenosti. Jednostavniji, sporiji algoritmi sortiranja niza od n elemenata imaju složenost najgoreg slučaja $O(n^2)$ i u tu grupu spadaju *bubble sort*, *insertion sort* i *selection sort*. *Shell sort* je negde između pomenute dve klase (u zavisnosti od implementacije složenost mu varira od $O(n^2)$ do $O(n \log^2 n)$). Kompleksniji, brži algoritmi imaju složenost najgoreg slučaja $O(n \log n)$ i u tu grupu spadaju *heap* i *merge sort*. Algoritam *quick sort* ima složenost najgoreg slučaja $O(n^2)$

ali, pošto mu je složenost prosečnog slučaja $O(n \log n)$ i pošto u praksi pokazuje veoma dobre rezultate, ovaj algoritam ubraja se u grupu veoma brzih algoritama i koristi najčešće.

Može se dokazati da algoritmi koji sortiraju niz permutovanjem elemenata niza (većina nabrojanih funkcioniše upravo ovako) ne mogu imati bolju složenost od $O(n \log n)$.

Svi algoritmi sortiranja u nastavku teksta biće ilustrovani na primeru sortiranja niza celih brojeva u neopadajućem poretku (iako su univerzalni i mogu da se implementiraju za bilo kakav poredak bilo kog tipa podataka). Kao što je već pomenuto, nekoliko algoritama opisanih u nastavku se, zbog neefikasnosti, ne koristi u praksi, ali su ovde ipak prikazani jer koriste algoritamske tehnike i ideje koje mogu biti korisne ne samo u sortiranju, već i u drugim problemima.

U većini algoritama sortiranja, osnovni korak je razmena dva elemenata niza, te će se koristiti sledeća funkcija:

```
void razmeni(int a[], int i, int j)
{
  int tmp = a[i]; a[i] = a[j]; a[j] = tmp;
}
```

Opšti oblik funkcije za sortiranje niza elemenata tipa `int` je (drugi argument, broj elemenata niza mogao bi da bude i tipa `unsigned int` ili `size_t`, koji se obično koristi za veličine objekata):

```
void sort(int a[], int n);
```

Preduslov funkcije je da je nenegativna promenljiva `n` jednaka[8] dimenziji niza `a`, dok je postuslov da su elementi niza nakon primene funkcije sortirani[9], kao i da se (multi)skup elemenata niza nije promenio (ova invarijanta trivijalno je ispunjena kod svih algoritama zasnovanih na razmenama elemenata niza).

U daljem tekstu, kada se, kratkoće radi, kaže „niz `a[i,j]`“ misliće se na podniz niza `a` od indeksa `i` do indeksa `j`, uključujući i njih.

5.3.1 Bubble sort

Bubble sort algoritam u svakom prolazu kroz niz poredi uzastopne elemente i razmenjuje im mesta ukoliko su u pogrešnom poretku. Prolasci kroz niz ponavljaju se sve dok se ne napravi prolaz u kojem nije bilo razmena, što znači da je niz sortiran.

Primer 5.14. Prikažimo rad algoritma na primeru sortiranja niza *(6 1 4 3 9)*:

[8] Preciznije, preduslov je da je `n` manje od ili jednako alociranom broju elemenata niza. Sve funkcije u nastavku ovog poglavlja će deliti ovaj ili neki sličan preduslov koji obezbeđuje da tokom sortiranja ne dolazi do pristupa nedozvoljenoj memoriji.

[9] Preciznije, elementi između pozicija `0` i `n-1` su sortirani, što je zaista ceo niz ukoliko je `n` jednako dimenziju niza.

Prvi prolaz:
*(**6 1** 4 3 9) → (**1 6** 4 3 9), razmena jer je* $6 > 1$
*(1 **6 4** 3 9) → (1 **4 6** 3 9), razmena jer je* $6 > 4$
*(1 4 **6 3** 9) → (1 4 **3 6** 9), razmena jer je* $6 > 3$
*(1 4 3 **6 9**) → (1 4 3 **6 9**)*
Drugi prolaz:
*(**1 4** 3 6 9) → (**1 4** 3 6 9)*
*(1 **4 3** 6 9) → (1 **3 4** 6 9), razmena jer je* $4 > 3$
*(1 3 **4 6** 9) → (1 3 **4 6** 9)*
*(1 3 4 **6 9**) → (1 3 4 **6 9**)*
Treći prolaz:
*(**1 3** 4 6 9) → (**1 3** 4 6 9)*
*(1 **3 4** 6 9) → (1 **3 4** 6 9)*
*(1 3 **4 6** 9) → (1 3 **4 6** 9)*
*(1 3 4 **6 9**) → (1 3 4 **6 9**)*

Primetimo da je niz bio sortiran već nakon drugog prolaza, međutim, da bi se to utvrdilo, potrebno je bilo napraviti još jedan prolaz.

Naredna funkcija primenom algoritma *bubble sort* sortira niz `a` dužine `n`.

```
void bubblesort(int a[], int n)
{
  int bilo_razmena, i;
  do {
    bilo_razmena = 0;
    for (i = 0; i < n - 1; i++)
      if (a[i] > a[i + 1]) {
        razmeni(a, i, i+1);
        bilo_razmena = 1;
      }
  } while (bilo_razmena);
}
```

Uslov koji obezbeđuje parcijalnu korektnost je da ako `bilo_razmena` ima vrednost 0, nakon unutrašnje petlje, tj. pre provere uslova izlaska iz spoljašnje petlje, onda je niz `a` sortiran (tj. sortiran je njegov početni deo dužine `n`). Kada se spoljašnja petlja završi, vrednost promenljive `bilo_razmena` je 0, te prethodna implikacija obezbeđuje korektnost. Invarijanta unutrašnje petlje koja obezbeđuje navedeni uslov je to da ako promenljiva `bilo_razmena` ima vrednost 0, onda je deo niza `a[0, i]` sortiran. Po završetku unutrašnje petlje `i` ima vrednost `n-1`, pa ako je tada `bilo_razmena` jednako 0, sortiran je deo `a[0, n-1]`, tj. ceo prefiks dužine `n`. Pošto je algoritam zasnovan na razmenama elemenata niza, njegova invarijanta je i to da se (multi)skup elemenata niza ne menja tokom njegovog izvršavanja.

Svojstvo algoritma koje obezbeđuje zaustavljanje je da je nakon k-te iteracije spoljašnje petlje k-ti najveći elemenat na svojoj finalnoj, ispravnoj poziciji.

Bubble sort je na osnovu ovog svojstva i dobio ime (jer veliki elementi kao mehurići „isplivavaju“ ka kraju niza).

Što se tiče vremenske složenosti, najgori slučaj nastupa kada je niz sortiran u obratnom redosledu. Razmotrimo kako izgleda niz posle svakog izvršavanja unutrašnje petlje.

(6 5 4 3 2 1)
(5 4 3 2 1 ***6*** *)*
(4 3 2 1 ***5*** *6)*
(3 2 1 ***4*** *5 6)*
(2 1 ***3*** *4 5 6)*
(1 ***2*** *3 4 5 6)*

Pre svake iteracije, stanje niza je takvo da postoji obratno sortirani prefiks (početni deo) niza (na primer, pre četvrtog koraka to je *3 2 1*) iza kojeg sledi ispravno sortirani sufiks (završni deo) niza (na primer, pre četvrtog koraka to je *4 5 6*). U svakoj iteraciji prvi element obratno sortiranog prefiksa niza pridružuje se ispravno sortiranom sufiksu niza. Zaista, najveći element u tekućem obratno sortiranom prefiksu niza se prebacuje na kraj tog prefiksa, dok elementi ispred njega ostaju u međusobno neizmenjenom redosledu. Pošto se poslednja izmena (u ovom, najgorem slučaju) vrši na poslednjoj poziciji u tom prefiksu, u svakoj iteraciji prefiks se skraćuje za jedan. Zato se u prvoj iteraciji vrši $n-1$ razmena, u drugoj $n-2$ razmene itd, pa je ukupni broj i ukupan broj razmena $\frac{n(n-1)}{2}$, što pripada klasi $O(n^2)$. I za broj poređenja važi da je reda $O(n^2)$ (u svakoj iteraciji vrši se n poređenja). Algoritam radi u mestu, pa njegova dodatna prostorna složenost (prostorna složenost koja ne uključuje ulazne podatke) pripada klasi $O(1)$, a ukupna prostorna složenost pripada klasi $O(n)$.

Postoji različita unapređenja *bubble sort* (na primer, unutrašnja petlja `for` može se izvršavati samo do pozicije poslednje razmene u prethodnoj iteraciji), ali sve one imaju istu vremensku složenost, $O(n^2)$.

Algoritam *bubble sort* smatra se veoma lošim algoritmom i ne treba ga koristiti u praksi (kao, uostalom, nijedan algoritam čija je prosečna složenost veća od $O(n \log n)$). Kao prednost ovog algoritma navodi se njegova jednostavnost i to što je provera da li je niz već sortiran ugrađena u sam algoritam. To što se porede i razmenjuju uvek susedni elementi je istorijski moglo da bude korisno kod sortiranja elemenata memorije kojoj se može pristupati sekvencijalno (na primer, kod sortiranja podataka zapisanih na magnetnim trakama). S druge strane, mnogi autori navode ovaj algoritam kao primer lošeg algoritma iz kog se ništa korisno ne može naučiti i preporučuju njegovo izbacivanje iz kurseva programiranja.

5.3.2 Selection sort

Algoritam *selection sort* se ukratko može opisati na sledeći način: ako niz ima više od jednog elementa, zameni početni element sa najmanjim elementom niza i zatim analogno sortiraj ostatak niza (elemente iza početnog). Primeti-

mo da je ovaj opis suštinski rekurzivan, međutim, nije praktično upotrebljiv (i interesantan je samo vežba za pisanje rekurzivnih funkcija). Stoga ćemo u nastavku opisati samo iterativnu implementaciju. U iterativnoj implementaciji, niz se sortira tako što se u svakoj iteraciji na svoju poziciju dovodi sledeći po veličini element niza, tj. u i-toj iteraciji se i-ti po veličini element dovodi na poziciju i. Ovo se može realizovati tako što se pronađe pozicija m najmanjeg elementa od pozicije i do kraja niza i zatim se razmene element na poziciji i i element na poziciji m. Algoritam se zaustavlja kada se pretposlednji po veličini element dovede na pretposlednju poziciju u nizu.

Primer 5.15. Prikažimo rad algoritma na primeru sortiranja niza *(5 3 4 2 1)*.
*(.**5** 3 4 2 1)*, $i = 0$, $m = 4$, *razmena elemenata 5 i 1*
*(1 .**3** 4 2 5)*, $i = 1$, $m = 3$, *razmena elemenata 3 i 2*
*(1 2 .**4** 3 5)*, $i = 2$, $m = 3$, *razmena elemenata 4 i 3*
*(1 2 3 .**4** 5)*, $i = 3$, $m = 3$, *razmena elemenata 4 i 4*
(1 2 3 4 .5)

Pozicija najmanjeg elementa u nizu `a`, dužine `n`, počevši od pozicije `i` može se naći narednom funkcijom.

```
int poz_min(int a[], int n, int i)
{
  int m = i, j;
  for (j = i + 1; j < n; j++)
    if (a[j] < a[m])
      m = j;
  return m;
}
```

Uz korišćenje navedene funkcije, *selection sort* algoritam izgleda ovako:

```
void selectionsort(int a[], int n)
{
  int i;
  for (i = 0; i < n - 1; i++)
    razmeni(a, i, poz_min(a, n, i));
}
```

Invarijanta petlje je da su elementi niza `a[0, i]` sortirani, kao i da je svaki od njih jednak ili manji od svakog elemenata niza `a[i+1, n-1]` (na primer, poslednji, najveći element prvog dela može da bude jednak najmanjem elementu drugog dela). Pošto je algoritam zasnovan na razmenama, (multi)skup elemenata polaznog niza se (trivijalno) ne menja. Zaustavljanje je, takođe, trivijalno.

Primetimo da je broj razmena jednak $n - 1$, tj. $O(n)$. Međutim, broj poređenja je $O(n^2)$. Zaista, broj poređenja koja se izvrše u okviru funkcije `poz_min` jednak je `n-i-1`, pa je ukupan broj poređenja jednak $\sum_{i=0}^{n-2} n-i-1 = \sum_{i=1}^{n-1} i =$

$\frac{(n-1)\cdot n}{2}$. Algoritam radi u mestu, pa njegova dodatna prostorna složenost pripada klasi $O(1)$, a ukupna prostorna složenost pripada klasi $O(n)$.

Ukoliko se ne koristi pomoćna funkcija `poz_min`, algoritam se može implementirati na sledeći način.

```
void selectionsort(int a[], int n)
{
  int i;
  for (i = 0; i < n - 1; i++) {
    int m = i, j;
    for (j = i + 1; j < n; j++)
      if (a[j] < a[m])
        m = j;
    razmeni(a, i, m);
  }
}
```

Ponekad se sreće i naredna implementacija:

```
void selectionsort(int a[], int n)
{
  int i, j;
  for (i = 0; i < n-1; i++)
    for (j = i + 1; j < n; j++)
      if (a[j] < a[i])
        razmeni(a, i, j);
}
```

Napomenimo da je ova implementacija znatno neefikasnija od prethodne (iako je kôd kraći) jer se u najgorem slučaju osim $O(n^2)$ poređenja vrši i $O(n^2)$ razmena. Zbog toga bi, naravno, ovaj način implementacije algoritma trebalo izbegavati.

Naglasimo da se korišćenjem naprednijih struktura podataka, ideje algoritma *selection sort* mogu iskoristiti da se dobije algoritam složenosti $O(n \log n)$ (takozvani *heap sort* algoritam koji koristi strukturu podataka poznatu kao hip (engl. heap), koji nije opisan u ovoj knjizi).

Algoritam *selection sort* može se realizovati i tako što se najveći element dovede na desni kraj niza i zatim sortira prefiks bez tog poslednjeg elementa. Na taj način bi se u svakom koraku spoljne petlje na desnom kraju niza slagali jedan po jedan maksimalni element, što se isto događa i kod algoritma *bubble sort*. Međutim, razlika između ova dva algoritma je to što *bubble sort* to radi isključivo razmenama uzastopnih elemenata, dok *selection sort* isti efekat može postići samo jednom razmenom elemenata koji mogu biti i udaljeni.

5.3.3 Insertion sort

Insertion sort algoritam sortira niz tako što jedan po jedan element niza umeće na odgovarajuće mesto u do tada sortirani deo niza.

Primer 5.16. Prikažimo rad algoritma na primeru sortiranja niza *5 3 4 1 2*.
5. *3 4 1 2*
3 *5*. *4 1 2*
3 ***4*** *5*. *1 2*
1 *3 4 5*. *2*
1 ***2*** *3 4 5*.
Podebljanim slovima prikazani su elementi umetnuti na svoju poziciju.

Algoritam *insertion sort* može se opisati na sledeći način: ako niz ima više od jednog elementa, sortiraj sve elemente ispred poslednjeg, a zatim umetni poslednji u taj sortirani podniz. Na osnovu ovog opisa, dobija se sledeća rekurzivna implementacija.

```
void insertionsort(int a[], int n)
{
  if (n > 1) {
    insertionsort(a, n-1);
    umetni(a, n-1);
  }
}
```

Primetimo da je ova rekurzivna verzija neupotrebljiva za velike nizove. Naime, ukoliko se sortira niz dužine n, onda će na programskom steku da bude formirano n stek okvira, što nije sprovodivo za duže nizove (isto važi i za rekurzivnu varijantu algoritma *selection sort*.

Iz navedene verzije može se eliminisati rekurzija i time se dolazi do sledeće iterativne implementacije.

```
void insertionsort(int a[], int n)
{
  int i;
  for (i = 1; i < n; i++)
    umetni(a, i);
}
```

Invarijanta petlje je da je deo niza `a[0, i-1]` sortiran, kao i da se (multi)skup elemenata u nizu `a` ne menja. Za obezbeđivanje korektnosti važan je preduslov funkcije `umetni` da je sortiran deo niza `a[0, i-1]`, dok ona treba da obezbedi postuslov da je sortiran deo niza `a[0, i]` . Funkcija `umetni` može biti implementirana na različite načine. Jedna mogućnost je da se element menja sa svojim prethodnikom sve dok je prethodnik veći od njega.

```
void umetni(int a[], int i)
{
  int j;
  for(j = i; j > 0 && a[j] < a[j-1]; j--)
    razmeni(a, j, j-1);
}
```

Funkcija `umetni` poziva se $n-1$ put i to za vrednosti i od 1 do $n-1$, dok u najgorem slučaju (obratno sortiranog niza) ona izvršava i razmena i $2i$ poređenja. Zbog toga je ukupan broj razmena, kao i broj poređenja $O(n^2)$ (jer je $\sum_{i=1}^{n-1} i = \frac{n\cdot(n+1)}{2}$).

Efikasnija implementacija može se dobiti ukoliko se ne koriste razmene, već se zapamti element koji treba da se umetne, zatim se pronađe pozicija na koju treba da se umetne, svi elementi od te pozicije pomere se za jedno mesto udesno i na kraju se zapamćeni element upiše na svoje mesto:

```
void umetni(int a[], int i)
{
  int j, tmp = a[i];
  for (j = i; j > 0 && a[j-1] > tmp; j--)
    a[j] = a[j-1];
  a[j] = tmp;
}
```

Prilikom svake razmene koriste se tri dodele, pa se navedenom izmenom broj dodela smanjuje 3 puta (broj poređenja ostaje nepromenjen).

U sledećoj verziji funkcije `insertionsort` kôd pomoćne funkcije je integrisan:

```
void insertionsort(int a[], int n)
{
  int i;
  for (i = 1; i < n; i++) {
    int j, tmp = a[i];
    for (j = i; j > 0 && a[j-1] > tmp; j--)
      a[j] = a[j-1];
    a[j] = tmp;
  }
}
```

Broj poređenja može se smanjiti ukoliko se za pronalaženje ispravne pozicije elementa u sortiranom prefiksu umesto linearne pretrage koristi binarna. Međutim, broj dodela nije moguće smanjiti te se ukupno vreme neće smanjiti značajno. Zbog toga, nećemo prikazati implementaciju ovakvog rešenja.

Iako je algoritam *insertion sort* neefikasan prilikom sortiranja dugačkih nizova, može se pokazati da je kod kratkih nizova (nizova sa nekoliko desetina elemenata) on brži od naprednijih algoritama, tako da se u realnim implementacijama funkcija sortiranja koristi za sortiranje kratkih nizova. Algoritam *insertion sort* se može primeniti i na sortiranja drugih struktura podataka, ne samo na nizove. Na primer, može se primeniti na sortiranje lista ali i na sortiranje podataka uz pomoć binarnih stabala što dovodi do algoritma složenosti $O(n \log n)$ (takozvani *tree sort* algoritam), o čemu će biti više reči u poglavljima 6.1 i 6.6. Algoritam *insertion sort*, takođe, predstavlja osnovu algoritma *shell sort*, koji će biti opisan u nastavku. Dakle, iako je *insertion sort* algoritam koji ne treba koristiti za potrebe sortiranja dužih nizova, njegovo proučavanje i te kako ima smisla i poželjno je poznavati ga detaljno.

5.3.4 Shell sort

Najveći uzrok neefikasnosti kod *insertion sort* algoritma je slučaj malih elemenata koji se nalaze blizu kraja niza. Pošto se nalaze blizu kraja, oni se umeću u relativno dugačak niz, a pošto su mali umeću se na početak te je potrebno izvršiti pomeranje velikog broja elemenata kako bi se oni postavili na svoje mesto. *Shell sort*[10] popravlja ovo. Osnovni cilj je da se „skrati put“ ovakvih elemenata. *Shell sort* koristi činjenicu da *insertion sort* funkcioniše odlično kod nizova koji su „skoro sortirani“. Algoritam radi tako što se niz deli na veći broj kratkih kolona koje se sortiraju primenom *insertion sort* algoritma, čime se omogućava direktna razmena udaljenih elemenata. Broj kolona se zatim smanjuje, sve dok se na kraju *insertion sort* ne primeni na ceo niz. Međutim, do tada su „pripremni koraci“ deljenja na kolone doveli niz u „skoro sortirano“ stanje te se završni korak prilično brzo odvija.

Primer 5.17. Ilustrujmo jednu varijantu *Shell sort* algoritma na primeru sortiranja niza *(9, 10, 16, 8, 5, 11, 1, 12, 4, 6, 13, 7, 14, 3, 15, 2)*.

U prvoj fazi podelimo niz na 8 kolona.

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

i primenimo *insertion sort* na sortiranje svake kolone ponaosob.

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

Ovim je dobijen niz *(4, 6, 13, 7, 5, 3, 1, 2, 9, 10, 16, 8, 14, 11, 15, 12)*. Primetimo da je ovim veliki broj malih elemenata (na primer, 2) kroz samo jednu operaciju razmene došao u prvu polovinu niza.

U sledećoj fazi delimo niz na 4 kolone

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

[10] Nazvan po D. L. Šelu koji ga je prvi opisao 1959. godine.

i primenimo *insertion sort* na sortiranje svake kolone ponaosob.

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

Ovim je dobijen niz *(4 3 1 2 5 6 13 7 9 10 15 8 14 11 16 12)*.

U sledećoj fazi delimo niz na dve kolone.

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

i primenimo *insertion sort* na sortiranje svake kolone ponaosob.

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

Ovim se dobija niz *(1, 2, 4, 3, 5, 6, 9, 7, 13, 8, 14, 10, 15, 11, 16, 12)*.

Na kraju se dobijeni sortira primenom *insertion sort* algoritma, čime se dobija niz *(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16)*.

Napomenimo da je podela na kolone samo fiktivna operacija i da se ona u implementaciji izvodi tako što se prilikom umetanja elementa ne razmatraju susedni elementi već elementi na rastojanju `gap` gde `gap` označava tekući broj kolona.

```
void shellsort(int a[], int n)
{
  int gap, i, j;
  for (gap = n/2; gap > 0; gap /= 2)
    for (i = gap; i < n; i++)
      for (j = i-gap; j >= 0 && a[j] > a[j+gap]; j -= gap)
        razmeni(a, j, j + gap);
}
```

S obzirom na to da u poslednjoj iteraciji spoljne petlje `gap` ima vrednost 1, algoritam u poslednjoj iteraciji izvodi običan *insertion sort* algoritam, te se korektnost *Shell sort* algoritma oslanja na već diskutovanu korektnost *insertion*

sort algoritma i invarijante da je sve vreme (multi)skup elemenata niza nepromenjen (koja je trivijalno ispunjena jer se algoritam zasniva na razmenama).

Ono što se može proizvoljno određivati je broj kolona na koji se vrši deljenje niza u fazama (broj kolona se obično označava sa `gap` i ovaj niz se u literaturi često označava `gap sequence`). Originalni Šelov predlog (koji je i korišćen u prethodnom primeru i implementaciji je $\lfloor n/2 \rfloor, \lfloor n/4 \rfloor, \ldots, \lfloor n/2^k \rfloor, \ldots, 1$). Kako bi se garantovala korektnost, na kraju je potrebno primeniti *insertion sort* bez podele na kolone (tj. poslednji član sekvence mora biti 1). U zavisnosti od sekvence varira i složenost najgoreg slučaja. Originalna sekvenca ima složenost $O(n^2)$ dok postoje druge sekvence koje garantuju složenost $O(n^{\frac{3}{2}})$, $O(n^{\frac{4}{3}})$ pa i $O(n \log^2 n)$. Napomenimo da je empirijski utvrđeno da postoje relativno nepravilne sekvence koje u praksi daju bolje rezultate (imaju najbolju složenost prosečnog slučaja). I ovaj algoritam radi u mestu, pa njegova dodatna prostorna složenost pripada klasi $O(1)$, a ukupna prostorna složenost pripada klasi $O(n)$.

5.3.5 Merge sort

Algoritam *merge sort* deli niz na dva dela čije se dužine razlikuju najviše za 1 (ukoliko je dužina niza paran broj, onda su ova dva dela jednakih dužina), rekurzivno sortira svaki od njih i zatim objedinjuje sortirane polovine. Problematično je što je za objedinjavanje neophodno koristiti dodatni, pomoćni niz. Na kraju se objedinjeni niz kopira u polazni niz. Izlaz iz rekurzije je slučaj jednočlanog niza (slučaj praznog niza ne može da nastupi).

Ključna operacija u ovom algoritmu je operacija objedinjavanja opisana u poglavlju 5.2.3, primer 5.12.

Funkcija `mergesort_` *merge sort* algoritmom sortira deo niza `a[l, d]`, uz korišćenje niza `tmp` kao pomoćnog.

```
void mergesort_(int a[], int l, int d, int tmp[])
{
  if (l < d) {
    int i, j;
    int n = d - l + 1, s = l + n/2;
    int n1 = n/2, n2 = n - n/2;

    mergesort_(a, l, s-1, tmp);
    mergesort_(a, s, d, tmp);
    merge(a + l, n1, a + s, n2, tmp);

    for (i = l, j = 0; i <= d; i++, j++)
      a[i] = tmp[j];
  }
}
```

Promenljiva `n` čuva broj elemenata koji se sortiraju u okviru ovog rekurzivnog poziva, a promenljiva `s` čuva središnji indeks u nizu između `l` i `d`. Rekurzivno se sortira `n1 = n/2` elemenata između pozicija `l` i `s-1` i `n2 = n - n/2` elemenata između pozicija `s` i `d`. Nakon toga, sortirani podnizovi objedinjuju se u pomoćni niz `tmp`. Primetimo korišćenje pokazivačke aritmetike: adresa početka prvog sortiranog podniza koji se objedinjuje je `a+l`, dok je adresa početka drugog `a+s`.

Pomoćni niz može se pre početka sortiranja dinamički alocirati i koristiti kroz rekurzivne pozive:

```
void mergesort(int a[], int n)
{
  int *tmp = malloc(n*sizeof(int));
  if (tmp == NULL) {
    printf("Nedovoljno memorije.\n");
    return;
  }
  mergesort_(a, 0, n-1, tmp);
  free(tmp);
}
```

Napravimo jednu paralelu između rekurzivne varijante algoritama *selection sort* i algoritma *merge sort*. Osnovna ideja *selection sort* algoritma je da se jedan element postavi na svoje mesto i da se zatim ista metoda rekurzivno primeni na niz koji je za jedan kraći od polaznog. S obzirom na to da je pripremna akcija zahtevala $O(n)$ operacija, ovaj rekurzivni pristup za vremensku složenost daje jednačinu $T(n) = T(n-1) + O(n)$, te $T(n)$ pripada klasi $O(n^2)$. S druge strane, kod *merge sort* algoritma sortiranje se svodi na sortiranje dva podniza polaznog niza dvostruko manje dimenzije. S obzirom na to da korak objedinjavanja dva sortirana niza zahteva $O(n)$ operacija, dobija se jednačina $T(n) = 2T(n/2) + O(n)$, pa (na osnovu teoreme 4.1) $T(n)$ pripada klasi $O(n \log n)$. Dakle, značajno je efikasnije da se problem dimenzije n svodi na dva problema dimenzije `n/2` nego na jedan problem dimenzije `n-1` — ovo je osnovna ideja „podeli i vladaj“ (engl. divide-and-conquer) algoritama, u koje spada i algoritam *merge sort*. Pošto rekurzija nije repna (postoje dva rekurzivna poziva i korak objedinjavanja nakon njih), njegova implementacija najčešće ostaje rekurzivna. Broj stek okvira na programskom steku logaritamski zavisi od broja elemenata niza (primetimo da se u funkciji drugi rekurzivni poziv vrši tek kada je prvi završen), te ne postoji opasnost da dođe do prekoračenja steka. U vezi sa tim je i prostorna složenost algoritma *merge sort*: algoritam koristi pomoćni niz veličine $O(n)$ i kreira najviše $O(\log n)$ stek okvira, te i njegova prostorna složenost i njegova dodatna prostorna složenost pripadaju klasi $O(n)$.

5.3.6 Quick sort

Kao i algoritam *merge sort* i algoritam *quick sort* pripada grupi *podeli i vladaj* algoritama. Slično kao kod algoritma *selection sort*, u svakom koraku težimo da neki element niza dovedemo na svoju poziciju u sortiranom nizu. Umesto da to obavezno bude minimum (ili maksimuma), u algoritam *quick sort* u svakom koraku na svoje mesto dovodi neki element (obično nazivan *pivot*) koji je relativno blizu sredine niza. Međutim, da bi nakon toga, problem mogao biti sveden na sortiranje dva manja podniza, potrebno je prilikom dovođenja pivota na svoje mesto grupisati sve elemente manje od njega ili jednake njemu levo od njega, a sve elemente veće od njega desno od njega (ako se niz sortira neopadajuće). To pregrupisavanje elemenata niza, *korak particionisanja* (već opisano u poglavlju 5.2.3, primer 5.13) ključni je korak algoritma *quick sort* .

Quick sort algoritam može se implementirati na sledeći način.

```
void quicksort_(int a[], int l, int d)
{
  if (l < d) {
    razmeni(a, l, izbor_pivota(a, l, d));
    int p = particionisanje(a, l, d);
    quicksort_(a, l, p - 1);
    quicksort_(a, p + 1, d);
  }
}
```

Poziv `quicksort_(a, l, d)` sortira deo niza `a[l, d]`. Funkcija `quicksort` onda se jednostavno implementira:

```
void quicksort(int a[], int n)
{
  quicksort_(a, 0, n-1);
}
```

Funkcija `izbor_pivota` bira za pivot neki element niza `a[l, d]` i vraća njegov indeks (u nizu `a`). Pozivom funkcije `razmeni` pivot se postavlja na poziciju `l`. Funkcija `particionisanje` vrši particionisanje niza (pretpostavljajući da se pre particionisanja pivot nalazi na poziciji `l`) i vraća poziciju na kojoj se nalazi pivot nakon particionisanja. Funkcija se poziva samo za nizove koji imaju više od jednog elementa, te joj je preduslov da je `l` manje od `d`. Postuslov funkcije `particionisanje` je da je (multi)skup elemenata niza `a` nepromenjen nakon njenog poziva, međutim njihov redosled je takav da su svi elementi niza `a[l, p-1]` manji od ili jednaki elementu `a[p]`, dok su svi elementi niza `a[p+1, d]` veći od ili jednaki elementu `a[p]`.

Na osnovu teoreme 4.1, kako bi se dobila jednačina $T(n) = 2T(n/2)+O(n)$ i vremenska složenost $O(n \log n)$, funkcija `particionisanje` (tj. korak particionisanja) treba da bude izvršena u linearnom vremenu $O(n)$. U nastavku će biti

prikazano nekoliko algoritama particionisanja koji ovo zadovoljavaju. Dalje, potrebno je da pozicija pivota nakon particionisanja bude blizu sredini niza (kako bi dužina dva podniza na koje se problem svodi bilo približno jednaka $n/2$). Međutim, određivanje središnjeg člana u nizu brojeva (što predstavlja idealnu strategiju za funkciju `izbor_pivota`) je problem koji nije značajno jednostavniji od samog sortiranja. S obzirom na to da se očekuje da je implementacija funkcije brza (obično $O(1)$), obično se ne garantuje da će za pivot biti izabiran upravo srednji član, već se koriste heuristike koje za pivot biraju elemente koji nisu daleko od središnje pozicije u nizu. Naglasimo da se za svaku strategiju izbora pivota (koja ne koristi slučajno izabrane brojeve) može konstruisati niz takav da u svakom koraku izbor pivota bude najgori mogući — onaj koji deli niz na nizove dužine `0` i `n-1`, što dovodi do jednačine $T(n) = T(n-1) + O(n)$ i kvadratne vremenske složenosti i linearne dodatne prostorne složenosti (u odnosu na dužinu niza). Međutim, većina strategija je takva da se u prosečnom slučaju može očekivati da se dužine podnizova ne razlikuju mnogo, te daju prosečnu vremensku složenost $O(n \log n)$ i prosečnu dodatnu prostornu složenost $O(\log n)$ (jer algoritam radi u mestu i na programskom steku se u proseku formira $O(\log n)$ stek okvira).

U praksi, najbolje rezultate kod sortiranja dugačkih nizova daje upravo algoritam *quick sort*. Međutim, za sortiranje kraćih nizova naivni algoritmi (na primer, *insertion sort*) mogu da se pokažu pogodnijim. Većina realnih implementacija *quick sort* algoritma koristi hibridni pristup — izlaz iz rekurzije se vrši kod nizova koji imaju nekoliko desetina elemenata i na njih se primenjuje *insertion sort*.

Implementacije particionisanja. Korak particionisanja može se implementirati kao što je to opisano u poglavlju 5.2.3, primer 5.13.

Korak particionisanja može se implementirati i na sledeći način:

```
int particionisanje(int a[], int l, int d)
{
  int p = l, j;
  for (j = l+1; j <= d; j++)
    if (a[j] < a[l])
      razmeni(a, ++p, j);

  razmeni(a, l, p);
  return p;
}
```

Invarijanta petlje je da je (multi)skup elemenata u nizu `a` nepromenjen, kao i da se u nizu `a` na poziciji `l` nalazi pivot, da su elementi `a[l+1, p]` manji od pivota, dok su elementi `a[p+1, j-1]` veći od ili jednaki pivotu. Nakon završetka petlje, `j` ima vrednost `d+1`, te su elementi `a[p+1, d]` veći od ili jednaki pivotu. Kako bi se ostvario postuslov funkcije `particionisanje`, vrši se još razmena

pivota i elementa na poziciji `p` — time pivot dolazi na svoje mesto (na poziciju `p`).

Treći način implementiranja particionisanja zasnovan je na Dejkstrinom algoritmu „trobojke“ (Dutch National Flag Problem). U ovom slučaju, radi se malo više od onoga što postuslov striktno zahteva — niz se permutuje tako da prvo idu svi elementi striktno manji od pivota, zatim sva pojavljivanja pivota i na kraju svi elementi striktno veći od pivota.

```
int particionisanje(int a[], int l, int d)
{
  int pn = l-1, pp = d+1, pivot = a[l], t = l;
  while (t < pp) {
    if (a[t] < pivot)
      razmeni(a, t++, ++pn);
    else if (a[t] > pivot)
      razmeni(a, t, --pp);
    else
      t++;
  }
  return pn+1;
}
```

Invarijanta petlje je da se (multi)skup elemenata niza ne menja, da su svi elementi niza `a[l, pn]` manji od pivota, da su svi elementi niza `a[pn+1, t-1]` jednaki pivotu i da su svi elementi niza `a[pp, d]` veći od pivota. Kada se petlja završi, važi da je `t` jednako `pp` tako da su svi elementi niza `a[l, pn]` manji od pivota, niza `a[pn+1, pp-1]` jednaki pivotu, a niza `a[pp, d]` veći od pivota.

Izbor pivota. Kao što je već rečeno, iako je poželjno da se pivot izabere tako da podeli niz na dve potpuno jednake polovine, to bi trajalo previše tako da se obično pribegava heurističkim rešenjima. Ukoliko se može pretpostaviti da su elementi niza slučajno raspoređeni (što se uvek može postići ukoliko se pre primene sortiranja niz permutuje na slučajan način), bilo koji element niza se može uzeti za pivot. Na primer,

```
int izbor_pivota(int a[], int l, int d)
{
  return l;
}
```

ili

```
int izbor_pivota(int a[], int l, int d)
{
  return slucajan_broj(l, d);
```

```
}
```

Nešto bolje performanse mogu se postići ukoliko se, na primer, za pivot uzima srednji od tri slučajno izabrana elementa niza.

5.3.7 Sistemska implementacija algoritma quick sort

Funkcija `qsort` (deklarisana u zagavlju `<stdlib.h>`) obezbeđuje generičku funkciju za sortiranje — može da se koristi za sortiranje podataka bilo kog tipa (u sličnom duhu kao generička funkcija za proveravanje da li je niz sortiran u skladu sa zadatim poretkom, opisana u poglavlju 5.1.3). U nekim implementacijama zasnovana je na algoritmu *quick sort*, ali standard jezika to ne zahteva. Način korišćenja ove funkcije veoma je blizak načinu korišćenja funkcije `bsearch`.

```
void qsort(void *base, size_t number, size_t size,
           int (*compare)(const void *e1, const void *e2));
```

Argumenti funkcije imaju sledeću ulogu:

`base` je pokazivač na početak niza koji se sortira; on je tipa `void *` kako bi mogao da prihvati pokazivač na bilo koji konkretan tip;

`number` je broj elemenata niza;

`size` je veličina jednog elementa u bajtovima; ona je potrebna kako bi funkcija mogla da izračuna adresu svakog pojedinačnog elementa niza;

`compare` je pokazivač na funkciju (često definisanu od strane korisnika) koja vrši poređenje dve vrednosti zadatog tipa, analogno funkcijama za poređenje iz poglavlja 5.1; da bi funkcija imala isti prototip za sve tipove, njeni argumenti su tipa `void *` (zapravo — `const void *` jer vrednost na koju ukazuje ovaj pokazivač ne treba da bude menjana u okviru funkcije `compare`). Funkcija vraća jednu od sledećih vrednosti:

- < 0, ako je u izabranom poretku vrednost na koju ukazuje prvi argument manja od vrednosti na koju ukazuje drugi argument (što znači da u sortiranom nizu vrednost na koju ukazuje prvi argument treba da bude ispred vrednosti na koju ukazuje drugi argument);
- 0, ako su u izabranom poretku vrednosti na koju ukazuju prvi i drugi argument jednake;
- > 0, ako je u izabranom poretku vrednost na koju ukazuje prvi argument veća od vrednosti na koju ukazuje drugi argument.

Niz se sortira u skladu sa relacijom poretka koja je zadata funkcijom poređenja `compare`. Nad podacima istog tipa mogu se definisati različita uređenja, na primer, celi brojevi mogu se sortirati neopadajuće ili nerastuće.

Primer 5.18. Naredni program gradi slučajan niz celih brojeva i uređuje ga rastuće.

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

void ispisi(int a[], int n)
{
  int i;
  for (i = 0; i < n; i++)
    printf("%d ", a[i]);
  printf("\n");
}

int poredi_brojeve(const void *pa, const void *pb)
{
  int a = *(int *)pa, b = *(int *)pb;
  if (a < b)
    return -1;
  else if (a > b)
    return 1;
  else
    return 0;
}

#define MAX 10
int main()
{
  int i, niz[MAX];
  for(i = 0; i < MAX; i++)
    niz[i] = rand() % MAX;
  ispisi(niz, MAX);
  qsort(niz, MAX, sizeof(int), poredi_brojeve);
  ispisi(niz, MAX);
  return 0;
}
```

Primer 5.19. Naredni program leksikografski sortira argumente komandne linije (kao niske).

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int poredi_niske(const void *pa, const void *pb)
```

```
{
  return strcmp(*(char **)pa, *(char **)pb);
}

int main( int argc, char **argv )
{
  int i;
  argv++; argc--;   /* argv[0] se ne sortira */
  qsort(argv, argc, sizeof(char *), poredi_niske);
  for(i = 0; i < argc; i++)
    printf("%s ", argv[i]);
  printf("\n");
}
```

Navedena funkcija `main` ispisuje:

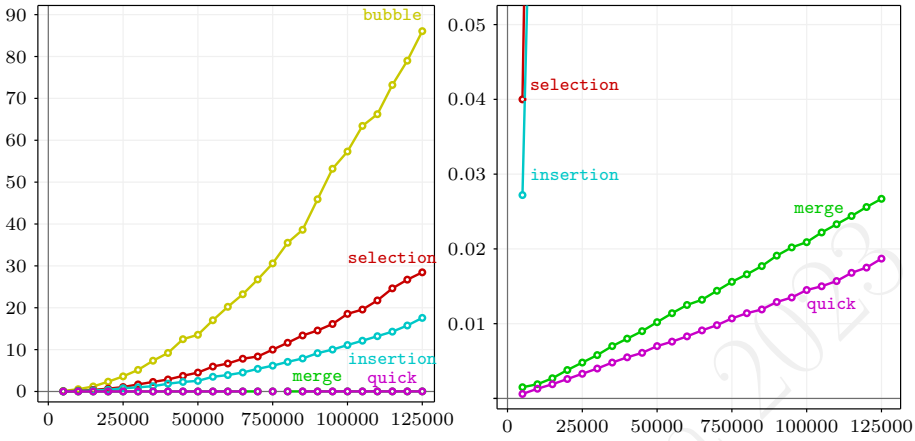
```
# ./qsort po jutru se dan poznaje
# dan jutru po poznaje se
```

5.3.8 Eksperimentalno poređenje algoritama sortiranja

Analiza vremenske (pa i prostorne) složenosti zasnovana na najgorem slučaju nekad može biti neadekvatna ili varljiva. U nekim situacijama kritično je znati da se algoritam nikada neće izvršavati duže od neke zadate granice, ali je u nekim situacijama dovoljno znati da se algoritam *u proseku* izvršava dovoljno brzo. Tako, na primer, algoritam *quick sort* ima složenost najgoreg slučaja $O(n^2)$, ali se u proseku ponaša bolje od algoritma *merge sort* koji ima složenost najgoreg slučaja $O(n \log n)$.

Prosečna složenost može se za neke algoritme izračunati rigorozno, analogno kao u analizi najgoreg slučaja. Međutim, često je to veoma teško i umesto rigoroznog izračunavanja pribegava se *aproksimiranju* vremena izvršavanja na osnovu pojedninačnih merenja. Da bi aproksimacija bila što kvalitetnija, potrebno je merenja sprovesti za razne veličine ulaza i za mnogo pojedinačnih ulaza. Na primer, za analizu složenosti algoritama za sortiranje, bilo bi potrebno primeniti ih na nizove različitih dužina i na veliki broj takvih nizova. U nastavku je prikazana eksperimentalna analiza efikasnosti nekoliko algoritama za sortiranje prikazanih u ranijem tekstu.

Algoritmi su primenjeni na nizovima celih brojeva. Korišćena je po jedna varijanta (najbolja koja je prikazana u ovoj glavi) algoritma *bubble sort*, *selection sort*, *insertion sort*, *merge sort*, *quick sort*. Za svaku dužinu niza i za svaki algoritam, generisano je po 10 nizova pseudoslučajnih brojeva i nad njima je, na istom računaru, vršeno sortiranje i mereno vreme izvršavanja (na način opisan u poglavlju 3.1.1). Na grafikonu na slici 5.2 (levo) prikazani su rezultati merenja za nizove dužine 5000, 10000, 15000, ..., 125000. Vreme izvršavanja algoritma *bubble sort* bliži se granici od 100s, dok je vreme izvršavanja algo-



Slika 5.2: Aproksimacija vremena izvršavanja algoritama za sortiranje za različite dužine ulaznih nizova celih brojeva Vreme je na oba grafikona izraženo u sekundama.

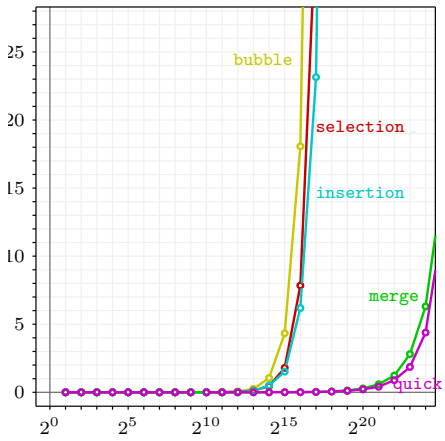
ritama *merge sort* i *quick sort* teško razlučivo od 0s. Ova slika pokazuje da su grafici algoritama koji imaju istu (prosečnu) složenost (u ovom slučaju algoritmi *bubblesort*, *selection sort* i *insertion sort* imaju prosečnu vremensku složenost $O(n^2)$) grupisani i, kako vrednost ulaza raste, sve su dalji od grafika algoritama iz druge klase. Ponašanje algoritama *merge sort* i *quick sort* (koji imaju prosečnu složenost $O(n \log n)$) može se jasnije razaznati tek na grafikonu na slici 5.2 (desno), sa drugačijom vremenskom skalom. Primetimo da je grafikon za *quick sort* u grupi sa algoritmom *merge sort*, a ne sa algoritmima koji imaju prosečnu složenost $O(n^2)$. Naime, *quick sort* ima složenost u najgorem slučaju $O(n^2)$, ali u prosečnom slučaju – koji aproksimira opisani eksperiment – ima složenost $O(n \log n)$. Štaviše, *quick sort* daje najbolje ponašanje u odnosu na sve razmatrane algoritme. Vizuelna uporedna analiza ponašanja dve grupe algoritama moguća je, donekle, tek kada se broj elemenata niza menja eksponencijalno (sortiranje se vrši nad nizovima dužine 2^1, 2^2, 2^3, …, 2^{24}) i kada su rezultati prikazani na logaritamskoj skali, kao na slici 5.3.

Pitanja i zadaci za vežbu

Pitanje 5.22. *Koje su dve operacije osnovne u većini algoritama sortiranja?*

Pitanje 5.23. *Navesti barem dva algoritma sortiranja iz grupe „podeli i vladaj“ i ukratko objasniti kako funkcionišu.*

Pitanje 5.24. *Navesti primer algoritma za sortiranje čija se složenost u prosečnom i u najgorem slučaju razlikuju.*



Slika 5.3: Aproksimacija vremena izvršavanja (u sekundama) algoritama za sortiranje za različite dužine ulaznih nizova celih brojeva (dužina ulaza je na eksponencijalnoj skali).

Pitanje 5.25. *Opisati osnovnu ideju algoritma* selection sort. *Prikazati stanje niza (nakon svake razmene) prilikom izvršavanja* selection sort *algoritma za niz* `5 3 4 2 1`.
Šta važi nakon i-tog prolaska kroz petlju u algoritmu selection sort*?*
Koji je slučaj najgori za algoritam sortiranja selection sort*?*
Kolika je složenost algoritma selection sort *u najgorem i u prosečnom slučaju?*

Pitanje 5.26. *Opisati osnovnu ideju algoritma* insertion sort. *Prikazati stanje niza prilikom izvršavanja* insertion sort *algoritma za niz* `5 3 4 2 1`.
Šta važi nakon i-tog prolaska kroz petlju u algoritmu insertion sort*?*
Kolika je složenost algoritma insertion sort *u najgorem i u prosečnom slučaju?*

Pitanje 5.27. *Opisati osnovnu ideju algoritma* merge sort.

Pitanje 5.28. *Opisati osnovnu ideju algoritma* quick sort.
Koja je složenost koraka particionisanja (za niz od n elemenata) koje se koristi u algoritmu quicksort?

Pitanje 5.29. *U okviru algoritma* quick sort*, kada se izabere pivot, potrebno je izvršiti:* (a) *permutovanje elemenata niza;* (b) *particionisanje elemenata niza;* (c) *invertovanje elemenata niza;* (d) *brisanje elemanata niza.*

Pitanje 5.30. *Kod algoritma* merge sort *je:*
(a) *jednostavno razdvajanje na dva dela niza koji se sortira, ali je komplikovano spajanje;*

(b) *komplikovano razdvajanje na dva dela niza koji se sortira, ali je jednostavno spajanje;*
(c) *jednostavno je i razdvajanje na dva dela niza koji se sortira i njihovo spajanje;*
(d) *komplikovano je i razdvajanje na dva dela niza koji se sortira i njihovo spajanje.*

Kod algoritma quick sort *je:*
(a) *jednostavno razdvajanje na dva dela niza koji se sortira, ali je komplikovano spajanje;*
(b) *komplikovano razdvajanje na dva dela niza koji se sortira, ali je jednostavno spajanje;*
(c) *jednostavno je i razdvajanje na dva dela niza koji se sortira i njihovo spajanje;*
(d) *komplikovano je i razdvajanje na dva dela niza koji se sortira i njihovo spajanje.*

Pitanje 5.31. *Koji ulazni niz predstavlja najgori slučaj za algoritam* quick sort*, ako se za pivot uzima prvi element dela niza koji se sortira?*

Pitanje 5.32. *Koja je složenost u najgorem slučaju algoritma* quick sort*:*
(a) *ako se za pivot uzima prvi element niza?*
(b) *ako se za pivot uzima poslednji element niza?*
(c) *ako se za pivot uzima srednji (po indeksu) element niza?*

Pitanje 5.33. *Da li se može popraviti efikasnost algoritma* quick sort *nekim specifičnim algoritmom za izbor pivota? Koji algoritmi za to se obično koriste? Da li se njima poboljšava i složenost najgoreg slučaja?*

Pitanje 5.34. *Ako se funkcija* `qsort` *iz standardne biblioteke koristi za sortiranje niza struktura tipa* `S` *po članu* `kljuc` *tipa* `int`*, definisati funkciju za poređenje koju treba proslediti funkciji* `qsort`*. Upotrebiti je da bi se sortirao niz.*

Zadatak 5.15. *Napisati program koji iz datoteke čije se ime zadaje kao argument komandne linije, čita prvo broj elemenata niza pa zatim i elemente niza (celi brojevi). Ovaj niz sortirati pozivom funkcije* `qsort` *a zatim za dodatnih m unetih celih brojeva sa standradnog ulaza proveriti, pozivom funkcije* `bsearch`*, da li se nalaze u nizu ili ne i ispisati odgovarajuće poruke.*

Zadatak 5.16. *Napisati program koji sa standardnog ulaza učitava prvo broj studenata, a zatim i podatke o studentima (ime studenata – niska dužine do* 30 *karaktera i broj indeksa studenta – ceo broj), sortira ih po imenu studenta leksikografski (pozivom funkcije* `qsort`*) i nakon toga (pozivom funkcije* `bsearch`*) određuje ime studenata čije brojeve indeksa korisnik zadaje sa standardnog ulaza.*

Zadatak 5.17. *Tačka je predstavljena svojim x i y koordinatama (celi brojevi). Sa standardnog ulaza se učitava prvo broj tačaka a zatim koordinate tačaka. Dobijeni niz struktura sortirati pozivom bibliotečke funkcije* `qsort`*. Niz sortirati po x koordinati, a ako neke dve tačke imaju istu x koordinatu onda ih sortirati po y koordinati.*

Zadatak 5.18. *Napisati program koji omogućava učitavanje artikala iz datoteke koja se zadaje prvim argumentom komandne linije. Jedan artikal definisan je strukturom*

```
typedef struct {
  char naziv[50];
  float tezina;
  float cena;
} Artikal;
```

Datoteka je ispravno formatirana i sadrži najviše 500 artikala. Program treba da, nakon učitavanja, ispiše sve artikle sortirane po zadatom kriterijumu. Kriterijum se zadaje kao drugi argument komandne linije i može biti: `i` *- sortirati po nazivu;* `t` *- sortirati po težini;* `c` *- sortirati po ceni. Ako je prisutna i opcija* `o` *sotiranje treba da bude u nerastućem redosledu.*

Podrazumeva se da je sortiranje u rastućem redosledu, a da se za sortiranje po imenu koristi leksikografski poredak (dozvoljeno je korišćenje funkcije `strcmp`*). Koristiti generičku funkciju za sortiranje* `qsort` *iz standardne biblioteke.*

Zadatak 5.19. *Sa standardog ulaza se zadaje ime tekstualne datoteke koja sadrži podatke o artiklima prodavnice. Datoteka je u formatu:* `<bar kod> <naziv artikla> <proizvodjac> <cena>`*. Broj artikala u datoteci nije unapred poznat. Učitati podatke o artiklima u niz i sortirati niz na osnovu bar-kodova, korišćenjem ručne implementacije algoritma quick sort. Zatim se sa standardnog ulaza unose bar kodovi artikala sve dok se ne unese 0. Izračunati ukupnu cenu unetih proizvoda (proizvode pretraživati ručno implementiranim algoritmom binarne pretrage).*

Zadatak 5.20. *Argument programa je putanja do tekstualne datoteke koja sadrži isključivo cele brojeve. Napisati program koji pronalazi i ispisuje na standardnom izlazu dva broja iz date datoteke koja se najmanje razlikuju (ako ima više parova koji se isto razlikuju, ispisati ih sve).*

Zadatak 5.21. *Za svaki rad naučnika poznat je broj citata. H-indeks je najveći broj h takav da naučnik ima bar h radova sa bar h citata. Napisati program koji u složenosti $O(n \log n)$ određuje H-indeks naučnika koji ima n radova. Da li se složenost može smanjiti na $O(n)$?*

Zadatak 5.22. *Poznati su termini (sat i minut) početka i završetka n časova. Napiši program koji određuje najmanji broj učionica potrebnih da se ti časovi rasporede (svi časovi moraju biti održani u zakazanim terminima).*

Zadatak 5.23. *U nizu celih brojeva odrediti najbrojniji podskup elemenata koji se mogu urediti u niz uzastopnih celih brojeva. Na primer, za niz* $4, 8, 1, -6, 9, 5, -9, 10, -1, 3, 0, 1, 2$ *treba prikazati* $-1, 0, 1, 2, 3, 4, 5$*. Ako ima više takvih podskupova, prikazati prvi (onaj u kojem su brojevi najmanji).*

Zadatak 5.24. *Dato je* N *parova tačaka koje predstavljaju krajeve duži u prostoru (tačke imaju celobrojne koordinate). Ispisati koliko različitih dužina duži se pojavljuje u zadatom skupu duži (obratiti pažnju na preciznost izračunavanja i poređenja realnih brojeva).*

Zadatak 5.25. *Napisati program koji određuje najbrojniji element koji se javlja u učitanom nizu brojeva. Memorijska složenost treba da je* $O(n)$*, a vremenska* $O(n \log n)$*.*

Zadatak 5.26. *Dat je niz koji sadrži cele brojeve i ceo broj* D*. Napiši program koji određuje koji se od nekoliko datih nizova može od polaznog dobiti razmenama elemenata koji su tačno na rastojanju* D*.*

Zadatak 5.27. *Napisati program koji u datom nizu niski određuje i ispisuje sve parove niski koji su anagrami (dve niske su anagrami ako se jedna od druge može dobiti permutacijom redosleda karaktera).*

Zadatak 5.28. *Napisati program koji učitava niz mejl adresa i određuje koliko je među njima jedinstvenih adresa (onih koji se dobiju kada se eliminišu duplikati).*

Zadatak 5.29. *Sportisti su skakali u dalj i u vis. Smatraćemo da je sportista ostvario odličan rezultat ako ne postoji neki drugi sportista koji je skočio od njega više i u jednoj i u drugoj disciplini. Ako su poznati rezultati* n *sportista, napisati program koji određuje broj onih koji su postigli odličan rezultat.*

5.4 Jednostavni algebarsko-numerički algoritmi

U ovom poglavlju biće prikazani neki algebarsko-numerički algoritmi u kojima je direktno izračunavanje („grubom silom“) zamenjeno efikasnijim.

5.4.1 Najveći zajednički delilac

Izračunavanje najvećeg zajedničkog delioca dva broja a i b, pri čemu je $a \geq b$, može se izračunati korišćenjem Euklidovog algoritma. Označimo sa $a \operatorname{div} b$ i $a \bmod b$ redom celobrojni količnik i ostatak pri deljenju brojeva a i b (ako je $a = b \cdot q + r$, tako da je $0 \leq r < b$, tada je $a \operatorname{div} b = q$ i $a \bmod b = r$). Algoritam razmatra dva slučaja:

bazni slučaj: ako je $b = 0$ tada je $nzd(a, 0) = a$,

rekurzivni korak: za $b > 0$ važi: $nzd(a, b) = nzd(b, a \bmod b)$.

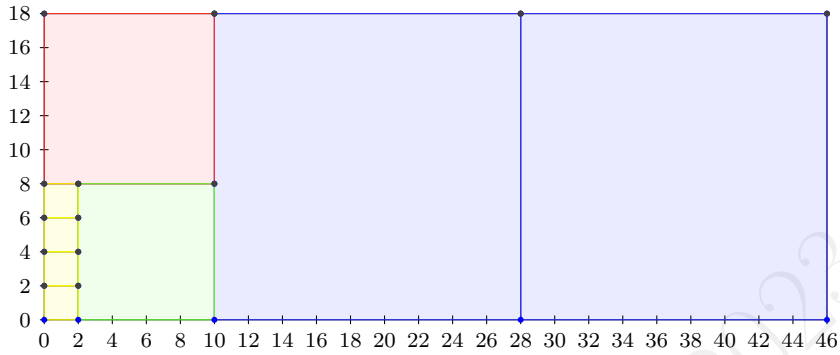
Na primer $nzd(46, 18) = nzd(18, 10) = nzd(10, 8) = nzd(8, 2) = nzd(2, 0) = 2$.

Postupak se uvek zaustavlja jer je vrednost $a \bmod b$ uvek nenegativna (i celobrojna) i smanjuje se, pa mora nekada doći do nule. Korektnost algoritma može se dokazati indukcijom. Važi da $nzd(a, 0) = a$, jer je a najveći broj koji deli i a i 0. Pretpostavimo, kao induktivnu hipotezu, da je $n = nzd(b, a \bmod b)$ najveći broj koji deli broj b i broj $a \bmod b$. Dokažimo da je taj broj ujedno i $nzd(a, b)$. Važi da je $a = b \cdot q + a \bmod b$. Pošto n deli i b i $a \bmod b$, važi da n mora da deli i a, tako da je n sigurno delilac brojeva a i b. Dokažimo još i da je najveći. Ako neko n' deli a i b, tada na osnovu $a = b \cdot q + a \bmod b$ taj broj sigurno mora da deli i $a \bmod b$. Međutim, pošto je n najveći zajednički delilac brojeva b i $a \bmod b$, tada n' deli n. Dakle, svaki delilac brojeva a i b deli n, pa je $nzd(a, b) = n$.

Na osnovu navedenog, može se jednostavno implementirati naredna rekurzivna funkcija.

```
unsigned nzd(unsigned a, unsigned b) {
  if (b == 0)
    return a;
  else
    return nzd(b, a % b);
}
```

Algoritam se može ilustrovati i geometrijski. Pretpostavimo da je dat pravougaonik čije su dužine stranica a i b i da je potrebno odrediti najveću dužinu stranice kvadrata takva da se pravougaonik može popločati kvadratima te dimenzije. Ako je polazni pravougaonik dimenzije $a = 46$ i $b = 18$, tada se prvo iz njega mogu iseći dva kvadrata dimenzije 18 puta 18 i ostaće nam pravougaonik dimenzije 18 puta 10. Jasno je da ako nekim manjim kvadratima uspemo da popločamo taj preostali pravougaonik, da ćemo tim kvadratima uspeti da popločamo i ove kvadrate dimenzije 18 puta 18 (jer će dimenzija tih malih kvadrata deliti broj 18), pa ćemo samim tim moći popločati i ceo polazni pravougaonik dimenzija 46 puta 18. Od pravougaonika dimenzije 18 puta 10 možemo iseći kvadrat dimenzije 10 puta 10 i preostaće nam pravougaonik dimenzije 10 puta 8. Ponovo, kvadratići kojima će se moći popločati taj preostali pravougaonik će biti takvi da se njima može popločati i isečeni kvadrat dimenzije 10 puta 10. Od tog pravougaonika isecamo kvadrat 8 puta 8 i dobijamo pravougaonik dimenzije 8 puta 2. Njega možemo razložiti na četiri kvadrata dimenzije 2 puta 2 i to je najveća dimenzija kvadrata kojim se može popločati polazni pravougaonik.



Kao što je već pokazano u poglavlju 4.7, Euklidov algoritam može se opisati i iterativno. U petlji koja se izvršava sve dok je broj b veći od nule, par promenljivih (a, b) zamenjuje se vrednostima $(b, a \bmod b)$. Naivni pokušaj da se to uradi na sledeći način:

```
a = b;
b = a % b;
```

nije korektan, jer se prilikom izračunavanja ostatka koristi već izmenjena vrednost promenljive `a`. Zato je neophodno upotrebiti pomoćnu promenljivu, na primer:

```
tmp = b;
b = a % b;
a = tmp;
```

Na kraju petlje, vrednost b jednaka je nuli, pa se kao rezultat može prijaviti tekuća vrednost broja a. Dakle, iterativna implementacija može biti ovakva:

```
unsigned nzd(unsigned a, unsigned b) {
  while (b != 0) {
    unsigned tmp = b;
    b = a % b;
    a = tmp;
  }
  return a;
}
```

Originalna formulacija Euklidovog algoritma, umesto manjeg broja i ostatka pri deljenju većeg broja manjim, razmatra manji broj i razliku između većeg i manjeg broja. Ako su dva broja značajno različita po veličini, toj varijanti potreban je mnogo veći broj koraka, pa zato, kao neefikasna, ona ovde nije implementirana.

5.4.2 Izračunavanje vrednosti polinoma

Vrednost polinoma oblika

$$P(x) = a_0 + a_1 x + \ldots + a_{n-1} x^{n-1} + a_n x^n$$

za konkretnu vrednost x može se izračunati („grubom silom“) korišćenjem $n + (n-1) + \ldots + 1$ množenja i n sabiranja, ali se može izračunati i znatno efikasnije korišćenjem sledećeg *Hornerovog* zapisa istog polinoma:

$$P(x) = a_0 + (a_1 + \ldots + (a_{n-1} + (a_n x)) \ldots x)x$$

Naredna funkcija, zasnovana na navedenom zapisu, koristi samo n množenja i n sabiranja.

```
double vrednost_polinoma(double x, double a[], int n) {
  int i;
  double v=a[n];
  for (i = n-1; i >= 0; i--)
    v = x*v+a[i];
  return v;
}
```

Pitanja i zadaci za vežbu

Pitanje 5.35. *Koja je složenost Euklidovog algoritma za pronalaženje najvećeg zajedničkog delioca?*

Pitanje 5.36. *Modifikovati Euklidov algoritam tako da za date uzajamno proste brojeve a i b određuje celobrojne koeficijente k_a i k_b za koje važi $k_a \cdot a + k_b \cdot b = 1$.*

Pitanje 5.37. *Opisati algoritam kojim se može izračunati vrednost polinoma $P(x)$, ako se koeficijenti obrađuju:*
(a) *sleva nadesno.*
(b) *zdesna nalevo.*
Koja je složenost svakog od opisanih algoritama?

Pitanje 5.38. *Po uzoru na algoritme za sabiranje i množenje polinoma, opiši algoritme za sabiranje i množenje velikih brojeva (predstavljenih nizovima svojih cifara).*

Zadatak 5.30. *Definisati funkciju koja skraćuje razlomak tako da su u dobijenom razlomku brojilac i imenilac uzajamno prosti.*

Zadatak 5.31. *Od pravougaonika poznatih dimenzija se u svakom koraku iseca najveći mogući kvadrat i zatim se isti postupak primenjuje na preostali deo pravougaonika. Definisati funkciju koja određuje broj kvadrata koji će biti isečeni.*

Zadatak 5.32. *Svaki k-ti kupac dobija na poklon koka-kolu, svaki m-ti čokoladicu, a svaki n-ti dobija čips. Napiši program koji određuje redni broj prvog kupca koji će na poklon dobiti sva tri proizvoda.*

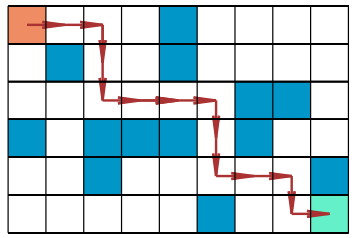
5.5 Pretraga

U računarstvu često postoji potreba da se obradi skup elemenata koji su na neki način međusobno povezani, tako da se zna koji su elementi susedni tj. tako da za svaki element zna na koji se element može preći. Takvi elementi i veze među njima mogu se reprezentovati eksplicitnom strukturom podataka koja se naziva *graf*. U rešavanju nekih problema, graf se ne čuva eksplicitno u memoriji računara, već je samo implicitan i u nastavku ćemo se baviti takvim primerima. Jedan čest primer takve obrade je obilazak matrice u kojoj se sa svakog polja može preći na susedna (najviše četiri ili osam, u zavisnosti od toga koji smerovi kretanja su dopušteni).

Dva najznačajnija algoritma obilaska su *obilazak u dubinu* i *obilazak u širinu*. Obilazak može podrazumevati samo da se nabroje sva polja do kojih se može stići iz nekog početnog. Obilazak se često primenjuje na problem pronalaženja nekog željenog polja, pa se ovi algoritmi nazivaju i *pretraga u dubinu* (engl. depth first search, DFS) i *pretraga u širinu* (engl. breadth first search, BFS). Pretraga u dubinu veoma se jednostavno implementira rekurzivno, mada je moguća i jednostavna nerekurzivna implementacija uz korišćenje steka (videti poglavlje 6.4.5). Pretraga u širinu ima svojstvo da se polja nabrajaju redom u odnosu na najmanji broj koraka koje je potrebno napraviti da bi se do njih stiglo sa početnog polja (pa se time do svakog polja pronalazi najkraći mogući put). Implementacija pretrage u širinu najčešće se vrši uz korišćenje reda kao pomoćne strukture podataka (videti poglavlje 6.5.6).

5.5.1 Ispitivanje da li postoji na mapi

Pretpostavimo da je mapa data u vidu matrice dimenzija $m \times n$ čiji su elementi nule i jedinice. Nule predstavljaju polja koja su dostupna, a jedinice prepreke. Potrebno je ispitati da li postoji put od gornjeg levog ugla do donjeg desnog ugla matrice. Dozvoljena su samo kretanja gore, dole, levo i desno (a ne i dijagonalna). Naredna slika ilustruje jedan put na takvoj mapi.



Pretpostavljamo da se matrica zadaje sa ulaza na sledeći način: najpre se zadaju dimenzije m i n, a zatim sadržaj polja u vidu nula i jedinica, pri čemu se očekuje prelazak u novi red nakon svake vrste.

U prikazanom rešenju, osnovna ideja je sledeća: put od polja A do polja B postoji, ako i samo ako postoji put od nekog polja koje je susedno polju A do polja B. Funkcija `postojiPut` kreće od polaznog polja (polja $(0, 0)$) i rekurzivno obilazi polja koja su gore, dole, levo i desno od njega – ukoliko dobije vrednost `1`, onda to znači da postoji put od tog susednog polja do ciljnog polja, pa može da se vrati rezultat `1`. Na samom početku funkcije proverava se da li je polje koje se trenutno ispituje ciljno polje i, ako jeste, vraća se vrednost 1. Nakon toga proverava se da li je polje koje se trenutno ispituje van table, a zatim i da li je već posećeno (što se pamti u nizu `poseceno`) ili je prepreka. U tim slučajevima, funkcija vraća vrednost `0` koja govori da ne postoji put od polja `(v,k)` do ciljnog polja. Primetimo da je niz `poseceno` deklarisan kao `static`, pa se alocira samo jednom.

```
#include <stdio.h>

#define MAX_DIM 50

int postojiPut(int prepreke[MAX_DIM][MAX_DIM], int m, int n,
               int v, int k) {
  static int poseceno[MAX_DIM][MAX_DIM] = {0};
  /* da li je (v,k) ciljno polje? */
  if (v == m-1 && k == n-1)
    return 1;
  /* da li je polje (v,k) van table? */
  if (!(0 <= v && v < m && 0 <= k && k < n)
    return 0;
  /* da li je polje (v,k) vec poseceno ili prepreka? */
  if (poseceno[v][k] || prepreke[v][k])
    return 0;
  poseceno[v][k] = 1;
  if (postojiPut(prepreke, m, n, v+1, k))
    return 1;
  if (postojiPut(prepreke, m, n, v-1, k))
    return 1;
  if (postojiPut(prepreke, m, n, v, k-1))
    return 1;
  if (postojiPut(prepreke, m, n, v, k+1))
    return 1;
  return 0;
}
```

```c
int main() {
  int prepreke[MAX_DIM][MAX_DIM];
  int m, n;
  char c;

  /* ucitavamo dimenzije matrice prepreka */
  if (scanf("%d %d\n", &m, &n) != 2)
    return -1;
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      c = getchar();
      prepreke[i][j] = (c == '1'); /* '1' oznacava prepreku */
    }
    c = getchar(); /* preskacemo '\n' */
  }
  /* trazi se put od polja (0,0) */
  if (postojiPut(prepreke, m, n, 0, 0))
    printf("\nPostoji put\n");
  else
    printf("\nNe postoji put\n");
  return 0;
}
```

Program za naredne ulazne podatke:

```
3 4
0000
0101
0100
```

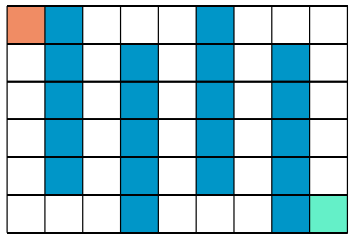
daje sledeći izlaz:

```
Postoji put
```

Ispravnost rešenja zasniva se na već navedenoj činjenici da put od polja A do polja B postoji ako i samo ako postoji put od nekog polja koje je susedno polju A do polja B. Pritom, niz `poseceno` koristimo na način koji nam obezbeđuje da ne vršimo dva puta proveru jednog istog polja (ali svako polje koje treba da bude obrađeno biva obrađeno bar jednom).

Navedena funkcija ima četiri rekurzivna poziva, te deluje da će lako doći do eksplozije broja polja koja se ispituju. Međutim, niz `poseceno` sprečava ponovnu obradu polja koja su obrađena, te je broj polja koja se obrađuju reda $O(mn)$ i tolika je i vremenska složenost funkcije `postojiPut`. Stek okvir za funkciju `postojiPut` je konstante veličine, ali se koristi statički niz `poseceno` čija je veličina `MAX_DIM` · `MAX_DIM`. Ako bi bila korišćena dinamička alokacija, prostor za niz `poseceno` bio bi reda $O(mn)$. Pitanje je još i koliko rekurzivnih

poziva može biti aktivno u jednom trenutku, tj. koliko najviše može biti stek okvira na programskom steku. Naredna slika ilustruje jedan tip situacija u kojma je broj stek okvira reda $\Theta(mn)$:



Ovaj problem može se rešiti i bez korišćenja rekurzije, ali zato uz korišćenje dodatnih struktura podataka, kao što su stek i red (videti poglavlja 6.4.5 i 6.5.6).

Algoritam se može jednostavno dopuniti tako da ako put postoji, onda će on biti i efektivno određen (i, na primer, ispisan).

Pitanja i zadaci za vežbu

Pitanje 5.39. *Koji algoritam je moguće primeniti da bi se pronašao najkraći put na mapi? Koju pomoćnu strukturu podataka taj algoritam koristi?*

Pitanje 5.40. *Ako je lavirint prikazan matricom dimenzije $m \times n$, koja je složenost algoritma koji pronalazi put u lavirintu primenom pretrage u dubinu?*

Zadatak 5.33. *Jedna avio-kompanija zajedno sa svojim partnerima izvodi letove između velikih svetskih aerodroma. Napiši program koji određuje da li je moguće da se korišćenjem tih letova stigne sa jednog na drugi dati aerodrom. Program sa ulaza unosi broj letova, i zatim letove (svaki let je opisan polaznim i dolaznim i aerodromom, a svaki aerodrom troslovnom jedinstvenom oznakom - npr. let od Beograda do Frankfurta je opisan pomoću niske* `BEG FRA`*). Na kraju se unose željeni polazni i dolazni aerodrom.*

Zadatak 5.34. *Napisati program kojim se određuje minimalni broj (P, Q)-konja kojima se mogu kontrolisati (obići) sva polja šahovske table dimenzija nxm.*

(P, Q)-konj je figura koja se u jednom skoku premeta za P polja po horizontali i Q polja po vertikali ili Q polja po horizontali i P polja po vertikali. Naprimer, $(1, 2)$-konj je običan šahovski konj.

Figura ili grupa figura kontrolišu polje ako mogu do njega stići u nula ili više skokova. Na primer, za kontrolu standardne šahovske table dimenzija $8x8$ potrebna su dva $(1, 1)$-konja (to su figure slične lovcima, ali se kreću za po jedno polje).

Zadatak 5.35. *Tabla se sastoji od polja zadatih matricom i svako polje je je obojeno jednom od 4 boje (crvena polja su obeležena brojem 1, plava brojem 2, žuta brojem 3 i zelena brojem 4). Pravila kretanja su takva da se mora poći sa crvenog polja, sa crvenog polja se može preći samo na plavo, sa plavog na žuto, sa žutog na zeleno i sa zelenog na crveno (mora se poći sa broja 1 i tokom kretanja brojevi moraju biti 1, 2, 3, 4, 1, 2, 3, 4, ...). Napiši program koji na osnovu učitane matrice boja određuje da li se od donjeg reda table može stići do gornjeg.*

5.6 Generisanje kombinatornih objekata

Prilikom rešavanja problema često je potrebno ispitati sve mogućnosti tj. generisati sve objekte koji zadovoljavaju određena svojstva. Nekad se ti objekti mogu predstaviti nizovima, a nekad objektima koji se razmatraju u kombinatorici: kombinacijama, varijacijama, permutacijama, particijama i slično.

5.6.1 Varijacije sa ponavljanjem

Varijacija sa ponavljanjem dužine k nad n elemenata je niz dužine k, pri čemu je svaki element niza jedan od n zadatih elemenata.[11] Za nabrajanje varijacija proizvoljnog skupa od n elemenata (koji ne moraju biti brojevne vrste) dovoljno je imati mehanizam za nabrajanje varijacija brojeva od 0 do $n-1$. Naime, svakom od n elemenata zadatog skupa različitih elemenata može se pridružiti indeks (jedan od brojeva od 0 do $n-1$), a onda se varijacija ovih brojeva može prevesti u varijaciju elemenata zadatog skupa. Varijacija sa ponavljanjem dužine k nad n elemenata ima n^k. Kao primer, u nastavku su nabrojane (u leksikografskom poretku) varijacije sa ponavljanjem dužine 2 nad 3 broja (0, 1 i 2):

```
0 0, 0 1, 0 2, 1 0, 1 1, 1 2, 2 0, 2 1, 2 2
```

Naredna rekurzivna funkcija ispisuje sve varijacije sa ponavljanjem dužine k brojeva između 0 i $n-1$.

```
void varijacije_(int a[], int n, int k, int i)
{
  if (i == k)
    ispisi(a, k);
  else {
    int j;
    for (j = 0; j < n; j++) {
```

[11] U varijacijama *bez* ponavljanja dužine k nad n elemenata nijedan element ne može da se pojavljuje više nego jednom.

```
      a[i] = j;
      varijacije_(a, n, k, i+1);
    }
  }
}
```

Varijacija se upisuje u niz `a` dužine `k`, koji se postepeno popunjava tokom rekurzivnih poziva. Parametar `i` označava narednu poziciju u nizu `a` koju treba popuniti. Kada taj broj dostigne dužinu `k`, jedna varijacija je generisana i ispisuje se.

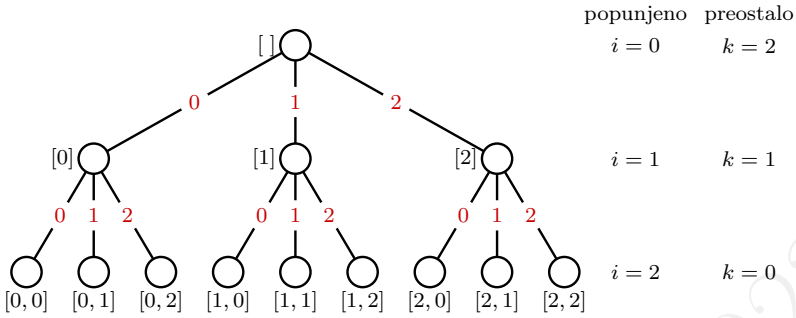
Kako bi se korisnik oslobodio potrebe za kreiranjem pomoćnog niza `a`, moguće je koristiti naredni omotač.

```
void varijacije(int n, int k)
{
  int *a = malloc(k * sizeof(int));
  if (a == NULL) {
    printf("Nedovoljno memorije.\n");
    return;
  }
  varijacije_(a, n, k, 0);
  free(a);
}
```

Stablo rekurzivnih poziva za $n = 3$ i $k = 2$ prikazano je na slici 5.4. Svaki rekurzivni poziv predstavljen je jednim čvorom stabla. Rekurzija kreće od korena, kada je `i` jednako 0 i kada je niz `a` suštinski prazan. U petlji se na njegovo prvo mesto upisuju redom elementi od 0 do $n - 1$ (to su brojevi 0, 1 i zatim 2) i za svaki tako dobijeni niz vrši se rekurzivni poziv, u kojem je uvećana pozicija `i` u nizu `a` na koju je potrebno upisati naredni element (to je ujedno i broj popunjenih elemenata u nizu `a`). U listovima stabla nalaze se sve varijacije dužine 2.

Iz navedenog primera, može se naslutiti da postoji neposredna veza između listova stabla rekurzivnih poziva i traženih varijacija, pa listova stabla ima tačno n^k (u datom primeru je $n = 3$, $k = 2$ i ima tačno $3^2 = 9$ listova stabla). Stablo je potpuno i svaki unutrašnji čvor (tj. svaki čvor koji nije list) ima n naslednika, što znači da više od polovine čvorova stabla čine upravo listovi (u potpunom binarnom stablu unutrašnjih čvorova ima za jedan manje nego listova, a u drugim potpunim stablima udeo unutrašnjih čvorova još je manji).[12] Pošto se u svakom čvoru (rekurzivnom pozivu) izvrši konstantan broj operacija, vremenska složenost asimptotski je jednaka broju listova, a to je u ovom primeru $O(n^k)$. Naravno, ovo je veoma gruba ocena i do istog rezultata može

[12] U ovom konkretnom slučaju lako je izračunati broj unutrašnjih čvorova i listova – prvih ima $1 + n + n^2 + \ldots + n^{k-1} = (n^k - 1)/(n - 1)$, a drugih n^k. Generalno, često je znatno lakše a i sasvim dovoljno samo grubo proceniti (ili ograničiti) neku veličinu.



Slika 5.4: Postupak generisanja varijacija sa ponavljanjem dužine $k = 2$, za $n = 3$.

se doći i preciznijom analizom. Složenost je eksponencijalna u odnosu na parametar k, te je navedena funkcija veoma neefikasna i praktično je upotrebljiva samo za male vrednosti parametra k (nekoliko desetina). Problem nije u lošoj implementaciji nego u ogromnom broju objekata koje je neophodno generisati i obraditi (bez obzira na izbor algoritma).

Tokom izvršavanja funkcije `varijacije` kreira se $k + 1$ stek okvira za instance funkcije `varijacije_`. Na hipu se rezerviše prostor reda $O(k)$, pa je ukupna prostorna složenost funkcije `varijacije`, dakle, reda $O(k)$.

Funkcija koja generiše varijacije od k elemenata skupa 0 do $n - 1$ lako se može prilagoditi tako da ispisuje varijacije elemenata bilo kog skupa. Jedan mogući pristup je da se u niz koji sadrži trenutnu varijaciju stavljaju odgovarajući elementi tog skupa. Na primer, naredna funkcija generiše i ispisuje sve varijacije datog skupa niski (tekuća varijacija pamti se u nizu `a`, a sam skup predstavljen je nizom `elementi` tipa `char*`).

```
void varijacije_(char* a[], int n, int k, int i,
                 char* elementi[])
{
  if (i == k)
    ispisi(a, k);
  else {
    int j;
    for (j = 0; j < n; j++) {
      a[i] = elementi[j];
      varijacije_(a, n, k, i+1, elementi);
    }
  }
}
```

Ipak, opštije rešenje je ono koje u prvoj fazi generiše varijacije samo nad sku-

pom indeksa tj. nad elementima skupa $0, 1, \ldots, n-1$, gde je n broj elemenata skupa čije se varijacije generišu. U drugoj fazi (prilikom ispisa traženih varijacija) ispisuju se elementi određeni tim indeksima. Na primer, ako je potrebno odrediti četvoročlane varijacije skupa predstavljenog nizom `jabuka`, `kruska`, `sljiva`, onda varijacija `2, 1, 0, 1` daje varijaciju `sljiva-kruska-jabuka-kruska`. Najjednostavniji način da se ovo postigne je da se zadrži funkcija koja generiše varijacije skupa $0, 1, \ldots, n-1$, koju smo ranije prikazali, a da se modifikuje samo funkcija koja ispisuje varijaciju (bolja implementacija ne bi trebalo da koristi globalne promenljive).

```
char* elementi[] = {"jabuka", "kruska", "sljiva"};

void ispisi(int a[], int k) {
  int i;
  for (i = 0; i < k; i++)
     printf("%s ", elementi[a[i]]);
  printf("\n");
}
```

Ako se traže dvočlane varijacije prethodni program daje sledeći izlaz.

```
jabuka jabuka
jabuka kruska
jabuka sljiva
kruska jabuka
kruska kruska
kruska sljiva
sljiva jabuka
sljiva kruska
sljiva sljiva
```

U nekim slučajevima, potrebno je za datu varijaciju sa ponavljanjem pronaći sledeću varijaciju u leksikografskom poretku. To može biti urađeno korišćenjem naredne funkcije:

```
int sledeca_varijacija(int a[], int n, int k)
{
  int i;
  for (i = k - 1; i >= 0 && a[i] == n-1; i--)
    a[i] = 0;
  if (i < 0)
    return 0;
  a[i]++;
  return 1;
}
```

Narednu varijaciju sa ponavljanjem u leksikografskom poretku navedena funkcija pronalazi tako što pronalazi prvi element zdesna koji nema maksimalnu vrednost (vrednost $n-1$), uveća ga za 1, a sve elemente iza njega postavlja na najmanju vrednost (vrednost 0). Na primer, ako je $n = 5$ i $k = 5$, naredna varijacija u odnosu na varijaciju 01244 je 01300, a naredna varijacija u odnosu na varijaciju 01300 je 01301. Ako postoji element koji se može uvećati, to znači da se uspešno prešlo na narednu varijaciju i funkcija vraća vrednost `1`, dok u suprotnom (na primer, za 44444), promenljiva `i` opada na vrednost `-1`, to znači da je tekuća varijacija poslednja i funkcija vraća vrednost `0`.

Prethodna funkcija može se iskoristiti i za nabrajanje svih varijacija sa ponavljanjem. Na primer:

```
void varijacije(int n, int k)
{
  int i;
  int *a = malloc(k*sizeof(int));
  if (a == NULL) {
    printf("Nedovoljno memorije.\n");
    return;
  }
  for(i = 0; i < k; i++)
    a[i] = 0;
  do {
    ispisi(a, k);
  } while (sledeca_varijacija(a, n, k));
}
```

Pošto i ova funkcija nabraja sve varijacije, njena vremenska složenost pripada klasi $\Omega(n^k)$, pa je i ona veoma neefikasna.

5.6.2 Kombinacije bez ponavljanja

Kombinacija bez ponavljanja dužine k nad n elemenata je jedan podskup veličine k skupa veličine n. Kombinacija dužine k nad n elemenata ima $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Za nabrajanje kombinacija skupa od n elemenata dovoljno je imati mehanizam za nabrajanje kombinacija brojeva od 0 do $n-1$ (različiti pristupi generisanja kombinacija proizvoljnog skupa analogni su pristupima za varijacije, poglavlje 5.6.1). Kao primer, u nastavku su nabrojane (u leksikografskom poretku) sve kombinacije dužine 3 nad 5 elemenata (od broja 0 do broja 4):

```
0 1 2, 0 1 3, 0 1 4, 0 2 3, 0 2 4, 0 3 4, 1 2 3, 1 2 4, 1 3 4, 2 3 4
```

Razmotrimo najpre sledeću rekurzivnu funkciju koja je ključna za ispisivanje svih kombinacija dužine k od elemenata $0, \ldots, n-1$.

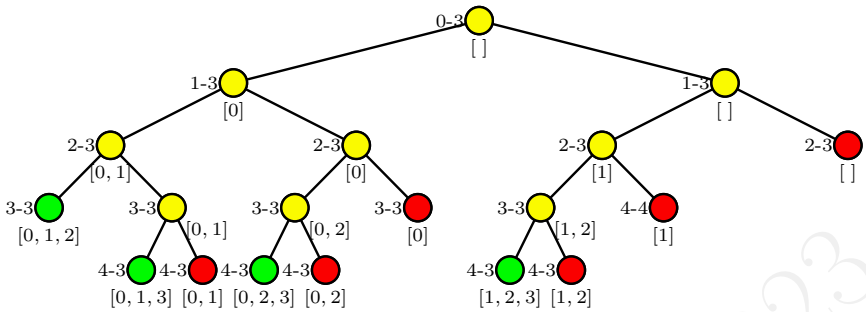
```
void kombinacije_(int a[], int n, int k, int i, int min)
{
  if (i == k)
    ispisi(a, k);
  else
    if (k - i <= n - min) {
      a[i] = min;
      kombinacije_(a, n, k, i+1, min+1);

      kombinacije_(a, n, k, i, min+1);
    }
}
```

Kombinacija (čiji su elementi sortirani) upisuje se u niz `a` dužine `k`, koji se postepeno popunjava tokom rekurzivnih poziva. Parametar `i` označava narednu poziciju u nizu `a` koju treba popuniti. Kada taj broj dostigne dužinu `k`, niz je popunjen i ispisuje se. Parametar `min` označava minimalnu vrednost koju je moguće upisati na preostale pozicije niza. Pošto je najveća moguća vrednost koju je moguće upisati `n-1`, u svakom trenutku su na raspolaganju vrednosti iz intervala `[min, n-1]` (njih `(n-1) - min + 1` = `n - min`) i njima treba popuniti preostale pozicije u nizu. Ako je tih vrednosti manje od broja preostalih pozicija `k - i`, onda je popunjavanje preostalog dela niza nemoguće i ta grana pretrage se prekida. U suprotnom, vrši se proširivanje niza na dva moguća načina. U prvoj grani na tekuću poziciju postavlja se vrednost `min`, dok se u drugoj grani vrednost `min` izostavlja iz tekuće kombinacije (u oba rekurzivna poziva `min` se uvećava za 1, pri čemu se u prvom pozivu prelazi na sledeću poziciju `i + 1`, a u drugom se to ne radi, pa i dalje treba popuniti pozicija `i`).

Kako bi se korisnik oslobodio potrebe za kreiranjem pomoćnog niza `a`, moguće je koristiti naredni omotač (popunjavanje kreće od pozicije `i = 0`, a minimum od `min = 0`).

```
void kombinacije(int n, int k)
{
  int *a = malloc(k*sizeof(int));
  if (a == NULL) {
    printf("Nedovoljno memorije.\n");
    return;
  }
  kombinacije_(a, n, k, 0, 0);
  free(a);
}
```

Rad navedenih funkcija za parametre $n = 4$ i $k = 3$ ilustrovan je na slici 5.5.



Slika 5.5: Generisanje kombinacija za $n = 4$ i $k = 3$: levo od svakog čvora prikazan je raspon preostalih vrednosti, a ispod čvora prikazana je tekuća kombinacija. U zelenim čvorovima su uspešno generisane kombinacije, a u crvenim nastupa odsecanje, jer raspon ne sadrži dovoljno vrednosti da bi se kombinacija generisala do kraja.

Pošto se svaka kombinacija, od njih ukupno $\binom{n}{k}$, ispisuje bar jednom, vremenska složenost pripada klasi $\Omega(\binom{n}{k})$. Može se pokazati i da vremenska složenost ujedno pripada i klasi $O(\binom{n}{k})$, pa onda i klasi $\Theta(\binom{n}{k})$. Vrednost $\binom{n}{k}$ raste jako brzo (može se dokazati da je odozdo ograničen sa $(\frac{n}{k})^k$, a odozgo sa $(\frac{n \cdot e}{k})^k$). Stoga je funkcija `kombinacije` praktično upotrebljiva samo za male vrednosti n i k (kada je n nekoliko desetina, osim u slučaju jako malih vrednosti k ili vrednosti k bliskih vrednosti n). Može se dokazati da je dodatna prostorna složenost navedene funkcije reda $O(k)$.

Ukoliko je data neka kombinacija, naredna funkcija pronalazi sledeću kombinaciju u leksikografskom poretku.

```
int sledeca_kombinacija(int a[], int n, int k)
{
  int i = k-1;
  while (i >= 0 && a[i] == n-k+i)
    i--;

  if (i < 0)
    return 0;

  a[i]++;
  for (i = i+1; i < k; i++)
    a[i] = a[i-1] + 1;
  return 1;
}
```

Prethodna funkcija se može iskoristiti i za nabrajanje svih kombinacija. Na primer,

```
void kombinacije(int n, int k)
{
  int i;
  int *a = malloc(k*sizeof(int));
  if (a == NULL) {
    printf("Nedovoljno memorije.\n");
    return;
  }
  for(i = 0; i < k; i++)
    a[i] = i;
  do {
    ispisi(a, k);
  } while (sledeca_kombinacija(a, n, k));
}
```

5.6.3 Permutacije

Permutacija je jedan poredak zadatog skupa elemenata. Ako zadatih elemenata ima n, onda permutacija nad tim elementima ima $n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$. Slično kao za varijacije i kombinacije, za nabrajanje permutacija proizvoljnog skupa od n elemenata (koji ne moraju biti brojevne vrste) dovoljno je imati mehanizam za nabrajanje permutacija brojeva od 0 do $n-1$. Naime, svakom od n elemenata može se pridružiti jedan od brojeva od 0 do $n-1$, a onda se permutacija ovih brojeva može prevesti u permutaciju zadatih elemenata. U nastavku su, kao primer, nabrojane (u leksikografskom poretku) sve permutacije brojeva od 0 do 2:

```
0 1 2, 0 2 1, 1 0 2, 1 2 0, 2 0 1, 2 1 0
```

Postoji više različitih algoritama za nabrajanje svih permutacija. U nastavku su data dva algoritma za nabrajanje svih permutacija brojeva od 0 do $n-1$. Prvi od njih odgovara generisanju varijacija bez ponavljanja, gde se na svaku od n pozicija mogu postaviti elementi od 0 do $n-1$.

```
void permutacije_(int a[], int n, int i, int upotrebljen[])
{
  if (i == n)
    ispisi(a, n);
  else {
    int j;
    for (j = 0; j < n; j++) {
```

```
      if (!upotrebljen[j]) {
        a[i] = j;
        upotrebljen[j] = 1;
        permutacije_(a, n, i+1, upotrebljen);
        upotrebljen[j] = 0;
      }
    }
  }
}
```

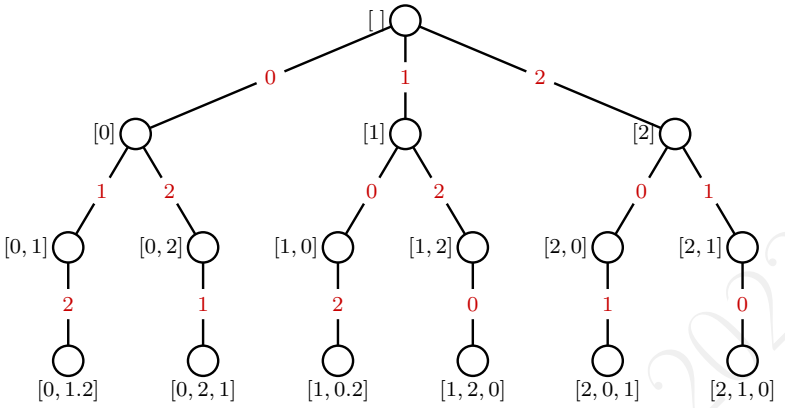
Navedena funkcija redom popunjava elemente niza `a` dužine `n`. Promenljiva `i` čuva indeks naredne pozicije u nizu `a` koju treba popuniti. Kada njena vrednost dostigne `n`, ceo niz je popunjen i tada se ispisuje generisana permutacija. Na poziciju `i` treba postavljati jedan po jedan element skupa od `0` do `n-1` koji nije već postavljen na neku od prethodnih pozicija u nizu. Pretraga tog dela niza zahteva linearno vreme i da bi se to ne bi radilo, tj. da bi se ubrzala procedura održava se i logički niz `upotrebljen` koji čuva informacije o tome koji elementi već postoje na prethodnim pozicijima: u nizu `upotrebljen`, na poziciji `k` je vrednost `1` ako i samo ako je element `k` već postavljen u deo niza `a` pre pozicije `i` (tako se u konstantnom vremenu može proveriti da li je neki element već uključen u permutaciju).

Prolazi se redom kroz sve kandidate (vrednosti od `0` do `n-1`), onaj koji nije već uključen u permutaciju postavlja se na mesto `i`, beleži se da je taj element upotrebljen i nastavlja se rekurzivno generisanje permutacije. Nakon toga, element se povlači tako što se zabeleži da više nije upotrebljen i prelazi se na narednog kandidata.

Naredna funkcija predstavlja omotač koji oslobađa korisnika potrebe za alociranjem pomoćnog niza. Na početku nijedan element nije upotrebljen (funkcijom `calloc` inicijalizuje se niz na nulu) i generisanje permutacije počinje od pozicije `0`.

```
void permutacije(int n)
{
  int *upotrebljen = calloc(n, sizeof(int));
  int *a = malloc(n*sizeof(int));
  if (upotrebljen == NULL || a == NULL) {
     printf("Nedovoljno memorije.\n");
     return;
  }
  permutacije_(a, n, 0, upotrebljen);
  free(upotrebljen); free(a);
}
```

Pošto se svaka od $n!$ permutacija ispisuje bar jednom, jasno je da je složenost ove funkcije $\Omega(n!)$, što je već izuzetno neefikasno i funkcija je praktično upotrebljiva samo za male vrednosti n (već za $n = 15$, vrednost $15!$ je oko 10^{12},



Slika 5.6: Generisanje permutacija skupa $\{0, 1, 2\}$ – na tekuću poziciju postavlja se jedan po jedan element skupa, a koji nije već postavljen na prethodne pozicije (ti elementi su predstavljeni crvenom bojom)

pa je i za samo ispisivanje svih permutacija 15 elemenata i najsavremenijim računarima potrebno jako mnogo vremena). Može se dokazati da je vremenska složenost algoritma $O(n \cdot n!)$.

U stablu rekurzivnih poziva permutacije se nalaze u listovima i stablo ima $n!$ listova. Procenimo ukupan broj čvorova stabla (i listova i unutrašnjih čvorova). Postoji jedan koren, koji se može proširiti na n načina, pa na narednom nivou ima n čvorova. Svaki od njih se proširuje na $n-1$ načina, pa na narednom nivou ima $n(n-1)$ čvorova, na narednom $n(n-1)(n-2)$ itd., sve do poslednjeg nivoa na kojem ima $n!$ listova. Dakle, na nivou i ima $n(n-1)\ldots(n-i+1) = n!/(n-i)!$ čvorova, pa je ukupan broj čvorova jednak

$$n! \cdot \left(\frac{1}{n!} + \frac{1}{(n-1)!} + \ldots + \frac{1}{1!} + \frac{1}{0!} \right).$$

Pošto suma u zagradama teži konstanti e kada n teži beskonačnosti, ukupan broj čvorova (tj. rekurzivnih poziva) pripada klasi $O(n!)$. I eksperimentalno se može pokazati da se količnik broja rekurzivnih poziva i broja $n!$, kada n raste približava konstantnoj vrednosti $e \approx 2,72$. Može se dokazati, pošto se prilikom svakog rekurzivnog poziva izvrši $O(n)$ operacija (ne računajući vreme utrošeno u daljim rekurzivnim pozivima), da je ukupna vremenska složenost algoritma reda $O(n \cdot n!)$. Prostorna složenost algoritma je reda $O(n)$. Prethodna analiza se, naravno, može izvršiti i preciznije, korišćenjem rekurentih jednačina.

Tokom izvršavanja funkcije `permutacije` kreira se $n + 1$ stek okvira za instance funkcije `permutacije_`. Na hipu se rezerviše prostor reda $O(n)$, pa je ukupna prostorna složenost funkcije `permutacije`, dakle, reda $O(n + n) = O(n)$.

Navedeno rešenje oslanja se na to da se permutuje baš skup $\{0, 1, \ldots, n-1\}$. Ako je potrebno generisati permutacije proizvojnog skupa, jedno rešenje je, kao što je rečeno, iskoristiti algoritam za permutovanje brojeva 0 do $n-1$: nakon generisanja svake permutacije, ona se koristi kao niz indeksa za polazni niz. Drugo rešenje (slično kao u slučaju varijacija, poglavlje 5.6.1) je da niz `a` sadrži elemente zadatog skupa (dakle, da ima odgovarajući tip). Tada informacija o već upotrebljenim elemenatima ne bi mogla da bude predstavljena običnim nizom logičkih vrednosti, već nekom naprednijom strukturom podataka.

Alternativa navedenom rešenju zasnovana je na razmenama (u ovoj varijanti permutacije se ne generišu u leksikografskom poretku):

```
void permutacije_(int a[], int n, int i)
{
  if (i == n)
    ispisi(a, n);
  else {
    int j;
    for (j = i; j < n; j++) {
      razmeni(a, i, j);
      permutacije_(a, n, i+1);
      razmeni(a, i, j);
    }
  }
}
```

Na početku, niz `a` dužine `n` treba da sadrži identičku permutaciju (na primer, permutaciju 0 1 2 $\ldots n-1$). Zadatak pomoćne funkcije je da pronađe sve moguće permutacije elemenata niza `a` na pozicijama od `i` nadalje, ne menjajući pri tom elemente niza `a` na pozicijama manjim od `i`. Kada `i` dostigne vrednost `n`, niz je popunjen i ispisuje se. U suprotnom, na poziciju `i` dovodi se jedan po jedan element dela niza na pozicijama od `i` do `n-1`, tako što se vrši razmena elementa na poziciji `i` sa elementima na pozicijama `j` za svako `j` od `i` do `n-1` i zatim se rekurzivno poziva funkcija da odredi sve moguće rasporede počevši od naredne pozicije. Važna invarijanta funkcije je da nakon rekurzivnog poziva deo niza na pozicijama od `i` do `n-1` ostaje nepromenjen (nakon svakog rekurzivnog poziva niz ceo niz se vraća u stanje pre rekurzivnog poziva). Da bi se to obezbedilo, nakon rekurzivnog poziva sa parametrom `i+1`, mora da se vrati element sa pozicije `i` na poziciju `j`, pre nego što se pređe na element na poziciji `i+1`.

Glavna funkcija koja predstavlja omotač ponovo se implementira jednostavno (alocira pomoćni niz i generiše početnu permutaciju).

```
void permutacije(int n)
{
```

```
  int i;
  int *a = malloc(n * sizeof(int));
  if (a == NULL) {
     printf("Nedovoljno memorije.\n");
     return;
  }
  for (i = 0; i < n; i++)
    a[i] = i;
  permutacije_(a, n, 0);
  free(a);
}
```

U nekim situacijama potrebno je za datu permutaciju pronaći sledeću permutaciju u leksikografskom poretku. Razmotrimo permutaciju 02431. Zamenom elementa 1 i 3 dobija se permutacija 02413, koja je leksikografski manja od polazne. Slično bi se desilo i da se zamene elementi 4 i 3. Činjenica da je podniz 431 strogo opadajući govori da nije moguće ni na koji način razmeniti neka od ta tri elementa tako da se dobije leksikografski veća permutacija, tj. govori da je 02431 najveća permutacija koja počinje sa 02. Dakle, naredna permutacija je leksikografski najmanja permutacija koja počinje sa 03, a to je 03124. Zato se, u prvom koraku narednog algoritma, pronalazi prva pozicija i zdesna takva da je $a_i < a_{i+1}$ (za sve k takve da je $i+1 \leq k < n-1$ važi da je $a_k > a_{k+1}$):

```
int sledeca_permutacija(int a[], int n)
{
  int i, j;

  for (i = n - 2; a[i] > a[i+1]; i--)
    if (i == 0)
      return 0;

  for (j = n - 1; a[i] > a[j]; j--)
    ;

  razmeni(a, i, j);
  obrni(a, i+1, n-1);
  return 1;
}
```

Pronalaženje prve pozicije `i` zdesna takve da je `a_i < a_{i+1}` sprovodi se linearnom pretragom. U datom primeru je `i = 1` i `a[i] = 2`. Ako takva pozicija ne postoji, tekuća permutacija je opadajuća i samim tim leksikografski najveća (funkcija tada vraća `0`). Nakon toga, pronalazi se prva pozicija `j` zdesna takva da je `a[i] < a[j]` (ponovo linearnom pretragom) i razmenjuju

se elementi na pozicijama `i` i `j`. U razmatranom primeru je `j = 3` i `a[j] = 3`. i nakon razmene dobija se permutacija 03421. Pošto je ovom razmenom rep iza pozicije i i dalje sigurno opadajući, da bi se dobila željena leksikografski najmanja permutacija koja počinje sa 14, potrebno je obrnuti redosled njegovih elemenata. Obrtanje dela niza između pozicija `i` i `j` (uključujući i njih) vrši se funkcijom `obrni`:

```
void obrni(int a[], int i, int j)
{
  int i1, j1;
  for (i1 = i, j1 = j; i1 < j1; i1++, j1--)
    razmeni(a, i1, j1);
}
```

Nakon obrtanja, funkcija vraća `1` što označava da je pronađena naredna permutacija.

Funkcija koja pronalazi sledeću permutaciju može biti upotrebljena i za generisanje svih permutacija. Na primer,

```
void permutacije(int n) {
  int i;
  int *a = malloc(n*sizeof(int));
  if (a == NULL) {
    printf("Nedovoljno memorije.\n");
    return;
  }
  for (i = 0; i < n; i++)
    a[i] = i;
  do {
    ispisi(a, n);
  } while (sledeca_permutacija(a, n));
}
```

Problem generisanja nasumične permutacije takođe je interesantan i ima česte primene, ali se suštinski razlikuje od problema nabrajanja svih permutacija. Najpoznatiji algoritam za generisanje nasumične permutacije je Fišer-Jejtsov algoritam, takođe poznat pod nazivom Knutov algoritam. U prvom koraku algoritma početni element u nizu razmenjujemo sa nasumično odabranim elementom iz celog niza (uključujući i taj početni element). Nakon toga, na narednu poziciju postavljamo nasumično odabran element iz ostatka niza (ne razmatrajući više početni element, tj. razmatrajući sve pozicije osim početne). Postupak ponavljamo i za sve naredne pozicije, dok ne stignemo do kraja niza.

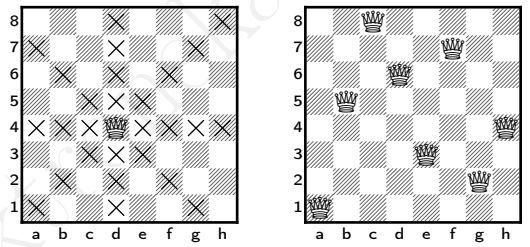
```
void promesaj(int a[], int n) {
  int i;
```

```
  for (int i = 0; i < n; i++)
    razmeni(a, i, random_int(i, n-1));
}
```

Funkcija `random_int(a, b)` generiše nasumični element celobrojnog intervala $[a, b]$. Lako je pokazati da su verovatnoće izbora svake od $n!$ permutacija jednake.

5.6.4 Problem osam dama – pretraga sa odsecanjem

Često je potrebno generisati sve kombinatorne objekte neke klase koji zadovoljavaju neko dodatno ograničenje (ili barem odrediti njihov broj). Jedan od mogućih pristupa, „pristup grube sile“ podrazumeva generisanje svih takvih objekata i, za svaki od njih pojedinačno, proveravanje da li je to dodatno ograničenje zadovoljeno. Ilustrujmo taj pristup na jednom klasičnom problemu, *problemu osam dama*, u kojem se zahteva da se osam dama (kraljica) rasporedi na šahovskoj tabli tako da nijedna ne napada neku drugu (dame se napadaju horizontalno, vertikalno i dijagonalno, kako je prikazano na slici 5.7, levo). Na slici 5.7, desno, prikazano je jedno od 92 rešenja koliko ih problem ima. Problemom se (bez upotrebe računara) bavio još znameniti matematičar Gaus, a uz primenu računara do rešenja se dolazi prilično jednostavno.



Slika 5.7: Problem osam dama: kretanje dame u šahu (levo) i jedno rešenje problema (desno)

Prvo pitanje koje se postavlja je kako reprezentovati problem. Jedna mogućnost je da se položaj dama predstavi matricom dimenzija 8×8 koja sadrži istinitosne vrednosti (recimo 0 na onim poljima na kojima nije dama i 1 na onim poljima na kojima jeste dama). Međutim, može se zaključiti da se u svakoj vrsti table mora naći tačno jedna dama, pa se svako rešenje može jednostavno predstaviti nizom brojeva (od 1 do 8, ili od 0 do 7) koji predstavljaju redne brojeve kolona u kojima se nalaze dame (po jedan broj za svaku vrstu, redom). Na primer, raspored na slici 5.7 može se predstaviti nizom $0, 6, 4, 7, 1, 3, 5, 2$. Dakle, svaki potencijani raspored dama predstavlja jednu varijaciju dužine osam niza brojeva od 0 do 7.

Provera da li se dame na poljima (k_1, v_1) i (k_2, v_2) napadaju može se izvršiti primenom aritmetičkih operacija. Naime, dve dame su u istoj koloni ako i samo ako je $k_1 = k_2$, a u istoj su vrsti ako i samo ako je $v_1 = v_2$. Dve dame se napadaju dijagonalno ako i samo ako je jednakokrak trougao čija su temena polja dve dame i polje koje se dobije projektovanjem polja jedne dame na vrstu u kojoj se nalazi druga dama. Dužine kateta tog jednakokrakog trougla određene su apsolutnim vrednostima razlike između rednih brojeva vrsta i između rednih brojeva kolona polja na kojih se dame nalaze. Dakle, provera da li se dve dame napadaju može se izvršiti pozivanjem naredne tri funkcije.

```
int ista_vrsta(int v1, int k1, int v2, int k2)
{
  return v1 == v2;
}

int ista_kolona(int v1, int k1, int v2, int k2)
{
  return k1 == k2;
}

int ista_dijagonala(int v1, int k1, int v2, int k2)
{
   return abs(v2 - v1) == abs(k2 - k1);
}
```

Provera da li niz kolona u kojima se dame nalaze predstavlja rešenje može se izvršiti na sledeći način (vrednost `8` zamenjena je simboličkim imenom `DIM`, pa se program može lako modifikovati da radi i za druge dimenzije table):

```
#define DIM 8
/* ... */

int jeste_resenje(int kolone_dama[])
{
  int v1, v2;
  for (v1 = 0; v1 < DIM; v1++)
    for (v2 = v1+1; v2 < DIM; v2++)
      if (ista_vrsta(v1, kolone_dama[v1],
                     v2, kolone_dama[v2]) ||
          ista_kolona(v1, kolone_dama[v1],
                      v2, kolone_dama[v2]) ||
          ista_dijagonala(v1, kolone_dama[v1],
                          v2, kolone_dama[v2]))
        return 0;
```

```
  return 1;
}
```

Kako je izabranom reprezentacijom osigurano da nema dve dame u istoj vrsti, nema potrebe pozivati funkciju `ista_vrsta`. Pozivi preostale dve funkcije mogu se zameniti njihovim (kratkim) telima, pa se kôd može malo uprostiti.

```
int jeste_resenje(int kolone_dama[])
{
  int v1, v2;
  for (v1 = 0; v1 < DIM; v1++)
    for (v2 = v1+1; v2 < DIM; v2++)
      if (kolone_dama[v1] == kolone_dama[v2] ||
          abs(v1-v2) == abs(kolone_dama[v1]-kolone_dama[v2]))
        return 0;
  return 1;
}
```

Ostaje pitanje kako nabrojati sve moguće nizove koji reprezentuju položaj dama. Jedno, prilično naivno rešenje je da se za to upotrebi 8 ugnežđenih petlji (za svaku vrstu po jedna). Umesto toga može se upotrebiti sledeće rekurzivno rešenje.

```
void ispisi(int kolone_dama[])
{
  static int i;
  printf("%i: ",i++);
  for (int j = 0; j < DIM; j++)
    printf("%i",kolone_dama[j]);
  printf("\n");
}

void dame(int kolone_dama[], int broj_postavljenih_dama)
{
  if (broj_postavljenih_dama == DIM) {
    if (jeste_resenje(kolone_dama))
      ispisi(kolone_dama);
  }
  else {
    int j;
    for (j = 0; j < DIM; j++) {
      kolone_dama[broj_postavljenih_dama] = j;
      dame(kolone_dama, broj_postavljenih_dama + 1);
    }
  }
```

```
}
```

Primetimo da navedena funkcija generiše sve varijacije sa ponavljanjem dužine `DIM` nad `DIM` elemenata, slično kao što je opisano u delu 5.6.1. Tih varijacija ima $8^8 = 16777216$. Iako navedeni kôd ispravno radi, može se značajno unaprediti. Pošto se ni u jednoj koloni ne mogu nalaziti dve dame, potrebno je da su svi brojevi različiti. Dakle, svaki potencijani raspored dama predstavlja jednu moguću permutaciju niza brojeva od 0 do 7 (s druge strane, ne zadovoljava svaka permutacija uslov da se nikoje dve dame ne napadaju). Imajući to u vidu, malko unapređen pristup grubom silom koristio bi generisanje svih permutacija (njih $8! = 40320$) i za svaku od njih proveru da li zadovoljaju uslov o nenapadanju dama, tj. da li predstavlja ispravno rešenje. I generisanje permutacija može se implementirati rekurzivno (kao što je opisano u delu 5.6.3). U ovom pristupu, provera da li se u nekoj koloni nalaze dve dame prebacuje se iz faze konačne provere u fazu generisanja potencijalnih kandidata za rešenje (u vidu uslova `!popunjena_kolona[j]`). To je primer *ranog odsecanja* tokom pretrage koje može dosta doprineti njenoj efikasnosti. Algoritmi koji odsecaju grane stabla pretrage za koje se rano može utvrditi da se u njima ne nalazi rešenje nazivaju se *bektreking* (engl. backtracking) algoritmi (što se ponekad prevodi kao *pretraga sa povratkom* ili *pretraga sa odsecanjem*).

```
int jeste_resenje(int kolone_dama[])
{
  int v1, v2;
  for (v1 = 0; v1 < DIM; v1++)
    for (v2 = v1+1; v2 < DIM; v2++)
      if (abs(v1-v2) == abs(kolone_dama[v1]-kolone_dama[v2]))
        return 0;
  return 1;
}

void dame(int kolone_dama[], int popunjena_kolona[],
          int broj_postavljenih_dama)
{
  if (broj_postavljenih_dama == DIM) {
    if (jeste_resenje(kolone_dama))
      ispisi(kolone_dama);
  }
  else {
    int j;
    for (j = 0; j < DIM; j++) {
      if (!popunjena_kolona[j]) {
        kolone_dama[broj_postavljenih_dama] = j;
        popunjena_kolona[j] = 1;
```

```
        dame(kolone_dama, popunjena_kolona,
             broj_postavljenih_dama + 1);
        popunjena_kolona[j] = 0;
      }
    }
  }
}
```

Uz navedeni kôd, pozivom naredne funkcije `main`:

```
int main()
{
  int kolone_dama[DIM], popunjena_kolona[DIM];
  for (int j = 0; j < DIM; j++)
    popunjena_kolona[j] = 0;
  dame(kolone_dama, popunjena_kolona, 0);
}
```

dobija se 92 rešenja problema:

```
0: 04752613
1: 05726314
2: 06357142
...
91: 73025164
```

U unapređivanju navedenog algoritma, može se otići još jedan korak dalje. Naime, često se do efikasnije pretrage dolazi ako se odsecanje vrši što ranije. Primetimo da se u prethodnom rešenju provera dijagonala vrši tek na kraju, kada su sve dame postavljene. Mnogo je, međutim, bolje odsecanje vršiti čim se na tablu postavi nova dama koja se napada sa nekom od postojećih dama. Tako se osigurava da će u svakom koraku pretrage dame na tabli zadovoljavati dati uslov nenapadanja i stoga više nije potrebno vršiti nikakve provere na samom kraju, onda kada su sve dame postavljene.

```
int dama_se_moze_postaviti(int kolone_dama[],
                           int broj_postavljenih_dama,
                           int v, int k)
{
  int v1;
  for (v1 = 0; v1 < broj_postavljenih_dama; v1++)
    if (ista_dijagonala(v1, kolone_dama[v1], v, k))
      return 0;
  return 1;
}
```

```
void dame(int kolone_dama[], int popunjena_kolona[],
          int broj_postavljenih_dama)
{
  if (broj_postavljenih_dama == DIM)
    ispisi(kolone_dama);
  else {
    int j;
    for (j = 0; j < DIM; j++) {
      if (!popunjena_kolona[j] &&
          dama_se_moze_postaviti(kolone_dama,
                                 broj_postavljenih_dama,
                                 broj_postavljenih_dama, j)) {
          kolone_dama[broj_postavljenih_dama] = j;
          popunjena_kolona[j] = 1;
          dame(kolone_dama, popunjena_kolona,
               broj_postavljenih_dama + 1);
          popunjena_kolona[j] = 0;
      }
    }
  }
}
```

5.6.5 Particionisanje broja

Particionisanje pozitivnog celog broja n je niz pozitivnih celih brojeva čiji je zbir n. Na primer, za broj $n = 4$ sva njegova particionisanja su:

```
4
3 1
2 2
2 1 1
1 3
1 2 1
1 1 2
1 1 1 1
```

Sledeća funkcija lista sva particionisanja zadatog broja n.

```
void particionisanja_(int a[], int n, int i)
{
  int j;
  if (n == 0)
    ispisi(a, i);
```

```
  else {
    for(j=n; j>0; j--) {
      a[i]=j;
      particionisanja_(a, n-j, i+1);
    }
  }
}
```

Naredni omotač oslobađa korisnika potrebe za obezbeđivanjem prostora za pomoćni niz a.

```
void particionisanja(int n)
{
  int *a = malloc(n * sizeof(int));
  if (a == NULL) {
    printf("Nedovoljno memorije.\n");
    return;
  }
  particionisanja_(a, n, 0);
  free(a);
}
```

Postavlja se pitanje koliko ima particionisanja broja n. Ako broj n predstavimo nizom od n cifara 1, onda svakom particionisanju odgovara jedan taj niz sa nekim umetnutim „pregradama“ između cifara, na primer:

$$1111|11|111|1$$

Ako postoji n cifara 1, onda se „pregrade“ mogu naći na ukupno $n-1$ pozicija. Na svakoj od tih pozicija, „pregrada“ može da postoji ili ne postoji, te ima ukupno 2^{n-1} načina da se postave „pregrade“ i svaki taj način daje po jedno ispravno particionisanje broja n. Štaviše, ako se razmatraju binarne reprezentacije brojeva od 0 do 2^{n-1}, svaka neposredno daje po jedno traženo particionisanje: tamo gde je cifra 1, tu treba da je „pregrada“, tamo gde je cifra 0, tu ne treba da je „pregrada“. Na primer, particionisanju 1111|11|111|1 odgovara raspored „pregrada“ 000101001, tj. broj 41. Dakle, svakom particionisanju broja n odgovara jedan broj između 0 i 2^{n-1} a i obratno. Na osnovu ovih zapažanja može se napraviti novo rešenje početnog problema: potrebno je u petlji obraditi sve brojeve od 0 do 2^{n-1} i za svaki, na osnovu njegove binarne reprezentacije, kreirati po jedno particionisanje polaznog broja n. Obrada binarne reprezentacije zahteva čitanje pojedinačnih bitova broja. O operatorima koji to omogućavaju govori naredni deo knjige.

Pitanja i zadaci za vežbu

Pitanje 5.41. *Koja je leksikografski sledeća varijacija sa ponavljanjem skupa* $\{1, 2, 3\}$ *dužine 4 u odnosu na varijaciju* 2313*?*

Pitanje 5.42. *Pseudokodom opisati rekurzivni algoritam koji generiše sve kombinacije bez ponavljanja skupa* $\{1, 2, \ldots, n\}$ *dužine k takav da se u svakom rekurzivnom pozivu obrađuje jedna pozicija u nizu koji predstavlja kombinaciju, na nju se stavlja svaki element koji može doći na tu poziciju i nakon toga se naredne pozicije popunjavaju u novom rekurzivnom pozivu.*

Pitanje 5.43. *Koja je leksikografski sledeća permutacija u odnosu na permutaciju 41532?*

Pitanje 5.44. *Navesti sve particije broja 4.*

Zadatak 5.36. *Napisati program koji ispisuje sve varijacije nula i jedinica dužine n koje ne sadrže dve uzastopne nule.*

Zadatak 5.37. *Definisati rekurzivnu funkciju koja ispisuje sve kombinacije sa ponavljanjem skupa* $\{1, 2, \ldots, n\}$ *koje sadrže k elemenata. Brojevi u svakoj kombinaciji treba da budu sortirani, a kombinacije ispisati u leksikografskom redosledu. Na primer, ako je* $n = 3$ *i* $k = 2$*, kombinacije su (1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3).*

Zadatak 5.38. *Definisati rekurzivnu funkciju kojom se ispisuju sve uređene (sortirane) particije broja.*

Zadatak 5.39. *Definisati funkciju koja ispisuje sve podskupove skupa* $\{1, 2, \ldots, n\}$*. Podskupovi treba da budu leksikografski sortirani (npr. za* $n = 3$*, program treba da ispiše* $\{\}, \{1\}, \{1, 2\}, \{1, 2, 3\}, \{1, 3\}, \{2\}, \{2, 3\}, \{3\}$*.*

Zadatak 5.40. *Definisati funkciju koja ispisuje sve n-tocifrene brojeve čiji je zbir cifara jednak datom broju z.*

Zadatak 5.41. *Napisati program koji ispisuje sve palindrome koji se mogu dobiti razmeštanjem slova date reči.*

Zadatak 5.42. *Particija je cik-cak ako susedni sabirci naizmenično rastu i opadaju (na primer,* $1+5+2+4$ *je jedna cik-cak particija broja 12). Napisati program koji ispisuje sve cik-cak particije broja n.*

5.7 Algoritmi zasnovani na bitovskim operatorima

U nekim slučajevima, efikasnija rešenja mogu se dobiti korišćenjem pojedinačnih bitova u zapisu celobrojnih izraza. C podržava naredne bitovske operatore (moguće ih je primenjivati samo na celobrojne argumente):

`&` – **bitovsko *i*** — primenom ovog operatora vrši se konjunkcija pojedinačnih bitova dva navedena argumenta (i-ti bit rezultata predstavlja konjunkciju i-tih bitova argumenata). Na primer, ukoliko su promenljive `x1` i `x2` tipa `unsigned char` i ukoliko je vrednost promenljive `x1` jednaka 74, a promenljive `x2` jednaka 87, vrednost izraza `x1 & x2` jednaka je 66. Naime, broj 74 se binarno zapisuje kao `01001010`, broj 87 kao `01010111`, i konjunkcija njihovih pojedinačnih bitova daje `01000010`, što predstavlja zapis broja 66. S obzirom na to da se prevođenje u binarni sistem efikasnije sprovodi iz heksadekadnog sistema nego iz dekadnog, prilikom korišćenja bitovskih operatora konstante se obično zapisuju heksadekadno. Tako bi u prethodnom primeru broj `x1` imao zapis `0x4A`, broj `x2` bi imao zapis `0x57`, a rezultat bi bio `0x42`.

`|` – **bitovsko *ili*** — primenom ovog operatora vrši se (obična) disjunkcija pojedinačnih bitova dva navedena argumenta. Za brojeve iz tekućeg primera, rezultat izraza `x1 | x2` bio bi `01011111`, tj. `0x5F`.

`^` – **bitovsko *ekskluzivno ili*** — primenom ovog operatora vrši se ekskluzivna disjunkcija pojedinačnih bitova dva navedena argumenta (vrednost ekskluzivne disjunkcije dva bita je `1` ako i samo ako ta dva bita imaju različite vrednosti). Za brojeve iz tekućeg primera, rezultat izraza `x1 ^ x2` je `00011101`, tj. `0x1D`.

`~` – **jedinični komplement** — primenom ovog operatora dobija se vrednost u kojoj su svi bitovi argumenta komplementirani (invertovani), a sâm argument ostaje nepromenjen. Na primer, vrednost izraza `~x1` u tekućem primeru je `10110101`, tj. `0xB5`.

`<<` – **pomeranje ulevo (šiftovanje)** — primenom ovog operatora dobija se vrednost u kojoj su bitovi prvog argumenta pomereni ulevo za broj pozicija naveden kao drugi argument (a argumenti operatora se ne menjaju). Početni bitovi (sleva) prvog argumenta se zanemaruju, dok su na završnim mestima rezultata nule. Pomeranje ulevo broja za jednu poziciju odgovara množenju sa dva (osim kada dođe do prekoračenja). Na primer, ukoliko promenljiva `x` ima tip `unsigned char` i vrednost `01010101`, tj. `0x55`, tj. 85, vrednost izraza `x << 1` jednaka je `10101010`, tj. `0xAA`, tj. 170.

`>>` – **pomeranje udesno (šiftovanje)** — primenom ovog operatora dobija se vrednost u kojoj su bitovi prvog argumenta pomereni udesno za broj pozicija naveden kao drugi argument (a argumenti operatora se ne menjaju). U izrazu `x >> n`, poslednjih `n` bitova vrednosti `x` (bitovi najmanje težine) se zanemaruje. Početni bitovi rezultata `x >> n` zavise od tipa i vrednosti promenljive `x`. Ako `x` ima neoznačen tip ili ima označen tip a nenegativnu vrednost, onda izraz `x >> n` ima vrednost dobijenu pomeranjem bitova `x` udesno za `n` pozicija, pri čemu je početnih `n` bitova rezultata jednako nuli (to je *logičko pomeranje*). U ovom slučaju, vrednost `x >> n` jednaka

je vrednosti celobrojnog deljenja `x` sa 2^n. Na primer, ukoliko je tip promenljive `x` `unsigned char` a vrednost `10010100`, tj. `0x94`, tj. 148 tada je vrednost izraza `x >> 1` broj `01001010`, tj. `0x4A`, tj. 74. Standard ne propisuje vrednost izraza `x >> n` za slučaj da `x` ima označen tip i negativnu vrednost, a većina kompilatora u ovakvim situacijama vrši *aritmetičko pomeranje* u kojem je početnih `n` bitova rezultata jednako početnom bitu vrednosti `x`.

`&=, |=, ^=, <<=, >>=` – **bitovske dodele** — ovi operatori kombinuju bitovske operatore sa dodelom (analogno, na primer, operatoru `+=`). Na primer, `a &= 1` ima isto značenje kao `a = a & 1`.

Kao unarni operator, operator `~` ima najviši prioritet i desno je asocijativan. Prioritet operatora pomeranja najviši je od svih binarnih bitovskih operatora — nalazi se između prioriteta aritmetičkih i relacijskih operatora. Ostali bitovski operatori imaju prioritet između relacijskih i logičkih operatora. Operator `&` ima viši prioritet od `^` koji ima viši prioritet od `|`. Ovi operatori imaju levu asocijativnost. Bitovski operatori dodele imaju niži prioritet (jednak ostalim operatorima dodele) i desnu asocijativnost.

Ponašanje i uloga bitovskih operatora ne smeju se mešati sa ponašanjem i ulogom logičkih operatora. Na primer, vrednost izraza `1 & 2` jednaka je `0` (`000...001 & 000..010 == 000...000`), dok je vrednost izraza `1 && 2` jednaka `1`.

Maskiranje. Kako bi se postigao željeni efekat nad bitovima datog broja, često se vrši njegovo kombinovanje bitovskim operatorima sa specijalno pripremljenim konstantama koje se obično nazivaju *maske*. U nastavku će biti prikazana upotreba nekih od najčešće korišćenih operacija ovog oblika.

Bitovska konjunkcija ima osobinu da je za svaki bit `b`, vrednost `b & 0` jednaka `0`, dok je vrednost `b & 1` jednaka `b`. Dakle, konjunkcijom broja `x` sa nekom maskom dobija se broj u kojem su očuvane sve cifre broja `x` na pozicijama na kojima maska ima bit `1`, a na svim drugim pozicijama su cifre `0`. Na primer, konjunkcijom sa maskom `0xFF` dobija se poslednji bajt broja. Pretposlednji bajt može se izdvojiti izračunavanjem `(x >> 8) & 0xFF`. Konjunkcijom sa maskom `0x01` dobija se poslednji bit broja. Da li je bit na poziciji `i` broja `x` jednak `1` može se ispitati naredbom `if (x & maska)` gde je vrednost `maska` kreirana tako da ima jedan jedini bit postavljen na `1` i to na poziciji `i` (na primer, `if (x & (1 << i))`).

Osobina disjunkcije je da je za svaki bit `b`, vrednost `b | 0` jednaka `b`, dok je vrednost `b | 1` jednaka `1`. Dakle, disjunkcijom broja `x` sa nekom maskom dobija se rezultat u kojem su jedinice upisane na sve one pozicije na kojima maska ima bit `1` (a ostali bitovi ostaju neizmenjeni). Na primer, disjunkcijom sa maskom `0xFF` upisuju se jedinice u poslednji bajt broja.

Kombinacijom prethodnih operacija maskiranja, moguće je umetati određene nizove bitova u dati broj. Tako se na primer, izrazom `(x & ~0xFF) | 0xAB`

upisuje 8 bitova `10101011` u poslednji bajt broja `x` (konjunkcijom se čiste postojeći bitovi, da bi se disjunkcijom upisao novi sadržaj).

Osobina ekskluzivne disjunkcije je da za svaki bit `b`, vrednost `b ^ 0` jednaka `b`, dok je vrednost `b ^ 1` jednaka `~b`. Ovo znači da se ekskluzivnom disjunkcijom sa nekom maskom dobija rezultat koji je jednak broju koji se dobije kada se invertuju bitovi na onim pozicijama na kojima se u maski nalazi jedinica. Na primer, ekskluzivna konjunkcija sa maskom `0xFF` invertuje poslednji bajt broja. Zanimljivo svojstvo ekskluzivne disjunkcije je da primenjena na vrednosti `x ^ maska` i `maska` daje vrednost `x`. To svojstvo omogućava, između ostalog, šifrovanje podataka: ako (samo) pošiljalac i primalac znaju šifru `maska`, pošiljalac može da šifruje svoju poruku `x` vrednošću `x ^ maska` i pošalje je primaocu; primalac, s druge strane, od vrednosti `x ^ maska` računa `(x ^ maska) ^ maska`, a to je upravo polazna poruka `x`.

Ispisivanje bitova broja. Naredna funkcija ispisuje bitove datog celog broja `x`. Vrednost bita na poziciji `i` je 0 ako i samo ako se pri konjunkciji broja x sa maskom `000..010....000` (sve nule osim jedinice na poziciji `i`) dobija 0. Funkcija kreće od pozicije najveće težine kreirajući masku koja ima jednu jedinicu i to na mestu najveće težine i zatim pomera ovu masku za jedno mesto udesno u svakoj sledećoj iteraciji sve dok maska ne postane `0`. Primetimo da je ovde bitno da maska ima tip `unsigned` kako bi se vršilo logičko pomeranje.

```
void print_bits(int x)
{
  unsigned mask = 1 << (sizeof(x) * 8 - 1);
  while (mask) {
    putchar(x & mask ? '1' : '0');
    mask >>= 1;
  }
}
```

Brojanje bitova. Prikažimo nekoliko različitih implementacija funkcije koja izračunava koliko ima bitova sa vrednošću `1` u zadatom broju `n`. Početne implementacije će biti slične i jednostavne, a kasnije nešto naprednije i sa boljom složenošću.

Osnovna ideja je da se razmatra jedan po jedan bit i da se brojač uveća kada god tekući bit ima vrednost `1`. Bit na poziciji `i` može se izdvojiti tako što se izvrši bitovska konjunkcija sa maskom koja ima sve nule osim jedinice na poziciji `i` (ona se može dobiti tako što broj `1` šiftuje za odgovarajući broj mesta ulevo).

```
int bit_count(int n)
{
  int i;
```

```
  int count = 0;
  for (i = 0; i < 8 * sizeof(n); i++)
    if (n & (1 << i))
      count++;
  return count;
}
```

Druga mogućnost je da se uvek ispituje poslednji (krajnji desni) bit broja, a da se u svakoj iteraciji broj pomera za jedno mesto udesno.

```
int bit_count(int n)
{
  int i;
  int count = 0;
  for (i = 0; i < 8 * sizeof(n); i++, n >>= 1)
    if (n & 1)
      count++;
}
```

Ukoliko je broj neoznačen, pošto se vrši logičko šiftovanje i na početne pozicije broja sleva upisuju se nule, nije potrebno vršiti sve iteracije već je moguće zaustaviti se u trenutku kada broj postane nula.

```
int bit_count(unsigned n)
{
  int count = 0;
  while (n) {
    if (n & 1) count++;
    n >>= 1;
  }
  return count;
}
```

Naglasimo da je moguće i malo drugačije rešenje, u kojem bi se umesto šiftovanja broja udesno u svakom koraku maska šiftovala ulevo (u takvom rešenju bilo bi potrebno ispitati sve bitove).

Ukoliko broj `n` nije nula, izraz `n & (n-1)` invertuje poslednju jedinicu u njegovom binarnom zapisu (ostale cifre ostaju nepromenjene). Zaista, ukoliko je broj `n` oblika `...10000`, u slučaju potpunog komplementa broj `n-1` je oblika `...01111` pa je njihova konjunkcija oblika `...00000`, pri čemu je prefiks (označen sa `...`) nepromenjen. Imajući ovo u vidu, moguće je implementirati prethodnu funkciju tako da broj iteracija bude jednak broju jedinica u broju `n`.

```
int bit_count(unsigned n)
{
  int count = 0;
  while (n) {
    count++;
    n = n & (n-1);
  }
  return count;
}
```

Ovo je moguće iskoristiti kako bi se ispitalo da li je broj stepen broja dva. Naime, broj je stepen broja dva ako i samo ako ima tačno jednu jedinicu u binarnom zapisu (specijalan slučaj nule se mora posebno obraditi).

```
int is_pow2(unsigned n)
{
  return n != 0 && !(n & (n - 1));
}
```

Asimptotski brže rešenje za problem brojanja bitova moguće je dobiti ako se primeni takozvano paralelno brojanje bitova. Taj algoritam zasnovan je na algoritamskoj tehnici *podeli i vladaj (engl. divide and conquer)*. Broj jedinica u binarnom zapisu 32-bitnog broja može se svesti na sabiranje broja jedinica u binarnom zapisu dve njegove 16-bitne polovine, broj jedinica u svakoj 16-bitnoj polovini na sabiranje broja jedinica u svakoj od njene dve 8-bitne polovine i tako dalje. „Trik“ koji se koristi je da se broj jedinica u svakoj od dve polovine broja može zapisati umesto samih bitova te polovine broja. Naime, pomoću k bitova neoznačenog broja mogu se zapisati brojevi od 0 do $2^k - 1$, a za svako $k \geq 1$ važi da je $2^k - 1 \geq k$. Nakon što prebrojimo jedinice u svakoj od dve 16-bitne polovine broja i zapišemo te brojeve u svaku od te dve 16-bitne polovine, preostaje da saberemo neoznačene brojeve koji su zapisani u te dve 16-bitne polovine. To možemo uraditi tako što levu polovinu šiftujemo za 16 mesta udesno i saberemo sa desnom polovinom. Postupak se nastavlja rekurzivno. Izlaz iz rekurzije predstavlja zadatak u kom je potrebno izračunati broj jedinica u svakom od tridest i dva 1-bitna bloka i zapisati rezultat u taj blok. Međutim, u tom slučaju ne treba raditi zapravo ništa jer bitovi koji imaju vrednost 0 imaju zapravo nula jedinica, a blokovi koji imaju vrednost 1 imaju zapravo jednu jedinicu, pa je na svakoj bitovskoj poziciji upravo zapisan broj jedinica na toj bitovskoj poziciji.

Algoritam se može realizovati i iterativno, eliminisanjem rekurzije. Ideja je razmatrati tridesetdvobitni broj kao niz od 16 uzastopnih parova jednobitnih brojeva i zatim sabiranjem svakog para dobiti 16 uzastopnih parova dvobitnih brojeva koji sadrže njihove sume. Ovo je moguće uraditi izdvajanjem bitova na neparnim pozicijama (konjunkcijom sa maskom `0x55555555`), zatim pomeranjem broja za jedno mesto udesno i ponovnim izdvajanjem bitova na neparnim

pozicijama (ono što se dobije su bitovi na parnim pozicijama polaznog broja pomereni za jednu poziciju desno) i zatim primenom operatora sabiranja. Na primer, ukoliko je polazni broj

01011011001010100110110101011011,

sabiraju se brojevi

01 01 00 01 00 00 00 00 01 00 01 01 01 01 00 01 i
00 00 01 01 00 01 01 01 00 01 01 00 00 00 01 01.

Dobija se rezultat

01 01 01 10 00 01 01 01 01 01 10 01 01 01 01 10

Rezultat ukazuje da se u svakom od prva tri para bitova polaznog broja nalazi po jedna jedinica, da se u sledećem paru nalaze dve jedinice, zatim ide par sa nula jedinica itd.

Prilikom ovog sabiranja „paralelno“ se sabiraju svi parovi pojedinačnih bitova (kako je svaki bit na parnim pozicijama argumenata jednak 0, sabiranja pojedinih parova su nezavisna u smislu da ne može da nastupi prekoračenje koje bi se sa jednog para proširilo na sledeći).

Ovaj postupak dalje se nastavlja tako što se sabira 8 parova dvobitnih brojeva i dobija 8 četvorobitnih brojeva koji čuvaju brojeve jedinica u svakoj četvorki bitova polaznog broja. U prethodnom primeru se dobija

0010 0011 0001 0010 0010 0011 0010 0011

što ukazuje na to da prva četvorka bitova polaznog broja ima dve jedinice, druga tri jedinice, treća jednu, itd.

Zatim se ova 4 para četvorobitnih brojeva sabiraju i dobijaju se četiri osmobitna broja koja sadrže brojeve jedinica u pojedinačnim bajtovima polaznog broja. U prethodnom primeru se dobija

00000101 00000011 00000101 00000101

što govori da prvi bajt ima 5 jedinica, drugi 3, treći 5 i četvrti 5.

Zatim se ova dva para osmobitnih brojeva sabiraju i dobijaju se dva šesnaestobitna broja koji sadrže brojeve jedinica u dvobajtima polaznog broja. U prethodnom primeru dobija se vrednost

0000000000001000 0000000000001010

što govori da prvi dvobajt ima 8 jedinica, a drugi 10.

Napokon, sabira se i ovaj par šesnaestobitnih brojeva i dobija se trideset-dvobitni broj koji sadrži konačan rezultat.

00000000000000000000000000010010

Prethodni postupak implementira funkcija:

```
int bit_count(unsigned n)
{
   n  = (n & 0x55555555) + ((n >> 1) & 0x55555555);
   n  = (n & 0x33333333) + ((n >> 2) & 0x33333333);
   n  = (n & 0x0F0F0F0F) + ((n >> 4) & 0x0F0F0F0F);
   n  = (n & 0x00FF00FF) + ((n >> 8) & 0x00FF00FF);
   n  = (n & 0x0000FFFF) + ((n >> 16) & 0x0000FFFF);
```

```
    return n;
}
```

Još jedna od mogućih ideja je napraviti statički niz u kojem je preračunat broj bitova za svaku od 256 vrednosti bajta, a zatim čitati podatke iz tog niza.

```
int bit_count(unsigned n)
{
  static int lookup[256] = {0, 1, 1, 2, 1, 2, 2, 3, ...};
  return lookup[n & 0xFF] + lookup[n >> 8 & 0xFF] +
         lookup[n >> 16 & 0xFF] + lookup[n >> 24 & 0xFF];
}
```

Izdvajanje bitova. Naredna funkcija vraća `n` bitova broja `x` koji počinju na poziciji `p` (pozicije se broje od nulte, zdesna nalevo) i prostiru se dalje ulevo (to su bitovi na pozicijama `p`, `p+1`, . . . , `p+n-1`).

```
unsigned get_bits(unsigned x, int p, int n)
{
  /* Kreiramo masku koja ima poslednjih n jedinica:
     0000000...00011111 */
  unsigned last_n_1 = ~(~0 << n);
  /* x pomeramo udesno za odgovarajuci broj mesta, a zatim
     konjunkcijom sa  maskom brisemo suvisne cifre */
  return (x >> p) & last_n_1;
}
```

Izmena bitova. Naredna funkcija vraća broj koji se dobija od broja `x` izmenom `n` bitova koji se prostiru od pozicije `p` (pozicije se broje od nulte, zdesna nalevo) pa dalje ulevo (to su bitovi na pozicijama `p`, `p+1`, . . . , `p+n-1`), tako što se na ta mesta upisuje `n` krajnjih desnih bitova broja `y`. Na primer, ako je `x` jednako `10110110`, `y` jednako `10111001` i razmatraju se `n=2` bita počevši od pozicije `p=3`, u broju `x` menjaju se bitovi `10` (to su podvučeni bitovi `101`<u>`10`</u>`110`) bitovima `01` broja `y` (to su podvučeni bitovi `1011100`<u>`1`</u>) i dobija se broj `101`<u>`01`</u>`110`.

```
unsigned set_bits(unsigned x, int p, int n, unsigned y)
{
  /* Maska 000000...000111111  - poslednjih n jedinica */
  unsigned last_n_1 = ~(~0 << n);
  /* Maska 1111100..000111111 - n nula pocevsi od pozicije p */
  unsigned middle_n_0 = ~(last_n_1 << p);
  /* Brisemo n bitova pocevsi od pozicije p */
  x = x & middle_n_0;
  /* Izdvajamo poslednjih n bitova broja y i
```

```
    pomeramo ih na poziciju p */
  y = (y & last_n_1) << p;
  /* Upisujemo bitove broja y u broj x i vracamo rezultat */
  return x | y;
}
```

Rotacija bitova. Razmotrimo funkciju koja vrši rotaciju neoznačenog broja `x` za `n` pozicija udesno. Naivan način da se ona realizuje je da se `n` puta izvrši rotacija za jedno mesto udesno.

```
unsigned right_rotate(unsigned x, int n)
{
  int i, width = sizeof(unsigned) * 8;
  /* Postupak se ponavlja n puta */
  for (i = 0; i < n; i++) {
    /* Poslednji bit (zdesna) broja x */
    unsigned last_bit = x & 1;
    /* x pomeramo za jedno mesto udesno */
    x >>= 1;
    /* Zapamceni poslednji bit stavljamo na pocetak broja x*/
    x |= last_bit << width - 1;
  }
  return x;
}
```

Bolje rešenje je da se rotacija izvrši u jednom koraku.

```
unsigned right_rotate(unsigned x, int n)
{
  int i, width = sizeof(unsigned) * 8;
  /* Poslednjih n-bitova broja x */
  unsigned last_n_bits = x & ~(~0 << n);
  /* Pomeramo x za n mesta udesno */
  x >>= n;
  /* Na pocetak mu upisujemo upamcenih n bitova */
  x |= last_n_bits << width - n;
  return x;
}
```

Refleksija bitova. Naredna funkcija obrće binarni zapis neoznačenog broja `x` tako što bitove „čita unatrag“.

```
unsigned mirror(unsigned x)
{
```

```
  int i, width = sizeof(unsigned) * 8;
  /* Rezultat inicijalizujemo na poslednji bit broja x */
  unsigned y = x & 1;
  /* Postupak se ponavlja width - 1 puta */
  for (i = 1; i < width; i++) {
    /* x se pomera udesno za jedno mesto */
    x >>= 1;
    /* rezultat se pomera ulevo za jedno mesto */
    y <<= 1;
    /* Poslednji bit broja x upisujemo na kraj rezultata */
    y |= x & 1;
  }
  return y;
}
```

Pitanja i zadaci za vežbu

Pitanje 5.45. *Koju vrednost ima izraz* `0xA3 & 0x24 << 2` *?*
Koju vrednost ima izraz `0xB8 | 0x51 << 3` *?*
Koju vrednost ima izraz `0x12 ^ 0x34` *?*
Koju vrednost ima izraz `0x34 << 2` *?*
Koju vrednost ima izraz `(13 & 17) << 2` *?*

Pitanje 5.46. *Neka je* `x` *neoznačeni ceo broj. Napisati izraz (ne funkciju ili naredbu)*
(a) *čija je vrednost broj koji se dobije invertovanjem bajta najmanje težine broja* `x`*?*
(b) *koji invertuje tri bita najmanje težine broja* `x`*.*
(c) *koji vraća treći bit najmanje težine broja* `x`*.*
(d) *čija je vrednost broj koji se dobija tako što se upišu sve jedinice u bajt najmanje težine u zapisu broja* `x`*.*

Pitanje 5.47.
(a) *Ako je promenljiva tipa* `unsigned int`*, kada izraz* `n & (n-1)` *ima vrednost* `0`*?*
(b) *Ako broj* `n` *nije nula, čemu je jednaka vrednost izraza* `n & (n-1)` *?*
(c) *Napisati funkciju kojom se proverava da li je dati neoznačen ceo broj stepen broja 2.*

Pitanje 5.48. *Neka su* `x` *i* `y` *promenljive tipa* `unsigned char`*. Pretpostavimo da se bitovi broje zdesna nalevo (bit najmanje težine je nulti bit). Napisati izraz koji je tačan ako i samo ako su*
(a) *različiti nulti bit vrednosti* `x` *i sedmi bit vrednosti* `y`*;*
(b) *jednake vrednosti bitova na trećem bitu za* `x` *i* `y`*;*
(c) *različite vrednosti bitova na sedmom bitu za* `x` *i* `y`*.*

Pitanje 5.49. *Neka je* `c` *promenljiva tipa* `unsigned char`*. Pretpostavimo da se bitovi broje zdesna nalevo (bit najmanje težine je nulti bit). Napisati izraz koji vraća:*
(a) *vrednost u kojoj je na nultom bitu odgovarajući bit vrednosti* `c`*, a svi ostali bitovu su jednaki* `0`*;*
(b) *vrednost u kojoj je na šestom bitu odgovarajući bit vrednosti* `c`*, a svi ostali bitovu su jednaki* `1`*.*

Pitanje 5.50. *Neka je* `x` *32-bitan neoznačeni ceo broj, a* `c` *neoznačeni karakter. Napisati izraz kojim se* `c` *inicijalizuje na vrednost bajta najveće težine broja* `x`*.*

Pitanje 5.51. *Po uzoru na algoritam paralelnog brojanja bitova, opisati algoritam kojim se efikasno određuje parnost broja bitova.*

Pitanje 5.52. *Opisati algoritam kojim se može izračunati vrednost celog dela logaritma za osnovu 2 neoznačenog 32-bitnog celog broja* `x`*. Algoritam treba da ima logaritamsku složenost u odnosu na broj bitova u zapisu broja* `x`*.*

Pitanje 5.53. *Po uzoru na algoritam paralelnog brojanja bitova, opisati efikasan algoritam za određivanje broja završnih nula u binarnom zapisu 32-bitnog neoznačenog celog broja* `x` *(npr. broj 1011..10110000 ima 4 završne nule). Algoritam treba da ima logaritamsku složenost u odnosu na broj bitova u zapisu broja* `x`*. Rešiti zadatak i algoritmom binarne pretrage.*

Pitanje 5.54. *Opisati algoritam koji zaokružuje dati 32-bitni neoznačen ceo broj na najmanji stepen dvojke koji je veći ili jednak od tog broja (pretpostaviti da je bit najveće težine polaznog broja jednak nuli). Algoritam treba da ima logaritamsku složenost u odnosu na broj bitova u zapisu broja* `x`*. Ideja: prekopirati jedinicu najveće težine na sve pozicije iza nje, a zatim dodati 1 (obratiti posebnu pažnju na slučaj kada je polazni broj stepen dvojke).*

Zadatak 5.43. *Definisati funkciju* `f` *sa argumentom* `c` *tipa* `unsigned char`*, koja vraća:*
(a) *vrednost* `c` *u kojoj su razmenjene vrednosti prvog i poslednjeg bita.*
(b) *vrednost* `c` *u kojoj su razmenjene vrednosti bita najmanje težine i njemu susednog bita.*

Zadatak 5.44. *Definisati funkciju koja invertuje* `n` *bitova počevši od pozicije* `p` *(bitovi se broje zdesna nalevo, bit najmanje težine je nulti bit) u broju* `x`*.*

Zadatak 5.45. *Definisati funkciju koja izračunava razliku zbira bitova na parnim i zbira bitova na neparnim pozicijama neoznačenog celog broja* `x`*.*

Zadatak 5.46. *Definisati funkciju koja vraća broj dobijen od neoznačenog celog broja* `x` *kada se prvih 8 bitova broja* `x` *na najvećim težinama postavi na 0, poslednja 4 bita sa najmanjim težinama broja* `x` *na 0110, a ostatak broja* `x` *ostaje nepromenjen.*

Zadatak 5.47. *Definisati funkciju koja menja mesta prvih i poslednjih* `p` *bitova neoznačenog broja* `x` *(na primer, za* `p = 3` *i broj* `x` = `111....010` *treba da se dobije rezultat* `x` = `010...111`*)*

Zadatak 5.48. *Definisati funkciju* `int count_zero_pairs(unsigned x)` *koja broji koliko se puta kombinacija* 00 *(dve uzastopne nule) pojavljuje u binarnom zapisu celog neoznačenog broja (ako se u bitskoj reprezentaciji nekog broja jave tri uzastopne nule, središnja se broji dva puta, u levom i desnom paru).*

Zadatak 5.49. *Definisati funkciju koja vraća broj koji je odraz u ogledalu polaznog neoznačenog karaktera* `c`*. Na primer, ako je ulaz broj čiji je binarni zapis* `10000101`*, izlaz je broj čiji je binarni zapis* `10100001`*.*

Zadatak 5.50. *Definisati funkciju koja parametru* `x` *tipa* `unsigned int` *razmenjuje vrednosti bitova na pozicijama* `i` *i* `j` *i vraća rezultat kao povratnu vrednost. Bitovi se broje zdesna (bit sa najmanjom težinom nalazi se na poziciji 0, bit do njega na poziciji 1, itd).*

Glava 6

Osnovne strukture podataka

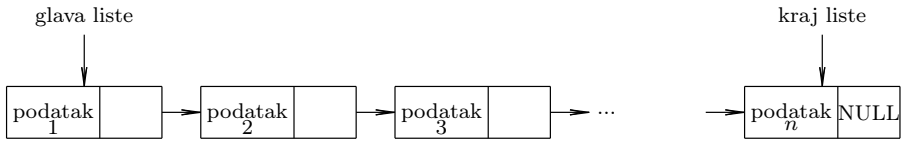
Dinamička alokacija memorije omogućava građenje specifičnih, dinamičkih struktura podataka koje su u nekim situacijama pogodnije od statičkih zbog svoje fleksibilnosti. Najznačajnije među njima su povezane liste i stabla čiji elementi mogu da sadrže podatke proizvoljnog tipa.

Ovakve i druge slične strukture u C-u nisu neposredno raspoložive, ni kao deo jezika ni kroz standardnu biblioteku, te se implementiraju namenski, za konkretne primene (obično u skladu sa nekom opštom, dobro poznatom formom). U drugim, novijim jezicima ovakve strukture obično se ne implementiraju za svaku primenu iznova, jer su raspoložive kroz namenske biblioteke, u vidu koji je prilagodljiv raznovrsnim potrebama.

6.1 Jednostruko povezana lista

Jednostruko povezana lista je dinamička struktura u kojoj su pojedinačni elementi povezani pokazivačima. U ovoj strukturi postoji veza pokazivačem (i to samo u jednom smeru, jednim pokazivačem) samo između susednih elemenata. U jeziku C ne postoji tip koji opisuje povezane liste, već se one kreiraju koristeći druge tipove i, najčešće, pokazivače. U tom slučaju, jednostruko povezanu listu čine elementi (zovemo ih i *čvorovi liste*) od kojih svaki sadrži podatak izabranog tipa i pokazivač. Svaki pokazivač pokazuje na jedan (sledeći) element liste i ti pokazivači povezuju elemente u jednu celinu — u listu. Izuzetak je poslednji čvor čiji pokazivač ima vrednost `NULL` (slika 6.1). U daljem tekstu će se, ukoliko se ne naglasi drugačije, pod povezanom listom podrazumevati jednostruko povezana lista.

Listi se obično pristupa preko pokazivača na njen početni element, tj. na njen *početak*, *glavu liste*. Zato se pokazivač na početak liste mora čuvati i održavati (tokom svih operacija koje mogu da ga promene) jer omogućavaju pristup svim elementima liste. Iz sličnih razloga, u nekim situacijama potrebno je čuvati i održavati pokazivač na poslednji element liste, tj. na njen *kraj*.



Slika 6.1: Ilustracija povezane liste

6.1.1 Odnos povezanih lista i nizova

Više podataka (elemenata) istog tipa u celinu mogu da objedinjuju povezane liste, statički alocirani nizovi, kao i dinamički nizovi (tj. dinamički alocirani blokovi memorije). Međutim, povezane liste i statički i dinamički nizovi bitno se razlikuju po više aspekata.

Niz (bilo statički ili dinamički) ima fiksnu veličinu. Veličina statičkog niza mora biti poznata u fazi kompilacije, a kod dinamičkog niza mora biti takva da u fazi izvršavanja, u trenutku alokacije, u memoriji postoji dovoljno veliki povezani blok slobodne memorije. Veličina dinamičkog niza može se, funkcijom `realloc`, menjati samo od njegovog kraja (a i to može da bude zahtevna operacija zbog potencijalnog ponovnog alociranja i kopiranja sa jedne na drugu lokaciju). S druge strane, broj elemenata liste može se smanjivati i povećavati tokom izvršavanja programa, jednostavnim dodavanjem ili brisanjem pojedinačnih elemenata. Prilikom dodavanja novog elementa u listu potrebno je dinamički alocirati prostor samo za taj jedan element. Česta dodavanja i brisanja elemenata liste, međutim, dovode do fragmentisanja memorije, pa konkretno vreme za dodavanje novog elementa u nekim programima može značajno da raste kako se program izvršava (taj problem je manje izražen kod dinamičkih nizova jer se alociraju blokovi više elemenata, pa se samim tim alokacija i dealokacije izvršavaju ređe).

Elementi i statičkog i dinamičkog niza smešteni su u uzastopnim memorijskim lokacijama, te se pozicija u memoriji i-tog elementa može jednostavno izračunati na osnovu pozicije početnog elementa i vrednosti i. Zbog toga se i-tom elementu niza može pristupiti u vremenu $O(1)$. S druge strane, elementi liste smešteni su u memorijskim lokacijama koje nisu nužno uzastopne i mogu biti razbacane po memoriji. Da bi se pristupilo jednom elementu liste, potrebno je krenuti od početnog elementa i pratiti pokazivače sve dok se ne naiđe na traženi element, te je vreme potrebno za pristup $O(n)$ (gde je n broj elemenata liste). Obilazak svih elemenata redom zahteva vreme $O(n)$ i kod lista i kod nizova (ipak, zbog boljeg iskorišćavanja keš-memorije, konkretno utrošeno vreme obično je nešto kraće u slučaju niza).

Umetanje elementa, izbacivanje elementa i promena poretka elemenata niza (bilo statičkog ili dinamičkog) vremenski je zahtevno jer može da uključuje kopiranje velikih blokova memorije. S druge strane, umetanje elementa u po-

vezanu listu, ako se ubacuje na početak ili ako se zna adresa elementa nakon kojeg se ubacuje, zahteva vreme $O(1)$. Izbacivanje elementa liste i promena poretka elemenata liste takođe su jednostavne operacije koje se svode na izmene pokazivača.

Sve u svemu — nijedna od navedenih struktura podataka (povezane liste, statički nizovi, dinamički nizovi) nije uvek najbolji izbor i nema svojstva uvek bolja od druga dva. Najbolji izbor vezan je za specifičnosti konkretnog problema i najvažnije zahteve. Na primer, ukoliko postoji potreba za brzim pristupom pojedinačnim elementima niza, dobar izbor je statički ili dinamički niz. Ukoliko je i pre izvršavanja programa moguće znati (ili makar grubo proceniti) potreban broj elemenata, najbolji izbor često je statički niz. Ukoliko se unapred ne zna broj elemenata, ali se očekuje da nema mnogo dodavanja i izbacivanja elemenata, najbolji izbor često je dinamički niz. Ukoliko se unapred ne zna broj elemenata, ali postoji opasnost da oni ne mogu biti smešteni u povezani memorijski blok i očekuje se da ima mnogo dodavanja i izbacivanja elemenata, najbolji izbor često je povezana lista. Liste su pogodnije od nizova i u situacijama u kojima su pojedinačni elementi veliki podaci – kod dinamičkih nizova bi se tokom realokacije kopirale velike količine podataka što može biti neefikasno, a toga nema ako se koriste liste. Dodatno, postoje i „hibridne“ strukture podataka koje kombinuju dobre osobine opisanih struktura (na primer, struktura u kojoj se za podatke dinamički alociraju blokovi podataka fiksne širine i ne vraćaju operativnom sistemu kada više nisu potrebni, nego se iznova koriste kada to zatreba).

6.1.2 Elementi povezane liste

Tip elementa ili čvora liste obično se definiše na sledeći način:

```
struct cvor {
  <type> podatak;
  struct cvor* sledeci;
};
```

U navedenoj definiciji, `<type>` označava tip podatka koji element sadrži. To može biti proizvoljan tip (moguće i struktura), a može biti i pokazivač `void*` ako se želi generička lista koja se lako može prilagoditi nekom konkretnom tipu. Element liste može da sadrži i više od jednog podatka (i, naravno, ti podaci ne moraju da budu istog tipa). Član strukture `struct cvor* sledeci;` je dozvoljen u okviru same definicije strukture `struct cvor`, jer se zna koliko memorije zahteva (onoliko koliko zahteva i bilo koji drugi pokazivač).

Koristeći `typedef` može se uvesti ime strukture koje ne sadrži ključnu reč `struct` (element `sledeci` ipak mora da se deklariše tipom `struct cvor`):

```
typedef struct cvor {
```

```
  <type> podatak;
  struct cvor* sledeci;
} cvor;
```

Dozvoljeno je da ime strukture i novo ime tipa budu jednaki (`cvor` u ovom primeru). Novo ime tipa za `struct cvor` moglo je da bude uvedeno i pre i nakon definicije strukture (ali se novo ime ne može koristiti pre naredbe `typedef`):

```
typedef struct _cvor cvor;

struct _cvor {
  <type> podatak;
  cvor* sledeci;
};
```

ili

```
struct _cvor {
  <type> podatak;
  struct _cvor* sledeci;
};

typedef struct _cvor cvor;
```

Ponekad se imenuje tip pokazivača na čvor liste.

```
typedef struct _cvor* pCvor;
struct _cvor {
  <type> podatak;
  pCvor sledeci;
};
```

Jednostavnosti radi, u primerima u nastavku biće smatrano da je element liste deklarisan na sledeći način (tj. da ima samo jedan podatak, podatak tipa `int`):

```
typedef struct cvor {
  int podatak;
  struct cvor *sledeci;
} cvor;
```

Elementi liste kreiraju se pojedinačno i onda spajaju sa tekućom povezanom listom. Novi element liste može biti kreiran sledećom funkcijom.

```
cvor *novi_cvor(int podatak)
{
  cvor *novi = malloc(sizeof(cvor));
  if (novi == NULL)
    return NULL;
  novi->podatak = podatak;
  return novi;
}
```

Zapis `novi->podatak` je kraći zapis za `(*novi).podatak` (isto važi za bilo koji pokazivač na strukturu, dakle umesto `(*a).podatak` može se pisati `a->podatak`).

Ukoliko element liste sadrži više podataka, onda bi navedena funkcija za kreiranje elementa imala po jedan parametar za svaki od njih.

Ako element liste treba da sadrži nisku, moguće je da ta niska bude predstavljena nizom karaktera koji se čuva u elementu liste. Tada je potrebno znati gornje ograničenje za dužinu niske (računajući i završnu nulu), kao u narednom primeru:

```
typedef struct {
  char podatak[MAX];
  struct cvor *sledeci;
} cvor;
```

Prilikom alokacije niske, tada bi umesto dodele, podatak bilo potrebno inicijalizovati funkcijom `strcpy`.

```
cvor *novi_cvor(const char *podatak)
{
  ...
  strcpy(novi->podatak, podatak);
}
```

Druga mogućnost je da se u elementu povezane liste čuva samo pokazivač, a da se niz karaktera alocira i dealocira nezavisno od liste:

```
typedef struct {
  char *podatak;
  struct cvor *sledeci;
} cvor;
```

Onda je, prilikom kreiranja novog elementa liste, moguće izvršiti bilo plitko, bilo duboko kopiranje (videti prvi deo ove knjige, poglavlje 10.1, „Pokazivači i adrese“). U prvom slučaju, sa plitkim kopiranjem, kreiranje novog elementa liste može imati sledeću formu:

```
cvor *novi_cvor(char *podatak)
{
  ...
  novi->podatak = podatak;
}
```

Sa ovim pristupom treba biti obazriv jer je moguće da se istom nizu karaktera pristupa preko više pokazivača (što ponekad može biti opravdano). Plitko kopiranje uglavnom ima smisla ako se podaci u čvorovima liste samo čitaju a ne menjaju se. Tada ima smisla primeniti kvalifikator `const` na tip argumenta čime se dodatno osiguravamo od slučajnih promena podataka unutar funkcije (ako bi tip argumenta bio `const char *podatak`, onda bi kompilator prijavio grešku pri svakom pokušaju modifikacije tog podatka unutar funkcije). Sa druge strane, ne postoji garancija da se podaci koji se čuvaju u listi neće promeniti iz neke druge funkcije. Naime, funkciji koja prima podatak tipa `const char*` može se proslediti i pokazivač tipa `char*`, a sadržaj na koji on ukazuje može se (negde drugde) nesmetano menjati.

U drugom slučaju (sa dubokim kopiranjem) potrebno je alocirati potreban prostor i iskopirati nisku (na primer, funkcijama `malloc` i `strcpy` ili, kraće, funkcijom `strdup`), a prilikom uklanjanja čvorova liste, potrebno je ukloniti i ovako alocirane kopije (funkcijom `free`).

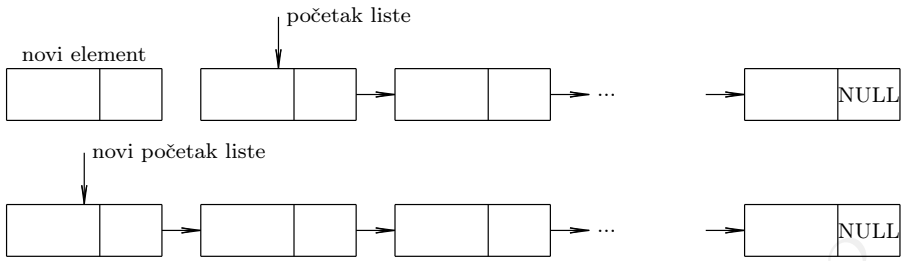
```
cvor *novi_cvor(const char *podatak)
{
  ...
  novi->podatak = strdup(podatak);
}
```

6.1.3 Osnovne operacije za rad sa povezanim listama

Kao što je već rečeno, ne postoji tip koji opisuje listu, već se listi obično pristupa preko pokazivača na njen početak i/ili kraj. Dodavanje novog elementa na početak liste može se implementirati na sledeći način (slika 6.2):

```
cvor *dodaj_cvor_na_pocetak(cvor *pocetak, cvor *novi)
{
  novi->sledeci = pocetak;
  return novi;
}
```

Umesto da funkcija vraća pokazivač na početak liste dobijene dodavanjem jednog elementa, ona može da menja zadati parametar. Takav stil implementacije često je pogodniji jer ostavlja mogućnost da se kroz povratnu vrednost šalje informacija o nekakvom statusu (a ne pokazivač na početak liste):



Slika 6.2: Ilustracija dodavanja elementa na početak povezane liste

```
typedef enum {
  OK,
  NEDOVOLJNO_MEMORIJE,
  NEMA_ELEMENATA
} status;

/* ... */

status dodaj_na_pocetak(cvor **pokazivac_na_pocetak,
                        cvor *novi)
{
  novi->sledeci = *pokazivac_na_pocetak;
  *pokazivac_na_pocetak = novi;
  return OK;
}
```

Dodatno, možemo napraviti funkciju koja vrši i alokaciju čvora, pre nego što ga uveže u listu.

```
status dodaj_na_pocetak(cvor **pokazivac_na_pocetak,
                        int podatak)
{
  cvor *novi = novi_cvor(podatak);
  if (novi == NULL)
    return NEDOVOLJNO_MEMORIJE;
   novi->sledeci = *pokazivac_na_pocetak;
   *pokazivac_na_pocetak = novi;
  return OK;
}
```

Navedena funkcija ima složenost $O(1)$.
Idiomska forma za prolazak kroz sve elemente povezane liste je:

```
for (p = pocetak; p != NULL; p = p->sledeci)
  ...
```

ili, ako se ide samo do poslednjeg elementa u povezanoj listi:

```
for (p = pocetak; p->sledeci != NULL; p = p->sledeci)
  ...
```

Navedena petlja podrazumeva da `pocetak` nema vrednost `NULL` (u suprotnom bi došlo do greške u izvršavanju petlje već u prvoj proveri uslova). Koristeći navedenu idiomsku formu, novi član može se dodati i na kraj liste, na primer, sledećom funkcijom:

```
cvor *dodaj_cvor_na_kraj(cvor *pocetak, cvor *novi)
{
  cvor *p;
  novi->sledeci = NULL;

  if (pocetak == NULL)
    return novi;

  for (p = pocetak; p->sledeci != NULL; p = p->sledeci)
    ;
  p->sledeci = novi;
  return pocetak;
}
```

Umesto da funkcija vraća pokazivač na početak liste dobijene dodavanjem jednog elementa na kraj, ona može da menja zadati parametar (pokazivač na početak liste) i ujedno sama poziva funkciju koja alocira novi čvor:

```
status dodaj_na_kraj(cvor **pokazivac_na_pocetak, int podatak)
{
  cvor *novi = novi_cvor(podatak);
  if (novi == NULL)
    return NEDOVOLJNO_MEMORIJE;

  cvor *p;
  novi->sledeci = NULL;

  if (*pokazivac_na_pocetak == NULL)
    *pokazivac_na_pocetak = novi;
  else {
    for (p = *pokazivac_na_pocetak;
         p->sledeci != NULL;
```

```
        p = p->sledeci)
      ;
    p->sledeci = novi;
  }
  return OK;
}
```

Složenost prethodne dve funkcije je reda $O(n)$, gde je n broj elemenata liste. Naglasimo da se ovakve funkcije nikada ne koriste zbog svoje neefikasnosti: ukoliko je potrebna opisana funkcionalnost, onda je potrebno promeniti sâmu definiciju strukture kako bi se omogućilo efikasno izvršavanje date funkcionalnosti. Na primer, dodavanje elementa na kraj liste može da se izvrši i u vremenu $O(1)$, ali onda je potrebno održavanje i pokazivača na poslednji element liste:

```
status dodaj_na_kraj(cvor **pokazivac_na_pocetak,
                     cvor **pokazivac_na_kraj, int podatak)
{
  cvor *novi = novi_cvor(podatak);
  if (novi == NULL)
    return NEDOVOLJNO_MEMORIJE;
  novi->sledeci = NULL;
  if (*pokazivac_na_pocetak == NULL)
    *pokazivac_na_pocetak = novi;
  else
    (*pokazivac_na_kraj)->sledeci = novi;
  *pokazivac_na_kraj = novi;
  return OK;
}
```

Navedena funkcija podrazumeva da su zadati pokazivači na pokazivače na početak i na kraj liste, kako bi se sami pokazivači na početak i na kraj mogli menjati. Novi element dodaje se na kraj, ali i početak liste se može promeniti – onda kada je lista inicijalno prazna. Analogno se može implementirati i funkcija koja novi element dodaje na početak liste, a održava i pokazivač na početak i pokazivač na kraj:

```
status dodaj_na_pocetak(cvor** pokazivac_na_pocetak,
                        cvor** pokazivac_na_kraj, int podatak)
{
  cvor *novi = novi_cvor(podatak);
  if (novi == NULL)
     return NEDOVOLJNO_MEMORIJE;
  novi->sledeci = NULL;
  if (*pokazivac_na_pocetak == NULL)
      *pokazivac_na_kraj = novi;
```

```
  else
      novi->sledeci = *pokazivac_na_pocetak;
  *pokazivac_na_pocetak = novi;
  return OK;
}
```

Umesto održavanja dva nezavisna pokazivača (na početak i na kraj liste), ponekad se kreira struktura koja ih objedinjuje (međutim, to rešenje neće ovde biti prikazano).

Sadržaj liste može se ispisati sledećom funkcijom:

```
void ispisi_elemente_liste(cvor *pocetak)
{
  cvor* tekuci;
  for (tekuci = pocetak; tekuci != NULL; tekuci = tekuci->sledeci)
    printf("%d ", tekuci->podatak);
}
```

ili njenom rekurzivnom verzijom:

```
void ispisi_elemente_liste(cvor *pocetak)
{
  if (pocetak != NULL) {
    printf("%d ", pocetak->podatak);
    ispisi_elemente_liste(pocetak->sledeci);
  }
}
```

Rekurzivno mogu da se implementiraju i druge gore navedene funkcije, ali te rekurzivne verzije nisu mnogo jednostavnije od iterativnih. Međutim, rekurzivna verzija ispisa elemenata liste može se (za razliku od iterativne) lako prepraviti da ispisuje elemente liste u obratnom poretku:

```
void ispisi_listu_obratno(cvor *pocetak)
{
  if (pocetak != NULL) {
    ispisi_listu_obratno(pocetak->sledeci);
    printf("%d ", pocetak->podatak);
  }
}
```

Naglasimo da rekurzivno rešenje u ovom kontekstu može biti veoma loše jer će na programskom steku biti formirano onoliko stek okvira koliko ima elemenata liste, što može dovesti do prekoračenja predviđene veličine programskog steka i prekida izvršavanja programa. Zato je, u ovakvim i sličnim situacijama, poželjno osloboditi se rekurzije.

Početni element liste može se pročitati i osloboditi primenom sledeće funkcije, koja kroz listu argumenata vraća podatak koji je sadržao taj element:

```
status obrisi_sa_pocetka(cvor **pokazivac_na_pocetak,
                         int *podatak)
{
  cvor *p;
  if (*pokazivac_na_pocetak == NULL)
    return NEMA_ELEMENATA;

  p = *pokazivac_na_pocetak;
  *podatak = (*pokazivac_na_pocetak)->podatak;
  *pokazivac_na_pocetak = (*pokazivac_na_pocetak)->sledeci;
  free(p);
  return OK;
}
```

Ako lista nije bila prazna, navedena funkcija vraća (kroz argument) podatak upisan u element koji je na početku povezane liste, briše ga i vraća vrednost `OK`. Ako je lista bila prazna, vraća se vrednost `NEMA_ELEMENATA`.

Ako je potrebno održavati i pokazivač na kraj liste, onda funkcija može da izgleda ovako:

```
status obrisi_sa_pocetka(cvor **pokazivac_na_pocetak,
                         cvor **pokazivac_na_kraj,
                         int *podatak)
{
  cvor *p;
  if (*pokazivac_na_pocetak == NULL)
    return NEMA_ELEMENATA;

  p = *pokazivac_na_pocetak;
  *podatak = (*pokazivac_na_pocetak)->podatak;
  *pokazivac_na_pocetak = (*pokazivac_na_pocetak)->sledeci;
  free(p);

  if (*pokazivac_na_pocetak == NULL)
     *pokazivac_na_kraj = NULL;
  return OK;
}
```

Oslobađanje čitave povezane liste (tj. brisanje svih njenih elemenata) može se izvršiti sledećom funkcijom:

```
void obrisi_listu(cvor *pocetak) {
  while (pocetak)  {
    cvor *p = pocetak;
    pocetak = pocetak->sledeci;
    free(p);
  }
}
```

U navedenom kodu treba obratiti pažnju na redosled operacija u petlji i ulogu pomoćnog pokazivača (`p`). Naime, potrebno je najpre sačuvati pokazivač na tekući element (kako bi mogao biti obrisan).

Oslobađanje povezane liste može se implementirati i rekurzivno (uz opasnosti koje rekurzivna implementacija nosi):

```
void obrisi_listu(cvor *pocetak)
{
  if (pocetak) {
    obrisi_listu(pocetak->sledeci);
    free(pocetak);
  }
}
```

Nakon poziva navedene funkcije, često je neophodno i pokazivaču na početak liste dodeliti vrednost `NULL`. Alternativno, to može da se prepusti sâmoj funkciji za brisanje liste (pri čemu njen parametar onda treba da bude tipa `cvor**`).

Sledeći kôd ilustruje upotrebu navedenih funkcija za rad sa povezanim listama („...“ označava kôd gore navedenih funkcija):

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

/* ... */

int main()
{
  int i;
  cvor *l = NULL, *p;
  for (i = 0; i < 20; i++)  {
    if (i % 2 != 0) {
      if (dodaj_na_pocetak(&l, i) != OK) {
        obrisi_listu(l);
        return -1;
      }
    } else {
```

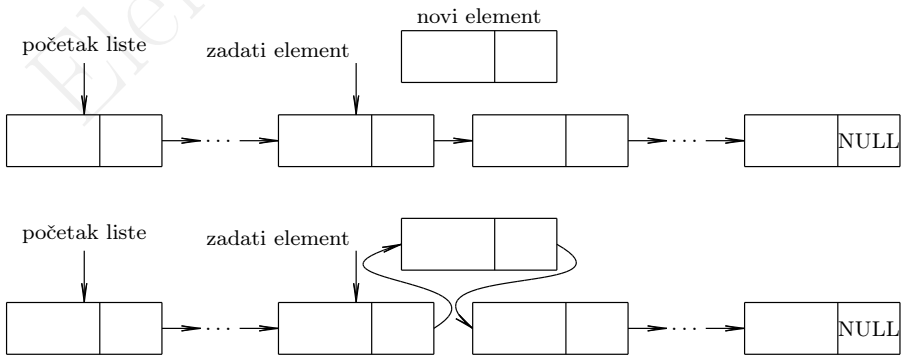
```
      if (dodaj_na_kraj(&l, i) != OK) {
        obrisi_listu(l);
        return -1;
      }
    }
  }

  ispisi_elemente_liste(l);
  obrisi_listu(l);

  return 0;
}
```

Navedeni program daje sledeći izlaz:

```
19 17 15 13 11 9 7 5 3 1 0 2 4 6 8 10 12 14 16 18
```

U povezanu listu moguće je umetnuti element na bilo koje mesto, ali ta operacija nije složenosti $O(1)$ ako je poznat samo početak povezane liste. Ukoliko je poznat pokazivač na element iza kojeg treba umetnuti novi element, onda ta operacija može da se izvrši u vremenu $O(1)$. Na primer, naredna funkcija ubacuje element `novi` iza elementa `element` (za koji se pretpostavlja da nije `NULL`) (slika 6.3).

```
void ubaci_iza_elementa(cvor *element, cvor *novi)
{
  novi->sledeci = element->sledeci;
  element->sledeci = novi;
}
```



Slika 6.3: Ilustracija ubacivanja elementa u povezanu listu iza zadatog elementa

Za ubacivanje elementa *ispred* datog elementa u vremenu $O(1)$ može se upotrebiti sledeći trik (novi se ubacuje iza datog, a onda im se sadržaj razmeni).

```
void ubaci_ispred_elementa(cvor *element, cvor *novi)
{
  novi->sledeci = element->sledeci;
  element->sledeci = novi;
  swap(&(element->podatak), &(novi->podatak));
}
```

Za brisanje elementa u vremenu $O(1)$ potrebno je imati informaciju o elementu koji mu prethodi. Ako treba obrisati poslednji element, čak i ako je poznata adresa poslednjeg elementa, da bi on bio obrisan (i pokazivač na kraj ažuriran), potrebno je obilaskom liste stići do elementa koji mu prethodi, što zahteva vreme $O(n)$ (te ta operacija i ne treba da se koristi kod povezane liste).

Dodajmo da je moguće ubaciti element na njegovo mesto u sortiranoj listi (pod pretpostavkom da je podatak nekog tipa nad kojim je definisan poredak).

```
cvor *ubaci_sortirano(cvor *pocetak, cvor *novi)
{
  if (pocetak == NULL)
    return novi;
  if (novi->podatak <= pocetak->podatak) {
    novi->sledeci = pocetak;
    return novi;
  }
  pocetak->sledeci = ubaci_sortirano(pocetak->sledeci, novi);
  return pocetak;
}
```

Kao što je već rečeno, ovakvo rekurzivno rešenje treba izbegavati, posebno jer se ista funkcionalnost može dobiti i iterativnim pristupom:

```
cvor *ubaci_sortirano(cvor *pocetak, cvor *novi)
{
  cvor *tekuci;
  if (pocetak == NULL || pocetak->podatak >= novi->podatak) {
    novi->sledeci = pocetak;
    return novi;
  }
  tekuci = pocetak;
  while (tekuci->sledeci != NULL &&
         tekuci->sledeci->podatak < novi->podatak)
    tekuci = tekuci->sledeci;
  novi->sledeci = tekuci->sledeci;
```

```
  tekuci->sledeci = novi;
  return pocetak;
}
```

Ovo se može iskoristiti za sortiranje elemenata liste algoritmom sortiranja umetanjem tj. *insertion sort* (umeće se jedan po jedan element u novu listu, sve dok ona ne postane sortirana). Taj pristup, naravno, ne daje zadovoljavajuću složenost sortiranja i za sortiranje treba koristiti druge, efikasnije algoritme (na primer, za sortiranje elemenata liste pogodan je algoritam *merge sort*). Dve sortirane liste mogu se jednostavno objediniti u treću, sortiranu (u ovom scenariju se samo prespajaju postojeći čvorovi, tako da se originalne liste gube nakon učešljavanja).

```
cvor *ucesljaj(cvor *l1, cvor *l2)
{
  if (l1 == NULL)
    return l2;
  if (l2 == NULL)
    return l1;
  if (l1->podatak <= l2->podatak) {
    l1->sledeci = ucesljaj(l1->sledeci, l2);
    return l1;
  } else {
    l2->sledeci = ucesljaj(l1, l2->sledeci);
    return l2;
  }
}
```

I ova funkcija može biti implementirana iterativno, na sledeći način:

```
cvor *ucesljaj(cvor *l1, cvor *l2)
{
  /* pravimo vestacki cvor da izbegnemo analizu slucajeva */
  cvor *pocetak = novi(0);
  cvor *kraj = pocetak;
  while (1) {
    if (l1 == NULL) {
      kraj->sledeci = l2;
      break;
    }
    if (l2 == NULL) {
      kraj->sledeci = l1;
      break;
    }
    if (l1->podatak < l2->podatak) {
```

```
      kraj->sledeci = l1;
      kraj = kraj->sledeci;
      l1 = l1->sledeci;
    } else {
      kraj->sledeci = l2;
      kraj = kraj->sledeci;
      l2 = l2->sledeci;
    }
  }
  /* uklanjamo vestacki cvor */
  return obrisi_sa_pocetka(pocetak);
}
```

Primetimo da je implementacija ove funkcije olakšana time što je na početku liste ubačen jedan veštački čvor, čime je postignuto da rezultujuća lista nikada nije prazna. Zahvaljujući tome prilikom ubacivanja čvorova na kraj liste ne moramo ispitivati da li je kraj liste jednak `NULL`. Na kraju obrade taj veštački čvor se uklanja.

Opisano učešljavanje omogućava i sortiranje liste primenom varijacije algoritma sortiranja objedinjavanjem tj. *merge sort*.

Pitanja i zadaci za vežbu

Pitanje 6.1. *Uporediti jednostruko povezane liste i nizove (statičke i dinamičke). Koje su im sličnosti, a koje razlike? Koje su prednosti, a koje mane ovih struktura podataka?*

Pitanje 6.2. *Ukoliko se znaju pokazivači na prvi i poslednji element jednostruko povezane liste, koja je složenost operacije brisanja prvog a koja složenost operacije brisanja poslednjeg elementa (obrazložiti odgovor)?*

Pitanje 6.3. *Da li je moguće da prototip funkcije koja ubacuje element u jednostruko povezanu listu bude*

```
void ubaci(cvor* pocetak, int vrednost);
```

Obrazložiti odgovor.

Pitanje 6.4. *Opisati algoritam kojim se proverava da li jednostruko povezana lista sadrži ciklus, to jest da li je poslednji element greškom usmeren ka nekom prethodnom elementu. Osmisliti i algoritam kojim se određuje dužina takvog ciklusa.*

Pitanje 6.5. *Opisati algoritam linearne vremenske i konstantne memorijske složenosti kojim se određuje središnji element jednostruko povezane liste.*

Zadatak 6.1. *Definisati rekurzivnu funkciju* `void obrisi(cvor *l)` *koja briše sve elemente jednostruko povezane liste. Šta je glavna mana takve rekurzivne implementacije?*

Zadatak 6.2. *Definisati iterativnu funkciju linearne složenosti koja iz liste celih brojeva izbacuje svaki drugi element (svaki element na neparnoj poziciji, pri čemu se pozicije broje od nule).*

Zadatak 6.3. *Definisati iterativnu funkciju linearne složenosti koja između svaka dva susedna elementa u listi celih brojeva umeće element koji sadrži njihovu razliku.*

Zadatak 6.4. *Definisati iterativnu funkciju koja od dve date liste u kojima se čuvaju celi brojevi, formira novu listu koja koja sadrž alternirajuće raspoređene elemente iz te dve liste (prvi element iz prve, prvi element iz druge, drugi element iz prve, drugi element iz druge itd.), sve dok ima elemenata u* obe *liste. Ne formirati nove čvorove, već samo postojeće čvorove povezati u jednu listu, a kao rezultat vratiti početak te formirane liste. Funkcija treba da bude linearne vremenske složenosti u odnosu na zbir dužina te dve liste.*

Zadatak 6.5. *Definisati iterativnu funkciju linearne složenosti koja na osnovu dve strogo rastući sortirane liste formira treću koja sadrži sve zajedničke elemente te dve liste.*

Zadatak 6.6. *Definisati funkciju koja algoritmom merge sort sortira jednostruko povezanu listu.*

Zadatak 6.7. *Definisati iterativnu funkciju linearne složenosti koja uklanja duplikate iz sortirane jednostruko povezane liste.*

Zadatak 6.8. *Definisati iterativnu funkciju koja sažima datu listu (modifikujući je, ne kreirajući novu listu) tako što izbacuje svaki element koji se više puta pojavljuje u listi. Funkcija može da ima kvadratnu vremensku složenost.*

Zadatak 6.9. *Definisati iterativnu funkciju linearne složenosti koja određuje n-ti element od kraja jednostruko povezane liste.*

Zadatak 6.10. *Definisati iterativnu funkciju linearne složenosti koja proverava da li je jednostruko povezana lista palindrom.*

Zadatak 6.11. *Definisati iterativnu funkciju linearne složenosti koja udvaja svako pojavljivanje datog elementa u datoj jednostruko povezanoj listi. Na primer, za listu* $1 \to 7 \to 6 \to 7 \to 1 \to 4 \to 7$ *i element 7 potrebno je dobiti listu:* $1 \to 7 \to 7 \to 6 \to 7 \to 7 \to 1 \to 4 \to 7 \to 7$.

Zadatak 6.12. *Definiši iterativnu funkciju koja u linearnoj složenosti briše poslednje pojavljivanje datog elementa u jednostruko povezanoj listi.*

Zadatak 6.13. *Definisati iterativnu funkciju linearne vremenske složenosti koja rotira jednostruko povezanu listu za k pozicija udesno (npr. kada se lista* $1 \to 2 \to 3 \to 4 \to 5 \to 6$ *rotira za dva mesta dobija se lista* $3 \to 4 \to 5 \to 6 \to 1 \to 2$*).*

Zadatak 6.14. *Definisati iterativnu funkciju linearne složenosti koja obrće jednostruko povezanu listu (ne praviti nove čvorove, već samo preusmeriti pokazivače).*

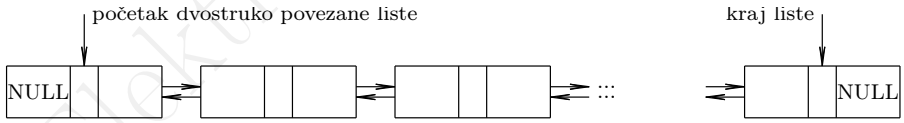
6.2 Dvostruko povezana lista

Svaki element u (jednostruko) povezanoj listi ima jedan pokazivač — pokazivač na sledeći element liste. Zato je listu jednostavno obilaziti u jednom smeru ali je vremenski zahtevno u suprotnom (stoga takav obilazak nikada ne treba vršiti). U dvostruko povezanoj (dvostruko ulančanoj) listi, svaki element sadrži dva pokazivača — jedan na svog prethodnika a drugi na svog sledbenika. Na primer, ako lista sadrži karaktere, čvor liste može biti definisan na sledeći način.

```
typedef struct _cvor {
  char c;
  struct _cvor *prethodni, *sledeci;
} cvor;
```

Dvostruko povezana lista omogućava jednostavno kretanje unapred i unazad kroz listu. U dvostruko povezanoj listi i umetanje i brisanje elementa sa oba kraja liste zahteva vreme $O(1)$. Isto važi i za umetanje novog čvora „u sredinu“ tj. ispred ili iza čvora na koji je poznat pokazivač, kao i za brisanje čvora na koji ukazuje dati pokazivač.

Po analogiji sa jednostruko povezanim listama, često se prethodnik prvog i sledbenik poslednjeg čvora postavljaju na vrednost `NULL`. Dvostruko povezana lista ilustrovana je na slici 6.4.



Slika 6.4: Dvostruko povezana lista

Implementacija dvostruko povezane liste je jednostavnija ako se uvede specijalni čvor (ponekad se naziva „sentinela“) koji istovremeno čuva poziciju početnog i krajnjeg čvora dvostruko povezane liste (njegov pokazivač na sledeći čvor ukazuje na stvarni početak, a pokazivač na prethodni čvor ukazuje na stvarni kraj liste, pri čemu oba pokazuju na sentinelu ako je lista prazna). Vrednost upisana u sentinelu se nikada ne koristi. Ovim se zapravo održava svojevrsna kružna lista.

Postojanje sentinele značajno olakšava implementacije procedura za umetanje i brisanja elemenata liste jer nema potrebe ispitivati specijalne slučajeve

da li je neki pokazivač jednak NULL (što je neophodno ako se koristi reprezentacija u kojoj su prethodnik prvog i sledebenik poslednjeg čvora postavljeni na vrednost NULL).

U nastavku je prikazan sadržaj datotete dvostruko_povezana_lista.h i datoteke dvostruko_povezana_lista.c sa implementacijom osnovnih funkcija za rad sa dvostruko povezanom listom.[1] Primetimo da se u ovoj implementaciji, zbog jednostavnosti, ne opslužuje mogućnost da je dinamička alokacija neuspešna (već se podrazumeva da će ona uvek biti uspešna i to naglašavaju naredbe assert, videti poglavlje 1.10).

```
#ifndef __DVOSTRUKO_POVEZANA_LISTA_H__
#define __DVOSTRUKO_POVEZANA_LISTA_H__

typedef struct _cvor {
  char c;
  struct _cvor *prethodni, *sledeci;
} cvor;

cvor *napravi_cvor(char c, cvor *p, cvor *s);
void obrisi(cvor *cv);
void ubaci_prethodni(cvor *cv, char c);
void ubaci_sledeci(cvor *cv, char c);
void ubaci_na_pocetak(cvor *sentinela, char c);
void ubaci_na_kraj(cvor *sentinela, char c);
int prazna(cvor *sentinela);
void ispisi_listu(cvor *sentinela);
void obrisi_listu(cvor *sentinela);

#endif
```

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "dvostruko_povezana_lista.h"

cvor *napravi_cvor(char c, cvor *p, cvor *s)
{
  cvor *novi = malloc(sizeof(cvor));
  assert(novi != NULL);
  novi->c = c;
  novi->prethodni = p;
```

[1] U datoteci dvostruko_povezana_lista.h koristi se pretprocesorski mehanizam uslovnog prevođenja (videti prvi deo ove knjige, poglavlje 9.2, „Organizacija izvornog programa“).

```
  novi->sledeci = s;
  return novi;
}

/* brise cvor na koji ukazuje pokazivac cv
   (pretpostavljamo da to nije sentinela */
void obrisi(cvor *cv)
{
  cv->prethodni->sledeci = cv->sledeci;
  cv->sledeci->prethodni = cv->prethodni;
  free(cv);
}

/* ubacuje novi cvor sa karakterom c
   tako da prethodi datom cvoru cv */
void ubaci_prethodni(cvor *cv, char c)
{
  cvor *novi = napravi_cvor(c, cv->prethodni, cv);
  assert(novi != NULL);
  novi->prethodni->sledeci = novi;
  novi->sledeci->prethodni = novi;
}

/* ubacuje novi cvor sa karakterom c
   tako da sledi nakon datog cvora cv */
void ubaci_sledeci(cvor *cv, char c)
{
  cvor *novi = napravi_cvor(c, cv, cv->sledeci);
  assert(novi != NULL);
  novi->prethodni->sledeci = novi;
  novi->sledeci->prethodni = novi;
}

/* ubacuje novi cvor na pocetak liste cija je sentinela data */
void ubaci_na_pocetak(cvor *sentinela, char c)
{
  ubaci_sledeci(sentinela, c);
}

/* ubacuje novi cvor na kraj liste cija je sentinela data */
void ubaci_na_kraj(cvor *sentinela, char c)
{
  ubaci_prethodni(sentinela, c);
}
```

```c
/* provera da li je lista prazna */
int prazna(cvor *sentinela)
{
  /* lista je prazna ako i samo ako sadrzi samo sentinelu
     (kojoj oba pokazivaca pokazuju na nju samu) */
  return sentinela->sledeci == sentinela;
}

/* ispisuje celu listu */
void ispisi_listu(cvor *sentinela)
{
  for (cvor *cv = sentinela->sledeci;
       cv != sentinela;
       cv = cv->sledeci)
    putchar(cv->c);
  putchar('\n');
}

/* brise celu listu */
void obrisi_listu(cvor *sentinela)
{
  while (!prazna(sentinela))
    obrisi(sentinela->sledeci);
}
```

6.2.1 Primer primene strukture dvostruko povezana lista: linijski editor

Prikažimo upotrebu dvostruko povezane liste na primeru takozvanog linijskog editora. Editori teksta podrazumevaju mogućnost kretanja kursora kroz neku liniju teksta u oba smera (nalevo i nadesno), umetanje karaktera na tekuću poziciju kursora i brisanje karaktera koji se nalaze levo i desno od kursora (tasterima `backspace` i `delete`). Ako bi se tekuća linija teksta predstavila nizom karaktera, umetanje karaktera na tekuću poziciju kursora zahtevalo bi pomeranje ostalih karaktera u nizu i to bi bila operacija linearne složenosti u odnosu na dužinu linije teksta, što je veoma neefikasno. Stoga je potrebno drugačije predstaviti podatke.

Jedno od mogućih rešenja je da se karakteri u tekućoj liniji čuvaju u povezanoj listi, jer ona omogućava efikasno umetanje i brisanje elemenata iz sredine. Pošto se kursor može kretati u oba smera, jasno je da je potrebno koristiti dvostruko povezanu listu. Iako kursor u editoru uvek ukazuje između dva karaktera, kursor ćemo u implementaciji predstaviti pokazivačem na element liste (karakter) koji se nalazi neposredno desno od stvarne pozicije kursora. Ako kursor ukazuje na sentinelu, smatraćemo da je kursor na kraju teksta, a ako

pokazuje na prvi element liste (element koji sledi nakon sentinele), smatraćemo da je kursor na početku teksta. Operacije se tada izvode na sledeći način:

- Kursor možemo pomeriti ulevo, samo ako mu ne prethodi sentinela, a udesno samo ako ne ukazuje na sentinelu.
- Umetanje karaktera se vrši tako što se novi element ubaci tako da prethodi trenutnoj poziciji kursora, a kursor se ne pomera (ovo radi ispravno i kada je lista prazna tj. kada sadrži samo sentinelu).
- Brisanje karaktera levo od kursora je moguće kad god kursor nije na početku liste tj. kada mu ne prethodi sentinela i vrši se tako što se obriše prethodni element liste, a kursor se ne pomera.
- Brisanje karaktera desno od kursora je moguće kada god kursor nije na kraju liste tj. kada ne ukazuje na sentinelu i vrši se tako što se obriše element na koji kursor ukazuje, a kursor se pomeri na sledeću poziciju.

Vežbe radi pretpostavimo da su sve akcije koje korisnik zadaje predstavljene niskom karaktera (u realnoj implementaciji editora teksta, one bi se izvršavale kao reakcije na događaje).

- `iX` – korisnik je otkucao karakter `X` (*insert* `X`)
- `<` – korisnik je pritisnuo taster levo
- `>` – korisnik je pritisnuo taster desno
- `b` – korisnik je pritisnuo taster `backspace` za brisanje karaktera levo od kursora
- `d` – korisnik je pritisnuo taster `delete` za brisanje karaktera desno od kursora

Na primer, akcije `iAiB<biC>>` označavaju unos karaktera `A`, unos karaktera `B`, pomeranje kursora ulevo, brisanje karaktera `A` tasterom `backspace`, unos karaktera `C` i pomeranje kursora udesno, dva puta. Nakon ovih akcija sadržaj linije je `CB` (kursor označavamo sa `|`, a linija je inicijalno prazna):

naredba	značenje	sadržaj linije
`iA`	kucanje karaktera `A`	`A|`
`iB`	kucanje karaktera `B`	`AB|`
`<`	taster levo	`A|B`
`b`	taster za brisanje karaktera levo od kursora	`|B`
`iC`	kucanje karaktera `C`	`C|B`
`>`	taster desno	`CB|`

Editor kreće od prazne linije, izvršavaju se sve akcije korisnika (zadate niskom) i na kraju je potrebno ispisati tekst koji sadrži ta linija u editoru.

```c
#include <stdio.h>
#include "dvostruko_povezana_lista.h"

void obradi_akcije(char akcije[])
{
  int i;
  /* karaktere linije cuvamo u dvostruko povezanoj listi */

  /* vestacki ubacen "prazan" cvor koji oznacava poziciju
     ispred pocetka tj. iza kraja liste */
  cvor *sentinela = napravi_cvor('_', NULL, NULL);
  sentinela->prethodni = sentinela->sledeci = sentinela;

  /* tekuca pozicija kursora */
  cvor *kursor = sentinela;

  /* obradjujemo redom sve akcije */
  i = 0;
  while (akcije[i] != '\0') {
    char akcija = akcije[i++];
    if (akcija == '<') {
      /* strelica ulevo */
      if (kursor->prethodni != sentinela)
        kursor = kursor->prethodni;
    } else if (akcija == '>') {
      /* strelica udesno */
      if (kursor != sentinela)
        kursor = kursor->sledeci;
    } else if (akcija == 'i') {
      /* umetanje karaktera na mesto kursora*/
      char c = akcije[i++];
      ubaci_prethodni(kursor, c);
    } else if (akcija == 'b') {
      /* brisanje karaktera levo od kursora (backspace) */
      if (kursor->prethodni != sentinela)
        obrisi(kursor->prethodni);
    } else if (akcija == 'd') {
      /* brisanje karaktera desno od kursora (delete) */
      if (kursor != sentinela) {
        cvor *sl = kursor->sledeci;
        obrisi(kursor);
        kursor = sl;
      }
    }
```

```
  }
  /* ispisujemo tekst u liniji */
  ispisi_listu(sentinela);
  /* brisemo listu */
  obrisi_listu(sentinela);
}

int main()
{
  char akcije[] = "iAiB<biC>>";
  obradi_akcije(akcije);
  return 0;
}
```

Navedeni program ispisuje:

```
CB
```

Pitanja i zadaci za vežbu

Pitanje 6.6. *Uporediti jednostruko i dvostruko povezane liste. Koje su prednosti i mane jedne u odnosu na drugu vrstu lista?*

Pitanje 6.7. *Ako su poznati pokazivači na prvi i na poslednji element, koja je složenost operacije brisanja poslednjeg elementa jednostruko povezane, a koja dvostruko povezane liste?*

Pitanje 6.8. *Ako su poznati pokazivači na prvi i na poslednji element, koja je složenost operacije određivanja n-tog elementa jednostruko povezane, a koja dvostruko povezane liste? ?*

Pitanje 6.9.
(a) *Šta su sentinele i za šta se one koriste?*
(b) *Definisati funkciju koja briše tekući čvor dvostruko povezane liste kada se koristi sentinela i kada se ne koristi sentinela.*
(c) *Definisati funkciju koja umeće novi čvor dvostruko povezane liste iza datog kada se koristi sentinela i kada se ne koristi sentinela.*
(d) *Definisati funkciju koja umeće novi čvor dvostruko povezane liste ispred datog kada se koristi sentinela i kada se ne koristi sentinela.*

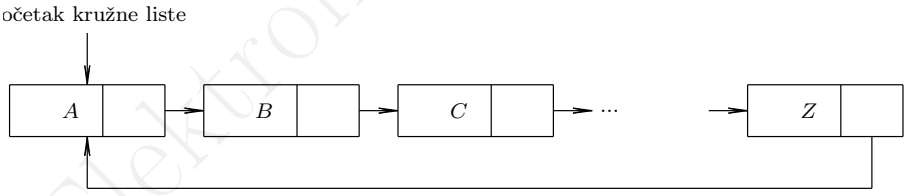
Zadatak 6.15. *Implementirati strukturu podataka u kojoj se čuvaju brojevi i nad kojom se u konstantnoj složenosti mogu izvršiti sledeće operacije:*
(a) *čitanje vrednosti najmanjeg elementa;* (b) *brisanje najmanjeg elementa;* (c) *čitanje vrednosti najvećeg elementa;* (d) *brisanje najvećeg elementa;* (e) *dodavanje novog elementa za koji se uvek zna da je ili manji ili veći od svih ranije ubačenih.*

Zadatak 6.16. *Muzička aplikacija omogućava reprodukovanje pesama. U aplikaciji se održava lista pesama. Na displeju se uvek prikazuje broj pesama u listi. Nove pesme je moguće dodavati na kraj liste. Moguće je isprazniti celu listu. U svakom trenutku je poznata tekuća pesma (ona koja se u tom trenutku pušta). Korisnik može da pređe na sledeću pesmu ili da se vrati na prethodnu pesmu, a može i da pusti listu iz početka. Implementirati strukturu podataka koja omogućava izvršavanje ovih operacija u konstantnom vremenu. Napisati i program kojim se testira rad te strukture (program učitava spisak operacija i izvršava ih jednu po jednu).*

6.3 Kružna lista

Često se javlja potreba da se kroz određene podatke prolazi „u krug", tj. da se nakon analize poslednjeg podatka vratimo ponovo na prvi. Ukoliko su podaci smešteni u niz, nakon što indeks dostigne dužinu niza, vrednost mu možemo ponovo postaviti na nulu (alternativno, uvećavanje za 1 možemo vršiti po modulu dužine niza). Međutim, ako se tokom kružnog obilaska javlja potreba da se elementi dodaju (na tekuću poziciju) ili uklanjaju (sa tekuće pozicije), tada niz očigledno nije dobar izbor i potrebno je koristiti drugačije strukture podataka.

U kružnoj (cikličnoj, cirkularnoj) listi, poslednji element liste ukazuje na prvi element liste. To omogućava da se do bilo kog elementa može doći pošavši od bilo kog drugog elementa. Zbog toga, izbor „prvog" i „poslednjeg" elementa je u kružnoj listi relativan. Kružna lista ilustrovana je na slici 6.5.



Slika 6.5: Kružna lista

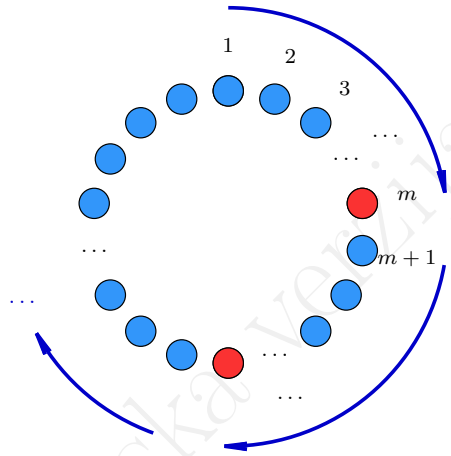
Lista koja je prikazana na slici je jednostruko povezana kružna lista i njen obilazak je moguće vršiti samo u jednom smeru. Naravno, moguće je definisati i dvostruko povezanu kružnu listu.

6.3.1 Primer primene strukture kružna lista: plesači

Ilustrujmo korišćenje jednostruko povezane kružne liste kroz naredni zadatak u kojem se prirodno nameće potreba prolaska kroz elemente određene kolekcije podataka „u krug", pri čemu se tokom obilaska određeni elementi brišu

iz kolekcije, sa tekuće pozicije (ništa se suštinski ne bi promenilo i da se tokom obilaska neki elementi dodaju na tekuću poziciju).

Grupa od n plesača (na čijim kostimima su u smeru kazaljke na satu redom brojevi od 1 do n) izvodi svoju plesnu tačku tako što formiraju krug iz kojeg najpre izlazi m-ti plesač (odbrojava se počev od plesača označenog brojem 1 u smeru kretanja kazaljke na satu). Preostali plesači obrazuju manji krug iz kojeg opet izlazi m-ti plesač (odbrojava se počev od prvog sledećeg za prethodno izbačenog, opet u smeru kazaljke na satu). Izlasci iz kruga nastavljaju se sve dok svi plesači ne budu isključeni.



Zadatak je napisati program koji će odrediti redni broj poslednjeg plesača koji ostaje u krugu. Celi brojevi n, m učitavaju se sa standardnog ulaza. Na primer, za unete vrednosti $n = 5$ i $m = 3$, redosled izlazaka je 3 1 5 2 4, pa je poslednji plesač onaj sa rednim brojem 4.

Nema nikakve značajne razlike između definicije čvora i osnovnih funkcija za rad sa jednostruko povezanom i sa kružnom listom. Stoga ćemo koristiti definicije tipova i neznatno izmenjene funkcije za rad sa jednostruko povezanim listama (navedene u nastavku).

Da bismo malo olakšali implementaciju, pretpostavićemo da pomoćne funkcije rade sa nepraznom listom (prvi čvor ćemo zasebno ubaciti u glavnom delu programa). Na primer, pretpostavićemo da je preduslov funkcije za dodavanje čvora na kraj date liste to da je lista neprazna. S obzirom na to da ta funkcija ne radi ispravno kada je lista prazna, ona se ne može koristiti u situacijama kad ne znamo da li je lista u koju se čvor dodaje prazna. Proširivanje funkcije tako da ispravno pokrije i taj slučaj bi sa jedne strane dovelo do koda koji je upotrebljiv i van konteksta ovog programa, ali bi sa druge strane implementacija bila komplikovanija i donekle neefikasnija (jer bi se u svakom koraku analiziralo da li je lista prazna ili ne). Primetimo da se ovde, dakle, pravi kompromis između nekoliko poželjnih osobina programa. Pri tom je uvek jako važno u komentarima naglasiti sve očekivane preduslove svake funkcije i funkciju pozivati samo

kada su svi njeni preduslovi ispunjeni. U implementaciji koja sledi, ti preduslovi su dodatno eksplicirani korišćenjem funkcije `assert`, koja u debug verziji izaziva grešku tokom izvršavanja programa ako logički uslov koji joj je zadat nije ispunjen, dok se u produkcionoj (*release*) verziji provera uslova potpuno preskače (ukoliko je definisana pretprocesorska direktiva `NDEBUG`, sve `assert` provere se preskaču).

```
typedef struct _cvor {
  int x;
  struct _cvor *sledeci;
} cvor;

/* kreira novi cvor sa datom vrednoscu i
   pokazivacem na sledeci element */
cvor *napravi_cvor(int x, cvor *sledeci)
{
  cvor *cvor = malloc(sizeof(cvor));
  if (cvor == NULL)
    return NULL;
  cvor->x = x;
  cvor->sledeci = sledeci;
  return cvor;
}

/* dodaje novi cvor na kraj neprazne liste kojoj je poznat
   pokazivac na kraj i vraca pokazivac na novoubaceni cvor
   (on postaje novi kraj liste) */
cvor *dodaj_na_kraj(cvor *kraj, int x)
{
  assert(kraj != NULL);
  kraj->sledeci = napravi_cvor(x, NULL);
  return kraj->sledeci;
}

/* brise se cvor na koji ukazuje pokazivac 'tekuci' (koji je
   različit od NULL) i vraca se pokazivac na naredni cvor */
cvor *obrisi_tekuci(cvor *tekuci)
{
  assert(tekuci != NULL);
  /* brisemo zapravo naredni cvor, a vrednost iz njega
     upisujemo u tekuci */
  cvor *tmp;
  tekuci->x = tekuci->sledeci->x;
  tmp = tekuci->sledeci;
```

```
  tekuci->sledeci = tekuci->sledeci->sledeci;
  free(tmp);
  return tekuci;
}
```

Sa ovim funkcijama na raspolaganju možemo jednostavno formirati kružnu listu, a zatim u svakom od $n-1$ koraka spoljne petlje možemo tekući pokazivač pomeriti za $m-1$ koraka (pošto je lista kružna, nakon što dođe na kraj liste, pokazivač će se automatski pomeriti na početak) i zatim ukloniti čvor na koji taj tekući pokazivač pokazuje.

```
int preostali_plesac(int n, int m)
{
  cvor *pocetak, *kraj, *tekuci;

  /* formiramo kruznu listu sa elementima 1, ..., n */
  /* prvi cvor formiramo zasebno */
  pocetak = napravi_cvor(1, NULL);
  if (pocetak == NULL)
    return -1;

  kraj = pocetak;
  /* narednih n-1 cvorova dodajemo redom */
  for (int i = 2; i <= n; i++) {
    kraj = dodaj_na_kraj(kraj, i);
    if (kraj == NULL)
      return -1;
  }
  kraj->sledeci = pocetak; /* spajamo kraj i pocetak */

  /* brojanje krece od pocetnog cvora */
  tekuci = pocetak;
  /* dok ne preostane jedan plesac */
  for (int k = n; k > 1; k--) {
    /* pomeramo se u krug na m-ti cvor od tekuceg */
    for (int i = 0; i < m-1; i++)
      tekuci = tekuci->sledeci;
    /* brisemo tekuci cvor i pokazivac pomeramo na sledeci */
    tekuci = obrisi_tekuci(tekuci);
  }
  /* vracamo preostali element */
  return tekuci->x;
}
```

Pitanja i zadaci za vežbu

Pitanje 6.10. *Kružne liste mogu biti i jednostruko i dvostruko povezane. Koje su prednosti i mane ovih vrsta kružnih lista? Opisati primere za koje je dobar izbor strukture podataka jednostruko povezana kružna lista i primere za koje je dobar izbor strukutre podataka dvostruko povezana kružna lista.*

Pitanje 6.11. *Ako su poznati pokazivači na prvi i poslednji element jednostruko povezane kružne liste, koja je složenost brisanja poslednjeg elementa? A ako je lista dvostruko povezana? Obrazložiti.*

Zadatak 6.17. *Definisati funkciju kojim se kružna lista obilazi k puta i ispisuje se svaki njen element (funkcija prima pokazivač na početni element jednostruko povezane kružne liste).*

Zadatak 6.18. *Operativni sistem implementira deljenje vremena tako što naizmenično aktivira jednu po jednu pokrenutu aplikaciju. Tekuća aplikacija u nekom trenutku može biti prekinuta, a moguće je da se pokrene i nova aplikacija, koja će se aktivirati neposredno nakon tekuće. Ako se pokrenute aplikacije čuvaju u kružnoj listi, napiši program koji simulira rad ovakvog sistema (implementirati funkcije za prelazak na naredni element kružne liste, dodavanje elementa ispred tekućeg i brisanje tekućeg elementa).*

6.4 Stek

Stek (engl. stack) je struktura koja funkcioniše na principu LIFO (*last in, first out*). Kao stek ponaša se, na primer, štap na koji su naređani kompakt diskovi. Ako sa štapa može da se uklanja samo po jedan disk, da bi bio skinut disk koji je na dnu, potrebno je pre njega skinuti sve druge diskove. Element steka može da sadrži više članova različitih tipova. Stek ima sledeće dve osnovne operacije (koje treba da budu složenosti $O(1)$):

push: dodaje element sa zadatim podatkom na vrh steka (čita se „puš“);

pop: uklanja element sa vrha steka (čita se „pop“).

Implementacija steka obično se dizajnira tako da korisniku pruža samo ključne operacije i da jedna implementacija steka može lako da se zameni drugom (kao što će biti ilustrovano u nastavku). Pored dve osnovne operacije, `push` i `pop`, obično su raspoložive i funkcije za inicijalizaciju steka, za brisanje steka i za proveru da li je stek prazan.

Struktura stek pogodna je za mnoge primene, od kojih su neke opisane u daljem tekstu.

6.4.1 Implementacija steka korišćenjem niza

Stek se može implementirati na različite načine, jedan od njih je korišćenjem dinamičkih nizova (moguća je i implementacija koja koristi statički niz, ali ona je manje fleksibilna jer se ne može birati niti menjati veličina steka). Pogodno je sve što je potrebno za stek upakovati u novu strukturu podataka koja se onda može elegenantno koristiti. Funkcije za rad sa stekom će kao parametar imati i stek sa kojim rade (tj. samo pokazivač na njega, efikasnosti radi). Zaglavlje `stek_niz.h` može da izgleda ovako:

```
#ifndef __STEK_NIZ_H__
#define __STEK_NIZ_H__

typedef enum {
   OK,
   NEDOVOLJNO_MEMORIJE,
   NEMA_ELEMENATA
} status;

typedef struct {
  int *podaci;
  int velicina;
  int vrh;
} Stek;

status inicijalizuj_stek(Stek* stek);
status push(Stek* stek, int podatak);
status pop(Stek* stek, int *podatak);
size_t broj_elemenata(const Stek* stek);
int stek_je_prazan(const Stek* stek);
void obrisi_stek(Stek* stek);

#endif
```

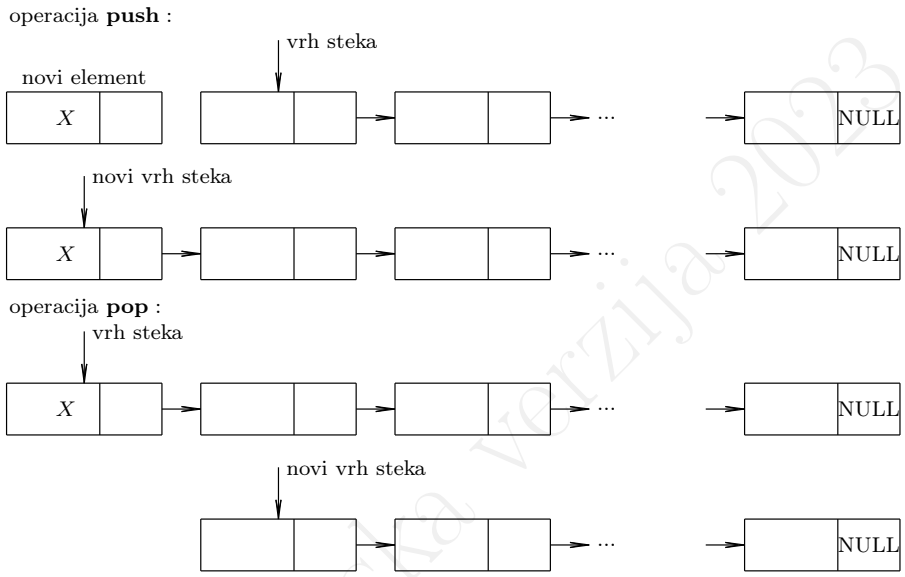
Datoteka `stek_niz.c` može da izgleda ovako:

```
#include <stdlib.h>
#include "stek_niz.h"

const int INICIJALNA_VELICINA_STEKA = 100;

status inicijalizuj_stek(Stek* stek)
{
  stek->podaci = malloc(INICIJALNA_VELICINA_STEKA*sizeof(int));
  if (stek->podaci == NULL)
```

```
    return NEDOVOLJNO_MEMORIJE;
  stek->vrh = 0;
  stek->velicina = INICIJALNA_VELICINA_STEKA;
  return OK;
}

status push(Stek* stek, int podatak)
{
  if (stek->vrh == stek->velicina-1) {
    int *p = realloc(stek->podaci, 2*stek->velicina*sizeof(int));
    if (p == NULL)
      return NEDOVOLJNO_MEMORIJE;
    stek->velicina *= 2;
    stek->podaci = p;
  }
  stek->podaci[stek->vrh++] = podatak;
  return OK;
}

status pop(Stek* stek, int *podatak)
{
  if (stek->vrh == 0)
    return NEMA_ELEMENATA;
  *podatak = stek->podaci[--stek->vrh];
  return OK;
}

size_t broj_elemenata(const Stek* stek)
{
  return stek->vrh;
}

int stek_je_prazan(const Stek* stek)
{
  return (stek->vrh == 0);
}

void obrisi_stek(Stek* stek)
{
  free(stek->podaci);
  stek->velicina = 0;
  stek->vrh = 0;
}
```

6.4.2 Implementacija steka korišćenjem povezane liste

Stek se može implementirati i korišćenjem povezanih lista. Steku se tada pristupa preko pokazivača na njegov vrh, tj. na početni element liste, koji mora da se održava. Osnovne operacije nad stekom implementiranim kao povezana lista ilustrovane su na slici 6.6.



Slika 6.6: Ilustracija osnovnih operacija nad stekom implementiranim korišćenjem povezane liste

Funkcije za rad sa stekom mogu se implementirati koristeći opšte funkcije za rad sa povezanim listama. Datoteka `stek_lista.h` može da izgleda ovako:

```
#ifndef __STEK_LISTA_H__
#define __STEK_LISTA_H__

typedef enum {
   OK,
   NEDOVOLJNO_MEMORIJE,
   NEMA_ELEMENATA
} status;

typedef struct cvor {
  int podatak;
  struct cvor *sledeci;
```

```
} cvor;

typedef struct {
  cvor *vrh;
  size_t broj_el;
} Stek;

status inicijalizuj_stek(Stek* stek);
status push(Stek* stek, int podatak);
status pop(Stek* stek, int *podatak);
size_t broj_elemenata(const Stek* stek);
int stek_je_prazan(const Stek* stek);
void obrisi_stek(Stek* stek);

#endif
```

Datoteka sa implementacijama funkcija `stek_lista.c` može da izgleda ovako:

```
#include <stdlib.h>
#include "stek_lista.h"

static cvor *novi_cvor(int podatak);
static status dodaj_na_pocetak(cvor **pokazivac_na_pocetak,
                               int podatak);
static status obrisi_sa_pocetka(cvor **pokazivac_na_pocetak,
                                int *podatak);

cvor *novi_cvor(int podatak)
{
  cvor *novi = malloc(sizeof(cvor));
  if (novi == NULL)
    return NULL;
  novi->podatak = podatak;
  return novi;
}

status dodaj_na_pocetak(cvor **pokazivac_na_pocetak,
                        int podatak)
{
  cvor *novi = novi_cvor(podatak);
  if (novi == NULL)
    return NEDOVOLJNO_MEMORIJE;
   novi->sledeci = *pokazivac_na_pocetak;
   *pokazivac_na_pocetak = novi;
```

```
  return OK;
}

status obrisi_sa_pocetka(cvor **pokazivac_na_pocetak,
                         int *podatak)
{
  cvor *p;
  if (*pokazivac_na_pocetak == NULL)
    return NEMA_ELEMENATA;

  p = *pokazivac_na_pocetak;
  *podatak = (*pokazivac_na_pocetak)->podatak;
  *pokazivac_na_pocetak = (*pokazivac_na_pocetak)->sledeci;
  free(p);
  return OK;
}

status inicijalizuj_stek(Stek* stek)
{
  stek->vrh = NULL;
  stek->broj_el = 0;
  return OK;
}

status push(Stek* stek, int podatak)
{
  status s = dodaj_na_pocetak(&(stek->vrh), podatak);
  if (s == OK)
    stek->broj_el++;
  return s;
}

status pop(Stek* stek, int *podatak)
{
  status s = obrisi_sa_pocetka(&(stek->vrh), podatak);
  if (s == OK)
    stek->broj_el--;
  return s;
}

size_t broj_elemanata(const Stek* stek)
{
  return stek->broj_el;
}
```

```
int stek_je_prazan(const Stek* stek)
{
  return (stek->vrh == NULL);
}

void obrisi_stek(Stek* stek)
{
  int i;
  while (pop(stek, &i) == OK);
}
```

Primetimo da smo u ovom primeru ponovo naveli implementaciju funkcija za rad sa listama. Alternativa je da se kreira biblioteka funkcija za rad sa listama i da se ta biblioteka upotrebi prilikom implementacije steka (ali i u drugim situacijama u kojima se koriste liste).

Primetimo da nekoliko funkcija koje su definisane u datoteci `stek_lista.c` nisu deklarisane u datoteci `stek_lista.h`, već u datoteci `stek_lista.c` i proglašene su `static` funkcijama. Time se obezbeđuje i daje do znanja da one treba da se koriste samo interno, u implementaciji steka.

Implementacija steka na opisani način, korišćenjem povezane liste, može da dâ određenu fleksibilnost (jer ne zahteva da je memorija koju stek zauzima povezana), ali je veoma neefikasna jer koristi dinamičku alokaciju za dodavanje svakog pojedinačnog elementa.

6.4.3 Primer primene strukture stek: izračunavanje vrednosti izraza

Razmotrimo, kao primer, izračunavanje vrednosti izraza zapisanih u *postfiksnoj* ili *obratnoj poljskoj notaciji*. U postfiksnoj notaciji, binarni operatori se ne zapisuju između operanada, nego iza njih. Na primer, izraz

$3 \cdot ((1 + 2) \cdot 3 + 5)$

se zapisuje na sledeći način:

$(3\ (((1\ 2\ +)\ 3\ \cdot)\ 5\ +)\ \cdot)$

Interesantno je da zagrade u postfiksnom zapisu uopšte ne moraju da se pišu i nema opasnosti od višesmislenog tumačenja zapisa. Dakle, navedeni izraz može se napisati i na sledeći način:

$3\ 1\ 2\ +\ 3\ \cdot\ 5\ +\ \cdot$

Vrednost navedenog izraza može se jednostavno izračunati čitanjem sleva nadesno i korišćenjem steka. Ako je pročitan broj, on se stavlja na stek. Inače (ako je pročitan znak operacije), onda se dva broja skidaju sa steka, na njih se primenjuje pročitana operacija i rezultat se stavlja na stek. Nakon čitanja čitavog izraza (ako je ispravno zapisan), na steku će biti samo jedan element i to broj koji je vrednost izraza.

Sledeći program koristi funkcije navedene datoteke zaglavlja i implementira opisani postupak za izračunavanje vrednosti izraza zapisanog u postfiksnom zapisu. Program čita aritmetički izraz (zapisan u postfiksnom zapisu) kao argument komandne linije. Jedine dozvoljene operacije su `+` i `*`. Jednostavnosti radi, podrazumeva se da su sve konstante u izrazu jednocifreni brojevi. Između cifara i znakova operacija mogu se koristiti razmaci, pa se izraz kao argument komande linije navodi u vidu niske, između navodnika.

Kada naiđe na znak operacije, program sa steka čita dve vrednosti i zamenjuje ih rezultatom primene pročitane operacije. Pretposlednji poziv funkcije `pop` očekuje da na steku pronađe finalni rezultat i da isprazni stek. Poslednji poziv funkcije `pop` onda proverava se da li na steku ima još elemenata – ukoliko ih ima, to znači da početni, zadati izraz nije ispravno zapisan. Ukoliko je zadati izraz ispravno zapisan, na kraju izvršavanja programa ispisuje se vrednost izraza. Kôd je dat u vidu datoteke `izracunaj.c`.

```c
#include <stdio.h>
#include <ctype.h>
/* #include "stek_niz.h" */
#include "stek_lista.h"

int main(int argc, char **argv)
{
  int i=0, podatak, podatak1, podatak2;
  char c;
  Stek stek;
  if (inicijalizuj_stek(&stek) != OK)
    return -1;

  if (argc != 2)
    return -1;

  while ((c = argv[1][i++]) != '\0') {
    if (isdigit(c)) {
      if (push(&stek, c-'0') == NEDOVOLJNO_MEMORIJE) {
        printf("Ne moze se kreirati element steka!\n");
        obrisi_stek(&stek);
        return -1;
      }
    }
    else if (c == '+' || c == '*') {
      if (pop(&stek, &podatak1) != OK) {
        printf("Neispravan izraz!\n");
        obrisi_stek(&stek);
        return -1;
```

```
      }
      if (pop(&stek, &podatak2) != OK) {
        printf("Neispravan izraz!\n");
        obrisi_stek(&stek);
        return -1;
      }

      switch(c) {
        case '+': podatak = podatak1 + podatak2; break;
        case '*': podatak = podatak1 * podatak2; break;
        default:  break;
      }

      if (push(&stek, podatak) == NEDOVOLJNO_MEMORIJE) {
        printf("Ne moze se kreirati element steka!\n");
        obrisi_stek(&stek);
        return -1;
      }
    }
    else if (c != ' ') {
      printf("Neispravan izraz!\n");
      obrisi_stek(&stek);
      return -1;
    }
  }
  if (pop(&stek, &podatak) != OK) {
    printf("Neispravan izraz!\n");
    obrisi_stek(&stek);
    return -1;
  }
  if (pop(&stek, &podatak1) == OK) {
    printf("Neispravan izraz!\n");
    obrisi_stek(&stek);
    return -1;
  }
  printf("Vrednost izraza %s je %i\n", argv[1], podatak);
  return 0;
}
```

Program može da koristi implementaciju steka bilo iz poglavlja 6.4.1 ili bilo iz poglavlja 6.4.2, sa jedinom izmenom u redu sa odgovarajućom direktivnom `#include` (bira se `stek_niz.h` ili `stek_lista.h`). To pokazuje da struktura stek može i treba da se implementira tako da je moguće njeno istovetno korišćenje bez obzira na detalje implementacije. Navedeni program može da se kompilira na sledeći način:

```
gcc izracunaj.c stek_lista.c -o izracunaj_vrednost_izraza
```

Ako se program pokrene sa:

```
./izracunaj_vrednost_izraza "3 1 2 + 3 * 5 + *"
```

dobija se izlaz:

```
Vrednost izraza 3 1 2 + 3 * 5 + * je 42
```

Ako se program pokrene sa:

```
./izracunaj_vrednost_izraza "3 1 2 +"
```

dobija se izlaz:

```
Neispravan izraz!
```

6.4.4 Generički stek

Kôd naveden u poglavljima 6.4.1 i 6.4.2 ilustruje kako se može implementirati stek čiji elementi sadrže podatak tipa `int`. Naravno, često je potrebno koristiti stek čiji elementi sadrže podatak nekog drugog konkretnog tipa `<type>`. To se može postići na nekoliko načina. Kompletan kôd za stek koji je naveden može se modifikovatti tako da elementi steka čuvaju podatak tipa `<type>` a ne podatak tipa `int`. Elegantniji način da se to postigne je da se u definiciji niza odnosno strukture ne koristi ni `int` ni `<type>`, već novo ime tipa (na primer, `Data`) koje se lako može promeniti sa `typedef` i time stek prilagoditi željenom tipu izmenom u samo jednom redu. Za implementaciju steka korišćenjem povezane liste, relevantni kôd mogao bi da izgleda ovako:

```
typedef <type> Data;

typedef struct cvor {
  Data podatak;
  struct cvor *sledeci;
} cvor;
```

Za implementaciju steka korišćenjem dinamičkog niza, opisani pristup može se, objedinjeno, realizovati u vidu neke specifične datoteke `podatak.h`, na primer:

```
#ifndef PODATAK_H
#define PODATAK_H
```

```
typedef int Data;

#endif
```

naredne datoteke `stek_generic.h`:

```
#ifndef __STEK_GENERICKI_H__
#define __STEK_GENERICKI_H__

#include "podatak.h"

typedef enum {
  OK,
  NEDOVOLJNO_MEMORIJE,
  NEMA_ELEMENATA
} status;

typedef struct {
  Data* podaci;
  int velicina;
  int vrh;
} Stek;

status push(Stek* stek, Data podatak);
status pop(Stek* stek, Data* podatak);
status inicijalizuj_stek(Stek* stek);
size_t broj_elemenata(const Stek* stek);
void obrisi_stek(Stek* stek);
int stek_je_prazan(const Stek* stek);

#endif
```

i naredne datoteke `stek_generic.c`:

```
#include <stdlib.h>
#include "stek_generic.h"

const int INICIJALNA_VELICINA_STEKA=1;

status push(Stek* stek, Data podatak)
{
  if (stek->vrh == stek->velicina-1) {
    stek->velicina *= 2;
    Data *p = realloc(stek->podaci,stek->velicina*sizeof(Data));
    if (p == NULL)
```

```
      return NEDOVOLJNO_MEMORIJE;
    stek->podaci = p;
  }
  stek->podaci[stek->vrh++] = podatak;
  return OK;
}

status pop(Stek* stek, Data* podatak)
{
  if (stek->vrh == 0)
    return NEMA_ELEMENATA;
  *podatak = stek->podaci[--stek->vrh];
  return OK;
}

status inicijalizuj_stek(Stek* stek)
{
   stek->podaci = malloc(INICIJALNA_VELICINA_STEKA*sizeof(Data));
   if (stek->podaci == NULL)
     return NEDOVOLJNO_MEMORIJE;
   stek->velicina = INICIJALNA_VELICINA_STEKA;
  return OK;
}

int stek_je_prazan(const Stek* stek)
{
  return (stek->vrh == 0);
}

size_t broj_elemenata(const Stek* stek)
{
  return stek->vrh;
}

void obrisi_stek(Stek* stek)
{
  free(stek->podaci);
  stek->velicina = 0;
  stek->vrh = 0;
}
```

Ako je podatak tipa `Data` velik, tada je bolje preneti ga funkciji `push` preko konstantnog pokazivača (drugi parametar tada bi bio `const Data* podatak`), čime se smanjuje obim kopiranja do kojeg dolazi zbog prenosa po vrednosti.

Opisani pristup može se još unaprediti, tako da nije potrebna ni izmena

kojom se navodi zadati tip. Naime, podatak koji elementi steka čuvaju može da bude tipa `void*` i, slično kao u poglavlju 5.1, time stek postaje generička struktura koja može da ima elemente bilo kod tipa:

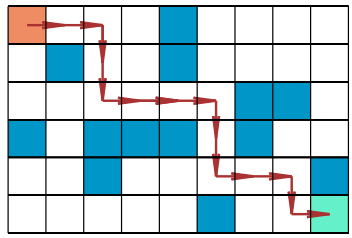
```
typedef void* Data;

typedef struct cvor {
  Data podatak;
  struct cvor *sledeci;
} cvor;
```

U ovakvom pristupu, naravno, nije neophodno koristiti novo ime tipa, već sve funkcije mogu da neposredno koriste tip `void*`. Primetimo da u ovom pristupu elementi steka samo pokazuju na postojeće elemente nekog konkretnog tipa i njih niti kreiraju niti uništavaju. Dodatno, funkcije koje koriste ovakav, generički stek, moraju da se staraju o tome da pokazivače koji pripadaju elementima steka kastuju na adekvatan način pre dereferenciranja. Drugim rečima, podrazumeva se da funkcije koje koriste stek znaju tip podatka koji se čuva u elementima steka.

Na analogne načine mogu se implementirati i generičke liste i druge slične strukture.

6.4.5 Primer primene strukture stek: pretraga u dubinu i ispitivanje da li postoji put na mapi

Razmotrimo ponovo problem ispitivanje da li postoji put na mapi (koji je opisan u poglavlju 5.5.1).



Ovaj problem može biti rešen i korišćenjem steka. Elementi steka će biti polja mape, reprezentovana sa dva broja: `v` i `k` i opisana novim tipom `Data`:

```
typedef struct data {
  int v,k;
} Data;
```

Biće potrebno da se ova definicija uključi u datoteku `stek_genericki.h`. Time će, na način opisan u poglavlju 6.4.4, biti omogućeno korišćenje steka čiji su elementi tipa `Data`.

U prikazanom rešenju, funkcija `postojiPut` kreće od polaznog polja (polja $(0, 0)$), stavlja ga na vrh steka i onda, dok god ih ima, obrađuje jedno po jedno polje sa vrha steka. Ukoliko je to polje ciljno – funkcija vraća vrednost `1`, što znači da postoji traženi put. Inače, ukoliko to polje nije ciljno, na vrh steka stavljaju se sva njegova susedna polja koja nisu prepreke i nisu već posećena (što se pamti u nizu `poseceno`). Ukoliko je stek prazan a nije se stiglo do ciljnog polja, to znači da traženi put ne postoji i funkcija vraća vrednost `0`. Koordinate susednih polja određuju se jednostavno, koristeći pomoćni niz `smer` koji sadrži pomeraje za četiri moguća smera. Prilikom analiziranja susednih polja potrebno je proveriti i da li se ona nalaze na mapi.

```
#include <stdio.h>
#include "stek_genericki.h"

#define MAX_N 50
const int smer[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

int postojiPut(int prepreke[MAX_N][MAX_N], int m, int n)
{
  int poseceno[MAX_N][MAX_N] = {0};
  Stek stek;
  inicijalizuj_stek(&stek);

  poseceno[0][0] = 1;
  Data polje = { 0, 0 }; /* polazno polje */
  push(&stek, polje);

  while (!stek_je_prazan(&stek)) {
    pop(&stek, &polje);
    /* da li je 'polje' ciljno polje? */
    if (polje.v == m-1 && polje.k == n-1) {
      obrisi_stek(&stek);
      return 1;
    }
    for (int i = 0; i < 4; i++) {
      int v1 = polje.v + smer[i][0], k1 = polje.k + smer[i][1];
      if (0 <= v1 && v1 < m && 0 <= k1 && k1 < n
          && !poseceno[v1][k1] && !prepreke[v1][k1]) {
        Data polje1 = { v1, k1 };
        push(&stek, polje1);
        poseceno[v1][k1] = 1;
```

```
      }
    }
  }
  obrisi_stek(&stek);
  return 0;
}

int main()
{
  int m, n;
  char c;
  /* ucitavamo dimenzije matrice prepreka */
  if (scanf("%d %d\n", &m, &n) != 2)
    return -1;
  int prepreke[MAX_N][MAX_N];
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      c = getchar();
      prepreke[i][j] = (c == '1'); /* '1' oznacava prepreku */
    }
    c = getchar(); /* preskacemo '\n' */
  }
  if (postojiPut(prepreke, m, n))
    printf("\nPostoji put\n");
  else
    printf("\nNe postoji put\n");
  return 0;
}
```

Program za navedene ulazne podatke:

```
3 3
000
001
100
```

daje sledeći izlaz:

```
Postoji put
```

Program za navedene ulazne podatke:

```
3 3
000
011
```

```
010
```

daje sledeći izlaz:

```
Ne postoji put
```

Opisano rešenje predstavlja sistematičan proces pretrage i ukoliko put između polaznog i ciljnog polja postoji, onda će navedena funkcija to sigurno da utvrdi. Poredak obilaska polja, koji u ovom rešenju proističe iz korišćenja steka, zove se „obilazak u dubinu“. Ovako pronađeni put između dva polja nije nužno najkraći mogući.

Pitanja i zadaci za vežbu

Pitanje 6.12. *Koja struktura podataka može da modeluje lift (iz lifta ljudi obično izlaze u obratnom redosledu od redosleda ulaska)?*

Pitanje 6.13. *Da li struktura stek ispunjava uslov*
(a) *LIFO (*last in, first out*);* (b) *FILO (*first in, last out*);* (c) *FIFO (*first in, first out*);* (d) *LILO (*last in, last out*)?*

Pitanje 6.14. *Koje su osnovne operacije steka i koja je njihova složenost u implementaciji baziranoj na listama?*

Pitanje 6.15. *Ako su* `s1` *i* `s2` *prazni stekovi, šta je njihova vrednost nakon operacija* `push(s1, 1)`*,* `push(s2, 2)`*,* `push(s1, 3)`*,* `push(s2, pop(s1))`*?*

Pitanje 6.16. *Operacije dodavanja elementa na vrh steka su konstantne složenosti i kada se koristi niz i kada se koristi lista. Koja od ovih implementacija je brža (za konstantni faktor) i zašto?*

Pitanje 6.17. *Svaki program koji koristi stek može da se transformiše tako da koristi, na primer, dinamički alociran niz. Za razliku od steka, niz pruža i efikasnu operaciju čitanja elementa koji se nalazi na datoj poziciji (a ne samo onog na vrhu), obilazak elemenata u oba smera i slično. Slično, direktno korišćenje lista omogućava sve operacije koje se mogu izvršiti nad stekom, ali i neke dodatne, koje stek ne podržava. Opisati zašto bi programeri u nekim programima koristili specifičnu strukturu steka, koja ograničava operacije koje se mogu izvršiti nad tom strukturom podataka.*

Pitanje 6.18. *Dečak slaže tanjire jedan na drugi. Tanjiri mogu biti različitog poluprečnika. U svakom koraku on može da doda novi tanjir na vrh gomile, da skine tanjir sa vrha (ako je gomila neprazna) i da ispita koji je najveći tanjir u gomili. Opisati strukturu podataka koja omogućava izvođenje ovih operacija u konstantnom vremenu.*

Zadatak 6.19. *Napiši program koji koristeći stek ispisuje sve linije koje se učitavaju sa standardnog ulaza u obratnom redosledu.*

Zadatak 6.20. *Pregledač veba pamti istoriju posećenih sajtova i korisnik ima mogućnost da se vraća unatrag na sajtove koje je ranije posetio. Napisati program koji simulira istoriju pregledača tako što se učitavaju adrese posećenih sajtova (svaka u posebnom redu), a kada se učita red u kome piše back pregledač se vraća na poslednju posećenu stranicu. Ako se naredbom* `back` *vratimo na početnu stranicu, ispisati -. Ako se na početnoj stranici izda naredba* `back`*, ostaje se na početnoj stranici. Program treba da ispiše sve sajtove koje je korisnik posetio.*

Zadatak 6.21. *Tokom rada sa stekom ukupno n puta je izvršena operacija* `push` *kojom se neka vrednost postavlja na vrh steka i ukupno n puta je izvršena operacija* `pop` *kojom je element skinut sa vrha steka. Ako je poznat niz brojeva koji su redom bili argumenti operacije* `push` *i niz brojeva koji su redom dobijani kao rezultat operacije* `pop`*, napiši program koji određuje redosled operacija* `push` *i* `pop`*. Na primer, ako je niz argumenata operacije* `push` *jednak 1, 2, 3, 4, 5, a operacije* `pop` *3, 2, 5, 4, 1, tada je redosled operacija bio* `push`*,* `push`*,* `push`*, ,* `pop`*,* `pop`*,* `push`*,* `push`*,* `pop`*,* `pop`*,* `pop`*.*

Zadatak 6.22. Putanja *u sistemu Linux sastoji se od naziva direktorijuma razdvojenih karakterom / i eventualno imenom datoteke na kraju. Pretpostavićemo da se imena direktorijuma i datoteka sastoje od malih slova engleske abecede, cifara, podvlaka i karaktera tačka. Specijalni direktorijumi u putanji su* `.` *koji označava tekući direktorijum i* `..` *koji označava roditeljski direktorijum (roditeljskim direktorijumom korenog direktorijuma smatra se on sâm). Sažimanje putanje podrazumeva da se iz njenog naziva uklone ti specijalni direktorijumi. Na primer, od putanje* `/abc/./def/../jkl/../mnp/doc.txt` *sažimanjem se dobija* `/abc/mnp/doc.txt`*. Napisati program koji učitava putanju i sažima je.*

Zadatak 6.23.
(a) *Korišćenjem steka nerekurzivno implementirati funkciju brzog sortiranja.*
(b) *Korišćenjem steka nerekurzivno implementirati rešenje problema Hanojskih kula.*

Zadatak 6.24. *Data je matrica koja sadrži samo nule i jedinice. Definišimo ostrvo kao povezanu oblast jedinica tj. oblast koja se sastoji samo od jedinica takvu da se od bilo kojeg polja na ostrvu može stići do bilo kojeg drugog polja, krećući se u 8 dopuštenih smerova. Definisati nerekurzivnu funkciju koji određuje broj ostrva u toj matrici (videti zadatak 4.5). Na primer, u narednoj matrici postoje 3 ostrva.*

```
0011100001
0011110011
0001110100
1000000000
1100000000
```

Uporediti efikasnost rekurzivnog i nerekurzivnog rešenja (na velikim, slučajno generisanim ulaznim matricama).

Zadatak 6.25. *U izrazu učestvuju sledeće vrste zagrada* `(`, `)`, `{`, `}`, `[` *i* `]`. *Napiši program koji proverava da li su u unetom izrazu zagrade ispravno uparene.*

Zadatak 6.26. *Niz se transformiše tako što se svakih k ili više uzastopnih pojavljivanja nekog elementa brišu. Napiši program linearne složenosti koji određuje sadržaj niza nakon iscrpne primene ove transformacije. Na primer, ako je $k = 3$ i početni niz je $1, 1, 2, 2, 2, 2, 1, 3, 4, 4, 5, 5, 5, 4, 4, 3, 2, 1, 1, 1$, nakon sažimanja se dobija niz $3, 3, 2$.*

Zadatak 6.27. *Definisati strukturu podataka koja nalikuje steku i dopušta izvođenje sledećih operacija u konstantnoj složenosti:*

- *Element x se postavlja na vrh steka.*
- *Uklanja se element koji se najčešće pojavljuje od svih elemenata koji su trenutno na steku. Ako je više takvih elemenata uklanja se onaj koji je poslednji dodat. Na primer, ako su na steku elementi 1, 2, 2, 1, 3, uklanja se element 1 i stek postaje 1, 2, 2, 3.*

Zadatak 6.28. *Definišimo da je potpuno zagrađeni izraz ili cifra ili je oblika $(i_1 op i_2)$, gde su i_1 i i_2 potpuno zagrađeni izrazi, a op je ili operator $+$ ili operator $*$. Na primer, $((3 + (4 * 5)) * 6)$ je potpuno zagrađeni izraz.*
(a) *Definisati rekurzivnu funkciju koja učitava potpuno zagrađeni izraz i izračunava njegovu vrednost.*
(b) *Korišćenjem steka definisati nerekurzivnu funkciju koja učitava potpuno zagrađeni izraz i izračunava njegovu vrednost.*

Zadatak 6.29. *Dat je aritmetički izraz koji sadrži jednocifrene brojeve i operacije sabiranja i množenja.*
(a) *Napisati program koji prevodi dati izraz u postfiksni oblik.*
(b) *Napisati program koji izračunava vrednost učitanog aritmetičkog izraza. Uputstvo: koristiti dva steka.*

6.5 Red

Red (engl. queue) je struktura koja funkcioniše na principu FIFO (*first in, first out*). Kao struktura red ponaša se, na primer, red u čekaonici kod zubara: u red se dodaje na kraj, a prvi iz reda izlazi (i ide na pregled) onaj ko je prvi došao (a ko je stigao poslednji – poslednji ide na pregled)[2]. Element reda

[2]Postoji varijanta reda koji se naziva *red sa prioritetom* (engl. Priority Queue) iz kojeg se elementi uklanjaju pre svega na osnovu njihovih prioriteta, a ne redosleda ulaska u red.

može da uključuje više članova različitih tipova. Red ima sledeće dve osnovne operacije (koje treba da budu složenosti $O(1)$):

enqueue: dodaje element na kraj reda (čita se „enkju“, ponegde se ova operacija zove i `add`);

dequeue: uklanja element sa početka reda (čita se „dikju“, ponegde se ova operacija zove i `get`).

Implementacija reda obično se dizajnira tako da korisniku pruža samo ključne operacije i da jedna implementacija reda može lako da se zameni drugom. Pored dve osnovne operacije, `enqueue` i `dequeue`, obično su raspoložive i funkcije za inicijalizaciju reda, za brisanje reda, očitavanje broja elemenata u redu i slično.

Struktura red pogodna je za mnoge primene, od kojih su neke opisane u nastavku udžbenika.

6.5.1 Implementacija reda korišćenjem niza

Red je moguće implementirati korišćenjem niza u koji se elementi smeštaju kružno (kada se popuni poslednji element niza, smeštanje elemenata nastavlja se od početka). Ovakva implementacija vrlo je jednostavna i prilično efikasna. Ako je poznat maksimalni kapacitet reda (maksimalni broj elemenata koji se u nekom trenutku nalaze u redu), u implementaciji se može koristiti i statički alociran niz ili niz koji je dinamički alociran, ali mu se dužina ne menja tokom rada programa (takvu implementaciju ćemo prikazati u nastavku). S druge strane, moguće je tokom rada vršiti i dinamičke realokacije (kada se tekući niz popuni, moguće je proširiti ga), čime se dobija puna fleksibilnost (ta varijanta nije razmatrana u nastavku).

Održavaćemo dve pozicije tj. dva pokazivača: na početak reda i na prvu slobodnu poziciju (iza poslednjeg elementa u redu). Na početku će obe pozicije biti inicijalizovane na nulu. Kada god su te dve pozicije jednake, znaćemo da je red prazan. Da bismo razlikovali situaciju praznog reda i popunjenog reda, niz ćemo alocirati tako da može da čuva jedan element više od maksimalnog kapaciteta reda (u niz koji sadrži $n + 1$ elemenata može se smestiti najviše n elemenata reda). Smatraćemo da je red popunjen ako se prva slobodna pozicija nalazi neposredno ispred prvog elementa reda (specijalno, moguće je da prvi element bude na poziciji 0, a da je prva slobodna pozicija poslednja pozicija u nizu). Ako je red popunjen, nije dopušteno dodavanje novog elementa, te nikada neće biti popunjeni svi elementi niza.

Dodavanje vršimo na slobodnu poziciju i povećavamo je za 1, po modulu dužine niza $n + 1$. Ovim se dobija elegantan kôd (`k = (k + 1) % (n + 1)`), ali pošto je operacija deljenja po modulu često vremenski zahtevna, bolje je izbeći je i upotrebiti jednostavno grananje (nakon uvećanja krajnje pozicije proveravamo da li je dostigla dužinu niza i ako jeste, vraćamo je na nulu).

Uklanjanje sa početka reda vršimo tako što poziciju prvog elementa u redu uvećavamo za 1 (ponovo – po modulu dužine niza).

Tekući broj elemenata u redu bi se mogao elegantno izračunati kao razlika između dva pokazivača, ponovo računata po modulu dužine niza, korišćenjem veze $(k - p) \bmod (n + 1) = (k - p + n + 1) \bmod (n + 1)$, tj. izraza `(k - p + n + 1) % (n + 1)`, u kojem se pre izračunavanja ostatka razlika uvećava za $n + 1$ da bi se izbeglo računanje ostatka pri deljenju negativnih brojeva. Ipak, poželjno je izbeći računanje ostatka na sledeći način: računa se razlika $k - p$ i ako je ona negativna, uvećava se za $n + 1$. U narednom kodu prikazana je takva, brža implementacija. Provera da li je red prazan se onda može svesti na proveru da li je broj elemenata jednak 0, a provera da li je red pun se može svesti na proveru da li je broj elemenata jednak n (nikada se ne dopušta da se svih $n + 1$ elemenata niza popuni).

Definisaćemo strukturu `Red` u kojoj se čuvaju svi potrebni podaci koji opisuju jedan red (pokazivač `podaci` na niz elemenata koji će se alocirati dinamički, dimenzija `velicina` tog niza i dva pokazivača `pocetak` i `kraj` koji ograničavaju popunjeni deo reda). Opisanu strukturu uvodi sledeća datoteka `red_niz.h` i ona se uključuje kada god se koristi ova implementacija reda.

```
#ifndef __RED_NIZ_H__
#define __RED_NIZ_H__

typedef enum {
  OK,
  NEDOVOLJNO_MEMORIJE,
  NEMA_ELEMENATA,
  PUN_RED
} status;

typedef struct {
  int pocetak, kraj;
  int velicina;
  int *podaci;
} Red;

status inicijalizuj_red(Red* red, int n);
status enqueue(Red* red, int podatak);
status dequeue(Red* red, int *podatak);
size_t broj_elemenata_reda(const Red* red);
int red_je_prazan(const Red* red);
int red_je_pun(const Red* red);
void obrisi_red(Red* red);

#endif
```

Sadržaj datoteke `red_niz.c` može da izgleda ovako:

```c
#include <stdlib.h>
#include "red_niz.h"

static int red_je_pun(const Red* red);

status inicijalizuj_red(Red* red, int n)
{
  red->velicina = n + 1;
  red->podaci = malloc(red->velicina*sizeof(int));
  if (red->podaci == NULL)
    return NEDOVOLJNO_MEMORIJE;
  red->pocetak = 0;
  red->kraj = 0;
  return OK;
}

status enqueue(Red* red, int podatak)
{
  if (red_je_pun(red))
    return PUN_RED;
  red->podaci[red->kraj] = podatak;
  red->kraj++;
  if (red->kraj == red->velicina)
    red->kraj = 0;
  return OK;
}

status dequeue(Red* red, int *podatak)
{
  if (red_je_prazan(red))
    return NEMA_ELEMENATA;
  *podatak = red->podaci[red->pocetak];
  red->pocetak++;
  if (red->pocetak == red->velicina)
    red->pocetak = 0;
  return OK;
}

size_t broj_elemenata_reda(const Red* red)
{
  int razlika = red->kraj - red->pocetak;
  if (razlika < 0)
```

```
    razlika += red->velicina;
  return razlika;
}

int red_je_prazan(const Red* red)
{
  return broj_elemenata_reda(red) == 0;
}

int red_je_pun(const Red* red)
{
  return broj_elemenata_reda(red) + 1 == red->velicina;
}

void obrisi_red(Red* red)
{
  free(red->podaci);
  red->velicina = 0;
  red->pocetak = 0;
  red->kraj = 0;
}
```

Naredna funkcija ilustruje korišćenje ovako implementiranog reda (a analogno se može koristiti i neka druga implementacija reda).

```
#include <stdio.h>
#include "red_niz.h"

int main()
{
  Red red;
  int podatak;
  if (inicijalizuj_red(&red) != OK)
    return -1;

  if (enqueue(&red, 1) != OK)
    obrisi_red(&red);

  if (enqueue(&red, 2) != OK)
    obrisi_red(&red);

  dequeue(&red, &podatak);
  printf("%d\n", podatak);
  if (enqueue(&red, 3) != OK)
    obrisi_red(&red);
```

```
  dequeue(&red, &podatak);
  printf("%d\n", podatak);
  dequeue(&red, &podatak);
  printf("%d\n", podatak);
  return 0;
}
```

Navedeni program daje sledeći izlaz:

```
1
2
3
```

Naglasimo da je funkcija `dequeue` pozivana u situacijama kada mora da uspe (jer red nije prazan), te nisu vršene provere povratne vrednosti.

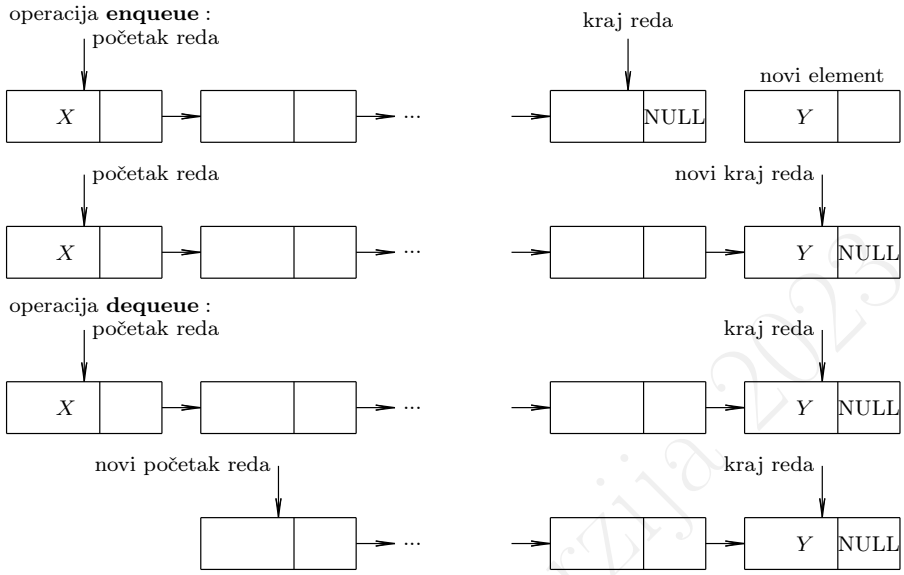
Primeri upotrebe ovako implementiranog reda biće navedeni kasnije.

6.5.2 Implementacija reda korišćenjem povezane liste

Red se može implementirati korišćenjem povezanih lista. Osnovne operacije nad redom implementiranim korišćenjem povezane liste ilustrovane su na slici 6.7.

Struktura reda podrazumeva čuvanje početka i kraja povezane liste i broja elemenata u redu (jer je određivanje broja elemenata liste operacija linearne složenosti). Koristeći ranije navedene opšte funkcije za rad sa povezanim listama, funkcije za rad sa redom mogu se implementirati u vidu datoteka `red_lista.h` i `red_lista.c` na sledeći način:

```
#ifndef __RED_LISTA_H__
#define __RED_LISTA_H__

typedef enum {
  OK,
  NEDOVOLJNO_MEMORIJE,
  NEMA_ELEMENATA
} status;

typedef struct cvor {
  int podatak;
  struct cvor *sledeci;
} cvor;

typedef struct {
  cvor *pocetak;
```



Slika 6.7: Ilustracija osnovnih operacija nad redom implementiranim kao povezana lista

```
  cvor *kraj;
  size_t broj_elemenata;
} Red;

status inicijalizuj_red(Red* red);
status enqueue(Red* red, int podatak);
status dequeue(Red* red, int *podatak);
size_t broj_elemenata_reda(const Red* red);
int red_je_prazan(const Red* red);
void obrisi_red(Red* red);

#endif
```

```
#include <stdlib.h>
#include "red_lista.h"

static cvor *novi_cvor(int podatak);
static status dodaj_na_kraj(cvor **pokazivac_na_pocetak,
                            cvor **pokazivac_na_kraj,
                            int podatak);
```

```
static status obrisi_sa_pocetka(cvor **pokazivac_na_pocetak,
                                 cvor **pokazivac_na_kraj,
                                 int *podatak);
static int red_je_pun(const Red *red);

static cvor *novi_cvor(int podatak)
{
  cvor *novi = malloc(sizeof(cvor));
  if (novi == NULL)
    return NULL;
  novi->podatak = podatak;
  return novi;
}

static status dodaj_na_kraj(cvor **pokazivac_na_pocetak,
                            cvor **pokazivac_na_kraj,
                            int podatak)
{
  cvor *novi = novi_cvor(podatak);
  if (novi == NULL)
    return NEDOVOLJNO_MEMORIJE;
  novi->sledeci = NULL;
  if (*pokazivac_na_pocetak == NULL)
    *pokazivac_na_pocetak = novi;
  else
    (*pokazivac_na_kraj)->sledeci = novi;
  *pokazivac_na_kraj = novi;
  return OK;
}

static status obrisi_sa_pocetka(cvor **pokazivac_na_pocetak,
                                 cvor **pokazivac_na_kraj,
                                 int *podatak)
{
  cvor *p;
  if (*pokazivac_na_pocetak == NULL)
    return NEMA_ELEMENATA;
  p = *pokazivac_na_pocetak;
  *podatak = (*pokazivac_na_pocetak)->podatak;
  *pokazivac_na_pocetak = (*pokazivac_na_pocetak)->sledeci;
  free(p);

  if (*pokazivac_na_pocetak == NULL)
     *pokazivac_na_kraj = NULL;
```

```
  return OK;
}

status inicijalizuj_red(Red *red) {
  red->pocetak = NULL;
  red->kraj = NULL;
  red->broj_elemenata = 0;
  return OK;
}

status enqueue(Red* red, int podatak)
{
  status s = dodaj_na_kraj(&(red->pocetak), &(red->kraj),
                           podatak);
  if (s == OK)
    red->broj_elemenata++;
  return s;
}

status dequeue(Red* red, int* podatak)
{
  status s = obrisi_sa_pocetka(&(red->pocetak), &(red->kraj),
                               podatak);
  if (s == OK)
    red->broj_elemenata--;
  return s;
}

size_t broj_elemenata_reda(const Red* red)
{
  return red->broj_elemenata;
}

int red_je_prazan(const Red* red)
{
  return broj_elemenata_reda(red) == 0;
}

static int red_je_pun(const Red* red)
{
   return 0;
}

void obrisi_red(Red* red)
```

```
{
  while (red->pocetak)  {
    cvor *p = red->pocetak;
    red->pocetak = red->pocetak->sledeci;
    free(p);
  }
}
```

Kvalifikatorom `static` označene su funkcije koje su predviđene samo za interno korišćenje (tj. korisnik strukture red ne treba i ne može da poziva ove funkcije).

Jednostavnosti radi, funkcija `red_je_pun` uvek vraća vrednost *netačno* (tj. `0`), umesto da proverava da li je listu koja se koristi moguće dinamički proširiti.

Kao i kod implementacije steka pomoću povezane liste, mana ovakvog rešenja je potreba za alokacijom i dealokacijom čvora prilikom svakog dodavanja i uklanjanja elementa, što je relativno skupa operacija (iako se smatra operacijom složenosti $O(1)$).

6.5.3 Implementacija reda korišćenjem dva steka

U funkcionalnim programskim jezicima su obično direktno podržane liste kojima se elementi mogu dodavati na početak i uklanjati sa početka. To je dovoljno da se implementira stek. Sa druge strane, implementacija reda nije jednostavna. Ipak, pokazuje se da se red može jednostavno implementirati pomoću dva steka (takva implementacija nije najbolji izbor za jezik C, ali je jako često korišćena u funkcionalnim jezicima).

Osnovna ideja je da su elementi podeljeni na jedan ulazni i jedan izlazni stek. Elemente dodajemo na vrh ulaznog steka, a skidamo ih sa vrha izlaznog steka. Kad god je izlazni stek prazan, a potrebno je da iz reda uklonimo element, tada sve elemente sa ulaznog steka prebacujemo na izlazni, pri čemo to moramo da uradimo u obratnom redosledu (vrh ulaznog steka sadrži element koji je dodat poslednji i on mora da se pojavi na dnu izlaznog steka). Dakle, redosled elemenata u redu se dobija tako što se spoje izlazni stek (od vrha ka dnu) i ulazni stek (od dna ka vrhu). Na primer, ako su elementi izlaznog steka od vrha ka dnu $5, 6, 2$, a ulaznog steka od vrha ka dnu $3, 7, 4$, tada je sadržaj reda od prvog do poslednjeg elementa $5, 6, 2, 4, 7, 3$.

Funkcija za dodavanje elementa u red je složenosti $O(1)$. Sa funkcijom za izbacivanje, stvar je malo komplikovanija. Ukoliko postoji element na izlaznom steku, njena složenost je takođe $O(1)$. Međutim, ukoliko ne postoji, onda ona prebacuje jedan po jedan element sa ulaznog steka na izlazni i njena složenost odgovara broju elemenata nakupljenih na ulaznom steku. Znači, u najgorem slučaju to je funkcija linearne složenosti. Međutim, analiza najgoreg slučaja ovde je previše pesimistična. Naime, kada se jednom prebace svi elementi sa ulaznog na izlazni stek, znamo da će u narednim pozivima te funkcije ona raditi veoma efikasno. Njeno ponašanje je takvo da ona veoma često radi veoma

efikasno, a s vremena na vreme naiđe trenutak u kojem se desi poziv koji traje dugo (ali, što se više vremena u tom pozivu provede, to će u većem broju narednih poziva ona raditi efikasno). Razmotrimo ukupno vreme potrebno da se n elemenata postavi i ukloni iz reda. Svaki element će jednom biti postavljen na ulazni stek, zatim jednom prebačen sa ulaznog na izlazni stek i na kraju jednom uklonjen sa izlaznog steka. Znači, ukupan broj operacija nad n elemenata će biti $O(3n)$, tako da će u proseku vreme rada funkcije uklanjanja sa steka biti $O(1)$. Ova vrsta analize u kojoj se razmatra ukupno (pa i prosečno) vreme n izvršavanja algoritma zove se *amortizovana analiza* i često daje realniju sliku o ponašanju tog algoritma nego analiza najgoreg slučaja.

Opisana podrška za red sastoji se od naredne datoteke `red_dva_steka.h`:

```
#ifndef __RED_DVA_STEKA_H__
#define __RED_DVA_STEKA_H__

#include "stek_niz.h"

typedef struct {
  Stek ulaz, izlaz;
} Red;

status inicijalizuj_red(Red* red);
status enqueue(Red* red, int podatak);
status dequeue(Red* red, int *podatak);
size_t broj_elemenata_reda(const Red* red);
int red_je_prazan(const Red* red);
void obrisi_red(Red* red);

#endif
```

i od naredne datoteke `red_dva_steka.c`:

```
#include <stdlib.h>
#include "red_dva_steka.h"

static int red_je_pun(const Red* red);

status inicijalizuj_red(Red* red)
{
  status s;
  if ((s = inicijalizuj_stek(&(red->ulaz))) != OK)
    return s;
  if ((s = inicijalizuj_stek(&(red->izlaz))) != OK)
    return s;
}
```

```
status enqueue(Red* red, int podatak)
{
  return push(&(red->ulaz), podatak);
}

status dequeue(Red* red, int *podatak)
{
  status s;
  if (stek_je_prazan(&(red->izlaz))) {
    /* prebacujemo sve elemente sa ulaznog na izlazni stek */
    while (!stek_je_prazan(&(red->ulaz))) {
      int tmp;
      if ((s = pop(&(red->ulaz), &tmp)) != OK)
        return s;
      if ((s = push(&(red->izlaz), tmp)) != OK)
        return s;
    }
  }
  /* vracamo podatak na vrhu izlaznog steka */
  return pop(&(red->izlaz), podatak);
}

size_t broj_elemenata_reda(const Red* red)
{
   return broj_elemenata(&(red->ulaz)) +
          broj_elemenata(&(red->izlaz));
}

int red_je_prazan(const Red* red)
{
   return (stek_je_prazan(&(red->ulaz)) &&
           stek_je_prazan(&(red->izlaz)));
}

static int red_je_pun(const Red* red)
{
   return 0;
}

void obrisi_red(Red* red)
{
   obrisi_stek(&(red->ulaz));
   obrisi_stek(&(red->izlaz));
```

```
}
```

Pretpostavljamo da su stekovi implementirani tako da se dinamički proširuju i da red nikada nije pun, te funkcija `red_je_pun` uvek vraća vrednost *netačno* (tj. `0`).

Opisana podrška za red zahteva i datoteke `stek_niz.h` i `stek_niz.c` iz poglavlja 6.4.1 (ili neku drugu implementaciju steka). Da bi se izbeglo preklapanje imena `broj_elemenata`, u datotekama za red je ova funkcija nazvana `broj_elemenata_reda`. Primetimo da `red_dva_steka.h` ne definiše nabrojivi tip `status` (jer se ta definicija uključuje iz datoteke `stek_niz.h`).

6.5.4 Primer primene strukture red: poslednjih k učitanih brojeva

Često se tokom obrade podataka učitava dugačak niz podataka, ali je u svakom trenutku potrebno poznavati i obrađivati samo poslednjih k učitanih elemenata. Na primer, ako se analizira temperatura, tada ona može da varira od dana do dana, međutim, ako se umesto svakog dana posmatra prosečna temperatura tokom prethodnih k dana (na primer, 7) dobija se mnogo stabilniji izveštaj. U programu tada nema potrebe učitavati i čuvati sve podatke istovremeno, već je moguće čuvati samo poslednjih k učitanih podataka (čime možemo značajno štedeti memoriju, jer k može biti mnogo manje od ukupnog broja podataka). Nakon učitavanja svakog novog podatka izbacuje se onaj koji je najranije učitan među k podataka koje čuvamo i u tu kolekciju se dodaje novi učitani podatak. Ta kolekcija k podataka ponaša se kao red (element koji je prvi upisan, prvi i ispada tj. elementi se uklanjaju sa početka, a dodaju na kraj). Pošto se obrađuju redom sve uzastopne k-torke, kažemo da je u pitanju „pokretni prozor".

Jedan veoma jednostavan zadatak koji ilustruje ovu upotrebu reda je onaj u kojem se traži da se među n učitanih brojeva ispiše poslednjih k. Učitavamo n elemenata i ubacujemo ih u red, ali pre ubacivanja elementa proveravamo da li red već sadrži tačno k elemenata i ako sadrži, tada pre ubacivanja novog uklanjamo element iz reda.

```c
#include <stdio.h>
#include "red_dva_steka.h"

int main() {
  int k, n, i;
  Red red;
  if (scanf("%d %d", &k, &n) != 2 || n<=0 || k<=0)
    return -1;
  if (inicijalizuj_red(&red) != OK)
    return -1;
```

```
  for (i = 0; i < n; i++) {
    int x;
    if (scanf("%d", &x) != 1)
      return -1;
    if (broj_elemenata_reda(&red) == k) {
      int tmp;
      dequeue(&red, &tmp);
    }
    if (enqueue(&red, x) != OK)
      return -1;
  }

  printf("\n");
  while (!red_je_prazan(&red)) {
    int x;
    dequeue(&red, &x);
    printf("%d ", x);
  }
  printf("\n");
  obrisi_red(&red);
  return 0;
}
```

Program za navedene ulazne podatke:

```
4 10
1 2 3 4 5 6 7 8 9 10
```

daje sledeći izlaz:

```
7 8 9 10
```

6.5.5 Generički red

Kôd naveden u poglavljima 6.5.1, 6.5.2 i 6.5.3 ilustruje kako se može implementirati red čiji elementi sadrže podatak tipa `int`. Naravno, veoma često potrebno je koristiti red čiji elementi sadrže podatak nekog drugog tipa `<type>`. To se može postići na nekoliko načina, analogno kao za generički stek (poglavlje 6.4.4).

6.5.6 Primer primene strukture red: pretraga u širinu i pronalaženje puta na mapi

Razmotrimo ponovo problem pronalaženja puta na mapi koji je opisan u poglavljima 5.5.1 i 6.4.5. Ovaj problem može se rešiti i korišćenjem reda. Kôd

je isti kao kôd prikazan u poglavlju 6.4.5, s tim što se umesto steka koristi red i umesto funkcija funkcija `push` i `pop` koriste se funkcije `enqueue` i `dequeue`. I ovako dobijeno rešenje predstavlja sistematičan proces pretrage i ukoliko put između polaznog i ciljnog polja postoji, onda će navedena funkcija to sigurno da utvrdi. Međutim, poredak obilaska polja kada se koristi red („obilazak u širinu“) drugačiji je nego kada se koristi stek. U slučaju kada se koristi red dobija se najkraći put između data dva polja.

Pitanja i zadaci za vežbu

Pitanje 6.19. *Koje su osnovne operacije reda i koja je njihova složenost u implementaciji baziranoj na listama?*

Pitanje 6.20. *Navesti nekoliko primera realnih programa u čijoj se implementaciji može upotrebiti struktura podataka red.*

Pitanje 6.21. *Kada i zašto programeri treba da koriste red umesto niza ili liste?*

Pitanje 6.22. *Definisati funkciju* `dequeue` *koja čita i briše element sa kraja reda.*

Zadatak 6.30. *Napiši program koji određuje jedan od načina da skakač na šahovskoj tabli obiđe svako polje table tačno jednom, krenuvši iz donjeg levog ugla.*

Zadatak 6.31. *Sa standardnog ulaza se učitava niz koji sadrži n celih brojeva. Napiši program koji korišćenjem memorije $O(k)$ određuje maksimalni zbir nekih k uzastopnih elemenata tog niza.*

Zadatak 6.32. *Brojevi u nizu su takvi da za svaki element važi ili da su svi elementi ispred njega manji od njega ili da su svi elementi ispred njega veći od njega. Na primer, niz 5, 8, 12, 4, 2, 13, 19, 1 zadovoljava to svojstvo. Napiši program koji u linearnoj složenosti sortira taj niz.*

Zadatak 6.33. *Razmatramo niz brojeva čiji su prosti činioci samo* 2, 3 *i* 5 *(svaki činilac može da se javi nula i više puta). To su brojevi* 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, ... *Napiši program koji određuje n-ti član ovog niza (brojanje kreće od 0).*

Zadatak 6.34. *Napisati program koji vrši popunjavanje konture na slici (opisan u poglavlju u rekurziji), tako da ne koristi rekurziju.*

Zadatak 6.35. *Korišćenjem reda definisati algoritam koji će u linearnom vremenu izlistati binarne reprezentacije brojeva od* 1 *do n. Na primer, za $n = 16$, program treba da ispiše 1 10 11 100 101 110 111 1000 1001 1010 1011 1100 1101 1110 1111 10000.*

6.6 Binarno stablo

Stablo (ili *drvo*) je struktura koja prirodno opisuje određene vrste hijerarhijskih objekata (na primer, porodično stablo, logički izraz, aritmetički izraz, ...). Stablo se sastoji od čvorova i usmerenih grana između njih. Svaka grana povezuje jedan čvor (u tom kontekstu — *roditelj*) sa njegovim *detetom*, *neposrednim potomkom*. Čvor koji se zove *koren stabla* nema roditelja. Svaki drugi čvor ima tačno jednog roditelja. *List* je čvor koji nema potomaka. Čvor A je *predak* čvora B ako je A roditelj čvora B ili ako je A predak roditelja čvora B. Koren stabla je predak svim čvorovima stabla (osim sebi). Zbog toga je moguće do bilo kog čvora stabla doći od korena (jedinstveno određenim putem). Stablu mogu da se jednostavno dodaju novi čvorovi kao deca postojećih čvorova.

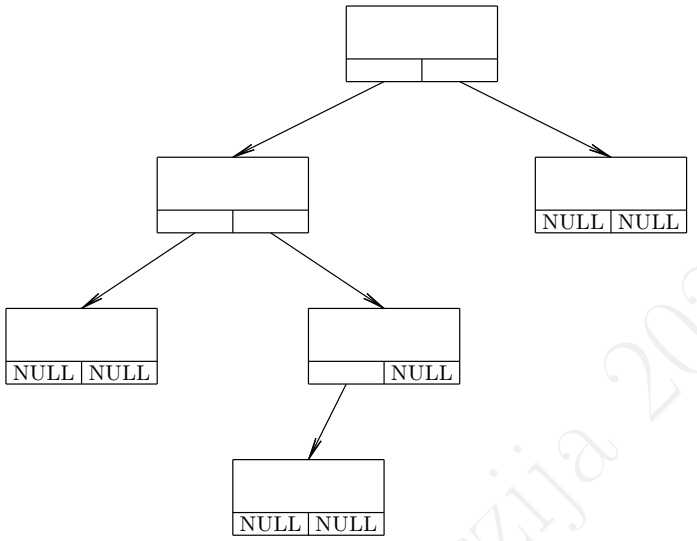
Stablo u kojem svaki čvor ima najviše dva deteta zovemo *binarno stablo* (ili *binarno drvo*). U nastavku će biti reči samo o binarnim stablima.

Binarna stabla mogu se implementirati koristeći dinamičku alokaciju i pogodne strukture sa pokazivačima (slika 6.8), analogno povezanim listama. Podaci u čvorovima stabla mogu biti različitog tipa (i obično ih ima više). Jednostavnosti radi, u primerima u nastavku biće smatrano da je element stabla deklarisan na sledeći način (tj. da ima samo jedan podatak, podatak tipa `int`):

```
typedef struct cvor_stabla {
  int podatak;
  struct cvor_stabla* levo;
  struct cvor_stabla* desno;
} cvor_stabla;
```

Elementi stabla kreiraju se pojedinačno i onda spajaju sa tekućim stablom. Novi element stabla može biti kreiran sledećom funkcijom, analogno kreiranju novog elementa liste:

```
cvor_stabla* novi_cvor_stabla(int podatak)
{
  cvor_stabla* novi = malloc(sizeof(cvor_stabla));
  if (novi == NULL)
    return NULL;
  novi->podatak = podatak;
  return novi;
}
```



Slika 6.8: Binarno stablo

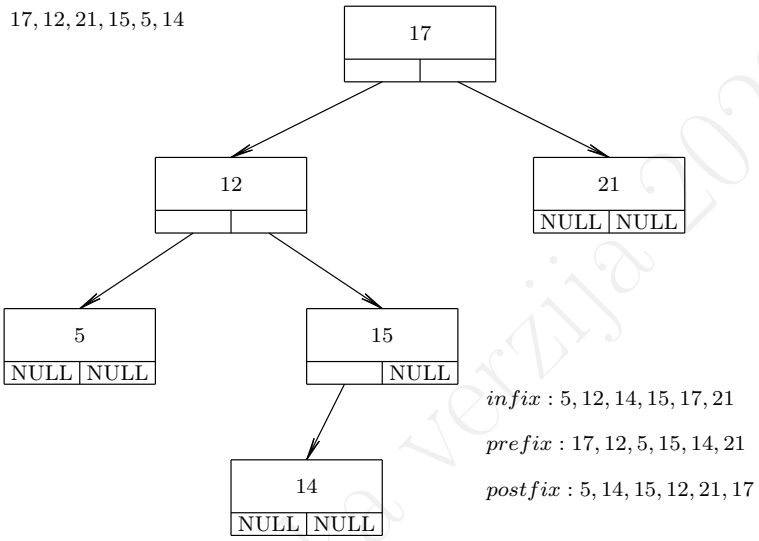
6.6.1 Uređeno binarno stablo

U *uređenom binarnom stablu*[3], podaci su organizovani u odnosu na neku izabranu relaciji poretka, na primer, za brojeve to može biti uobičajeno uređenje (opadajuće ili rastuće) nad brojevima. U nastavku ćemo podrazumevati da se termini „manje“ i „veće“ odnose na tu relaciju poretka, fiksnu za jedno binarno stablo. Za svaki čvor n uređenog binarnog stabla, vrednosti podatka svakog čvora iz njegovog levog podstabla manje su od ili jednake vrednosti podatka u čvoru n, a vrednosti podatka svakog čvora iz njegovog desnog podstabla su veće od vrednosti podatka u čvoru n (ilustrovano na slici 6.9).

Organizacija podataka u uređenom binarnom stablu omogućava efikasnije obrade podataka u stilu binarne pretrage. Međutim, ta efikasnost se gubi ako je binarno stablo degenerisano (na primer, ako svaki čvor ima samo desnog neposrednog potomka) ili skoro degenerisano i tada je složenost operacija dodavanja i pronalaženja elementa u stablu linearna (po broju elemenata stabla). Postoje vrste *balansiranih* binarnih stabala u kojima se garantuje (zahvaljujući netrivijalnom održavanju balansiranosti) da se visine levog i desnog podstabla u svakom čvoru razlikuju najviše za jedan (na primer, crveno-crna stabla (engl. red-black trees, RBT) ili Adeljson-Veljski-Landis-ova stabla (AVL)) i u takvim stablima složenost operacija dodavanja, brisanja i pronalaženja elementa je logaritamska (po broju elemenata stabla). Implementacija takvih stabala je dosta komplikovana i prevazilazi obim ovog udžbenika, te će biti prikaza-

[3]Uređeno binarno stablo naziva se i *binarno pretraživačko stablo* ili *stablo binarne pretrage*, a na engleskom *binary search tree* (BST).

na samo implementacija nebalansiranih stabala. Iako su ona u praksi najčešće neupotrebljiva, zbog loše složenosti najgoreg slučaja, ona mogu pomoći da se usvoje osnovni koncepti i osnovne ideje, koje se dalje nadograđuju i dovode do efikasne, upotrebljive implementacije. Uređeno binarno stablo ima mnoge primene, od kojih su neke opisane u nastavku.



Slika 6.9: Uređeno binarno stablo

Dodavanje elementa u uređeno binarno stablo

Binarno stablo kreira se jednostavno dodavanjem jedne po jedne ulazne vrednosti. Od prve ulazne vrednosti kreira se koren stabla i ulazna vrednost pridružuje se korenu. Svaka sledeća nova ulazna vrednost v dodaje se na jedinstveno određenu (tj. jedinu ispravnu) poziciju. Rekurzivnu implementaciju prilično je jednostavno napraviti:

- Ako je stablo prazno, rezultat je stablo koje se sastoji od samo jednog čvora.
- Inače, ako je vrednost elementa koji se ubacuje manja od ili jednaka vrednosti u korenu, ubacivanje se vrši rekurzivno u levo podstablo.
- Inače, ubacivanje se vrši rekurzivno u desno podstablo.

Naglasimo da opisani pristup omogućava višestruka ponavljanja vrednosti u čvorovima stabla (ako je vrednost koja se ubacuje jednaka vrednosti u korenu, onda se ona ubacuje u levo podstablo). Time se u stablu čuva (sortirani)

multiskup vrednosti. Ako bismo umesto multiskupa želeli implementaciju skupa (tj. ako ne želimo višestruka pojavljivanja iste vrednosti), opisani rekurzivni postupak trebalo bi neznatno izmeniti: prilikom ubacivanja nove vrednosti u stablo, ako je ona jednaka vrednosti u korenu, onda se dalji postupak jednostavno prekida, ne menjajući stablo.

Naredna rekurzivna funkcija kreira čvor koji sadrži zadati podatak `podatak` i umeće ga u uređeno binarno stablo sa korenom `**pokazivac_na_koren`. Ukoliko uspe kreiranje čvora vraća se vrednost nabrojivog tipa `OK`, inače druga vrednost koja daje status (tj. tip greške). Prosleđeni koren može da se promeni u slučaju da je stablo inicijalno bilo prazno.

```
typedef enum {
  OK,
  NEDOVOLJNO_MEMORIJE,
  NEMA_ELEMENATA
} status;

/* ... */

status dodaj_u_stablo(cvor_stabla** pokazivac_na_koren,
                      int podatak)
{
  if (*pokazivac_na_koren == NULL) {
    cvor_stabla* novi = novi_cvor_stabla(podatak);
    if (novi == NULL)
      return NEDOVOLJNO_MEMORIJE;
    novi->levo = NULL;
    novi->desno = NULL;
    *pokazivac_na_koren = novi;
    return OK;
  }
  if (podatak <= (*pokazivac_na_koren)->podatak)
    return dodaj_u_stablo(&((*pokazivac_na_koren)->levo),
                          podatak);
  else
    return dodaj_u_stablo(&((*pokazivac_na_koren)->desno),
                          podatak);
}
```

Rekurzija je repna i lako se može eliminisati. Ako se u stablo dodaju podaci u uređenom redosledu, oni se stalno smeštaju na jednu stranu i stablo se može izdegenerisati u listu, što, kako što je rečeno, dovodi do loših performansi (složenost najgoreg slučaja je linearna u odnosu na broj elemenata u stablu). Zbog toga je moguće da u rekurzivnoj implementaciji dođe do prekoračenja steka. U slučaju kada se koriste samobalansirajuća stabla, visina stabla

logaritamski zavisi od broja čvorova, složenost osnovnih operacija (umetanja, pretrage) je logaritamska i tada je opasnost od prekoračenja programskog steka jako mala, te se u implementaciji mogu koristiti i rekurzivne funkcije.

Iterativni postupak može se izraziti na sledeći način:

- Uporedi vrednost v sa vrednošću korena; ako je vrednost v manja od ili jednaka vrednosti korena, idi levo, inače idi desno;
- Nastavi sa poređenjem vrednosti v sa vrednostima čvorova stabla sve dok ne dođeš do nepostojećeg čvora; dodaj novi čvor na to mesto.

Implementacija u može se napraviti na sledeći način:

```
status dodaj_u_stablo(cvor_stabla** pokazivac_na_koren,
                      int podatak)
{
  cvor_stabla* tmp;
  cvor_stabla* novi = novi_cvor_stabla(podatak);
  if (novi == NULL)
    return NEDOVOLJNO_MEMORIJE;
  novi->levo = NULL;
  novi->desno = NULL;

  if (*pokazivac_na_koren == NULL) {
    *pokazivac_na_koren = novi;
    return OK;
  }

  tmp = *pokazivac_na_koren;
  while (1) {
    if (podatak <= tmp->podatak) {
      if (tmp->levo == NULL) {
          tmp->levo = novi;
        return OK;
      }
      else
        tmp = tmp->levo;
    }
    else {
      if (tmp->desno == NULL) {
          tmp->desno = novi;
          return OK;
      }
      else
        tmp = tmp->desno;
```

```
    }
  }
  return OK;
}
```

Pronalaženje elementa u uređenom binarnom stablu

Pretraživanje uređenog binarnog stabla (traženje čvora koji ima vrednost v) je jednostavno i slično umetanju elementa u stablo. U slučaju da su elementi levo od korena manji od njega ili mu jednaki, a desno veći, pretraživanje se realizuje na sledeći način (analogno ako su elementi levo od korena veći od njega ili mu jednaki):

1. Ukoliko je tekuće stablo prazno, vrati rezultat da ne postoji traženi element u stablu.
2. Počni pretraživanje od korena stabla.
3. Ukoliko je v jednako vrednosti u tekućem čvoru stabla, vrati tekući čvor.
4. Inače, ukoliko je v manje od vrednosti u tekućem čvoru stabla, pretraži levo podstablo.
5. Inače, pretraži desno podstablo.

Naredna rekurzivna funkcija implementira opisani način za pronalaženje elementa u stablu:

```
cvor_stabla* pronadji_cvor(cvor_stabla* koren, int podatak)
{
  if (koren == NULL)
    return NULL;
  if (podatak < koren->podatak)
    return pronadji_cvor(koren->levo, podatak);
  else if (podatak > koren->podatak)
    return pronadji_cvor(koren->desno, podatak);
  else
    return koren;
}
```

Ako se u stablu dopuštaju duplikati (ako stablo čuva multiskup elemenata), tada funkcija vraća pokazavač na jedan od čvorova koji sadrži traženu vrednost (a koji od njih – zavisi od strukture stabla).

Nije teško eliminisati rekurziju iz ove funkcije.

6.6.2 Obilazak binarnog stabla

Elementi binarnog stabla (ne obavezno uređenog) mogu biti obrađeni na sistematičan način. Obrada redom svih čvorova stabla naziva se *obilazak stabla*. Postoje dve suštinske različite sistematične metode za obilazak stabla (kao, uostalom, i proizvoljnog grafa tj. proizvoljnog skupa čvorova povezanih granama). To su *obilazak u dubinu* i *obilazak u širinu*.

Obilazak binarnog stabla u dubinu

Obilazak binarnog stabla u dubinu je obilazak u kojem se kreće od korena stabla i nastavlja obrada čvorova duž jedne grane sve dok se ne naiđe na list, kada se vrši vraćanje unazad i obrada preostalih čvorova. Postoji nekoliko vrsta obilaska u dubinu (slika 6.9):

- infiksni: za svaki čvor se obilazi (rekurzivno) najpre njegovo levo podstablo, pa sâm taj čvor, pa njegovo desno podstablo (rekurzivno). Kratko to zapisujemo L-K-D (od *levo-koren-desno*);
- prefiksni: K-L-D;
- postfiksni: L-D-K.

Svaki od navedenih obilazaka može da se sprovede i u drugom smeru: zdesna nalevo (mada je to manje uobičajeno). Tako, na primer, infiksni obilazak može da bude i D-K-L.

Obilazak stabla u dubinu se često vrši rekurzivnim funkcijama. Za povezane liste je rečeno da je rekurzivni obilazak elemenata obično neprihvatljiv jer može da zahteva da se na programskom steku kreira onoliko stek okvira koliko ima elemenata liste. Kod stabla je, međutim, situacija drugačija: prilikom rekurzivnog obilaska na programskom steku se kreira onoliko stek okvira koliko je maksimalna dubina stabla. Ukoliko je stablo balansirano, taj broj je logaritamskog reda u odnosu na broj elemenata stabla, pa je u tom slučaju prekoračenje dubine steka malo verovatno. Alternativno, obilazak u dubinu može biti realizovan i iterativno uz korišćenje ručno upotrebljenog steka.

Ispis elemenata binarnog stabla u infiksnom obilasku može se rekurzivno implementirati na sledeći način:

```
void ispisi_stablo(cvor_stabla* p)
{
  if (p != NULL) {
    ispisi_stablo(p->levo);
    printf("%d\n", p->podatak);
    ispisi_stablo(p->desno);
  }
}
```

Brisanje stabla radi se u postfiksnom obliku, jer je pre brisanja samog čvora potrebno ukloniti sve njegove potomke.

```
void obrisi_stablo(cvor_stabla* p)
{
  if (p != NULL) {
    obrisi_stablo(p->levo);
    obrisi_stablo(p->desno);
    free(p);
  }
}
```

Obilazak uređenog binarnog stabla u dubinu

Obilazak binarnog stabla u dubinu može biti prefiksni, infiksni ili postfiksni, bez obzira na to da li je stablo uređeno ili nije. Ipak, kod uređenih binarnih stabala, obično se koristi infiksni obilazak jer se prilikom tog obilaska čvorovi stabla obilaze u neopadajućem redosledu vrednosti (tj. u sortiranom redosledu).

Navedeno svojstvo može se iskoristiti da se formuliše još jedan algoritam sortiranja koji koristi binarna stabla. Algoritam *tree sort* je varijacija algoritma sortiranja umetanjem u kojoj se jedan po jedan element umeće u uređeno binarno stablo, a zatim se stablo obilazi infiksno, čime se dobijaju elementi u uređenom redosledu. Ako je operacija umetanja u stablo složenosti $O(\log n)$ (ako je stablo samobalansirajuće), složenost ovog algoritma sortiranja je $O(n \log n)$. Međutim, koristi se dodatna memorija za alokaciju čvorova i narušava se lokalnost pristupa keš memoriji jer čvorovi ne moraju biti smešteni uzastopno, pa iako imaju istu asimptotsku složenost, ovaj algoritam ima nešto lošije performanse od ranije opisanih algoritama sortiranja složenosti $O(n \log n)$. S druge strane, ovaj pristup dopušta da se i nakon sortiranja efikasno dodaju elementi, pri čemu se zadržava sortirani redosled (što u slučaju niza nije moguće ostvariti).

Obilazak binarnog stabla u širinu

U obilasku stabla u širinu, obilazi se (i obrađuje) nivo po nivo stabla. Obilazak u širinu može se implementirati korišćenjem strukture red koja ima interfejs opisan u poglavlju 6.5, a čiji su elementi pokazivači na čvorove stabla (takav red može biti zasnovan na generičkom redu, opisanom u poglavlju 6.5.5). Obilazak stabla u širinu korišćenjem reda može se grubo opisati na sledeći način: na kraj (tada praznog) reda najpre se stavlja koren stabla; zatim se sa početka reda uklanja i obrađuje jedan po jedan element a na kraj reda se stavljanju njegovi potomci; postupak se nastavlja sve dok red nije prazan. Koristeći opisanu ideju, obilazak binarnog stabla u širinu (u ovom slučaju, obrada je jednostavni ispis) može se implementirati na sledeći način:

```
int ispisi_stablo_po_sirini(cvor_stabla* koren)
{
  Red red;
  inicijalizuj_red(&red);
  if (enqueue(&red, koren) != OK)
    return -1;
  void* p;
  while (dequeue(&red, &p) == OK) {
    cvor_stabla* c = (cvor_stabla*)p;
    printf("%d ", c->podatak);
    if (c->levo != NULL)
      if (enqueue(&red, c->levo) != OK)
        return -1;
    if (c->desno != NULL)
      if (enqueue(&red, c->desno) != OK)
        return -1;
  }
  return 0;
}
```

Naredna funkcija `main`

```
/* ... */

int main()
{
  cvor_stabla* koren = NULL;
  if (dodaj_u_stablo(&koren, 3) != OK) {
    obrisi_stablo(koren);
    return -1;
  }
  if (dodaj_u_stablo(&koren, 5) != OK) {
    obrisi_stablo(koren);
    return -1;
  }
  if (dodaj_u_stablo(&koren, 4) != OK) {
    obrisi_stablo(koren);
    return -1;
  }
  if (dodaj_u_stablo(&koren, 2) != OK) {
    obrisi_stablo(koren);
    return -1;
  }
  if (dodaj_u_stablo(&koren, 1) != OK) {
```

```
    obrisi_stablo(koren);
    return -1;
  }
  ispisi_stablo_po_sirini(koren);
  obrisi_stablo(koren);
  return 0;
}
```

onda daje sledeći izlaz:

```
3 2 5 1 4
```

6.6.3 Primer primene strukture binarno stablo: izrazi u formi stabla

Matematički izrazi prirodno se mogu predstaviti u vidu stabla. Na primer, izraz `3*(4+5)` može se predstaviti kao stablo u čijem je korenu `*`, sa levim podstablom `3`, i sa desnim podstablom u čijem je korenu čvor `+` i koji ima neposredne potomke `4` i `5`. Ispisivanjem elemenata ovog stabla u infiksnom obilasku (sa dodatnim zagradama) dobija se izraz u uobičajenoj formi. Ispisivanjem elemenata ovog stabla u prefiksnom obilasku dobija se izraz zapisan u takozvanoj prefiksnoj poljskoj notaciji. Vrednost izraza predstavljenog stablom može se izračunati jednostavnom funkcijom.

6.6.4 Generičko uređeno binarno stablo

Slično kao što je u poglavljima 6.4.4 i 6.5.5 opisano za generički stek i generički red, i stablo se može implementirati tako da, samo uz trivijalne izmene, može da barata sa različitim tipovima podataka.

U datoteci `stablo_podatak.h` može biti uveden tip `Data` (u sledećem jednostavnom primeru – `int`) i deklarisane osnovne funkcije za baratanje tim tipom podataka:

```
#ifndef __STABLO_PODATAK_H
#define __STABLO_PODATAK_H

typedef int Data;

int poredi(const Data *d1, const Data *d2);
int dodeli(Data *d1, const Data *d2);
void ispisi_podatak(const Data *d);

#endif
```

Te funkcije definisane su u datoteci `stablo_podatak.c`:

```c
#include <stdio.h>
#include "stablo_podatak.h"

int poredi(const Data *d1, const Data *d2)
{
  if (*d1 < *d2)
    return -1;
  if (*d1 > *d2)
    return 1;
  return 0;
}

int dodeli(Data *d1, const Data *d2)
{
  *d1 = *d2;
}

void ispisi_podatak(const Data *d)
{
    printf("%d ", *d);
}
```

Naredna implementacija generičkog binarnog uređenog stabla koristi tip `Data`, definiše tip `Stablo` i svega nekoliko jednostavnih funkcija. Sadržaj datoteke `stablo_genericko.h` može biti ovakav:

```c
#ifndef __STABLO_GENERICKO_H__
#define __STABLO_GENERICKO_H__

#include <stdio.h>
#include <stdlib.h>
#include "stablo_podatak.h"

typedef enum {
   OK,
   NEDOVOLJNO_MEMORIJE,
   NEMA_ELEMENATA
} status;

typedef struct cvor_stabla {
  Data podatak;
  struct cvor_stabla* levo;
  struct cvor_stabla* desno;
} cvor_stabla;
```

```
typedef cvor_stabla* Stablo;

status inicijalizuj_stablo(Stablo* s);
status dodaj_u_stablo(Stablo* s, const Data* podatak,
                      int (*poredi)(const Data*, const Data*));
status pronadji(Stablo s, const Data* trazeno, Data** nadjeno,
                int (*poredi)(const Data*, const Data*));
int broj_elemenata(Stablo s);
void ispisi_stablo(Stablo s);
void obrisi_stablo(Stablo* s);

#endif
```

Sadržaj datoteke `stablo_genericko.c` može biti ovakav:

```
#include <stdio.h>
#include <stdlib.h>
#include "stablo_genericko.h"

cvor_stabla* novi_cvor_stabla(const Data* podatak)
{
  cvor_stabla* novi = malloc(sizeof(cvor_stabla));
  if (novi == NULL)
    return NULL;
  dodeli(&(novi->podatak), podatak);
  novi->levo = NULL;
  novi->desno = NULL;
  return novi;
}

status inicijalizuj_stablo(Stablo* s)
{
  *s = NULL;
  return OK;
}

status dodaj_u_stablo(Stablo* s, const Data* podatak,
                      int (*poredi)(const Data*, const Data*))
{
  cvor_stabla* tmp;
  cvor_stabla* novi = novi_cvor_stabla(podatak);
  if (novi == NULL)
    return NEDOVOLJNO_MEMORIJE;
  novi->levo = NULL;
```

```
  novi->desno = NULL;

  if (*s == NULL) {
    *s = novi;
    return OK;
  }
  tmp = *s;
  while (1) {
    if (poredi(podatak, &(tmp->podatak)) <= 0) {
      if (tmp->levo == NULL) {
          tmp->levo = novi;
        return OK;
      }
      else
        tmp = tmp->levo;
    }
    else {
      if (tmp->desno == NULL) {
          tmp->desno = novi;
        return OK;
      }
      else
        tmp = tmp->desno;
    }
  }
  return OK;
}

status pronadji(Stablo s, const Data* trazeno, Data** nadjeno,
                int (*poredi)(const Data*, const Data*))
{
  if (s == NULL)
    return NEMA_ELEMENATA;
  if (poredi(trazeno, &(s->podatak)) == 0) {
    *nadjeno = &(s->podatak);
    return OK;
  }
  if (poredi(trazeno, &(s->podatak)) < 0)
    return pronadji(s->levo, trazeno, nadjeno, poredi);
  return pronadji(s->desno, trazeno, nadjeno, poredi);
}

int broj_elemenata(Stablo s)
{
```

```
  return s == NULL ?
           0 :
           broj_elemenata(s->levo)+1+broj_elemenata(s->desno);
}

void ispisi_stablo(Stablo s)
{
  if (s != NULL) {
     ispisi_stablo(s->levo);
     ispisi_podatak(&(s->podatak));
     ispisi_stablo(s->desno);
  }
}

void obrisi_stablo(Stablo* s)
{
  if (*s != NULL) {
    obrisi_stablo(&((*s)->levo));
    obrisi_stablo(&((*s)->desno));
    free(*s);
    *s = NULL;
  }
}
```

Jedan primer korišćenja ove generičke implementacije binarnog uređenog stabla biće dat u poglavlju 6.7.

6.7 Skup i mapa

U mnogim praktičnim problemima imamo potrebu da u programu održavamo neki *skup* podataka, tako da tokom rada programa taj skup možemo proširivati dodavanjem novih elemenata, sužavati uklanjanjem postojećih elemenata i tako da efikasno možemo proveravati da li se neka vrednost nalazi u skupu.

Mapa (kaže se i *asocijativni niz* ili *rečnik*) je struktura koja čuva parove podataka, u kojima prvi element služi kao *ključ*, a drugi mu je neka pridružena vrednost (na primer, ključ može da bude JMBG broj građana, a pridružena vrednost ime i prezime). Obično se podacima pristupa preko ključa i važno je da možemo efikasno vršiti pridruživanje i pretragu vrednosti na osnovu zadatog ključa.

Ako bi se skup i mapa realizovali pomoću niza podataka u koji se podaci upisuju redom, u redosledu u koji se umeću u skup tj. mapu, dodavanje bi bilo konstantne složenosti, ali bi pretraga bila linearne složenosti (po broju elemenata). Složenost pretrage bi mogla biti logaritamska, ako bi se podaci čuvali u sortiranom redosledu ključeva, ali bi tada umetanje bilo linearno, jer bi se svaki novi podatak morao upisati na odgovarajuće mesto u sortiranom nizu, što

bi dovelo do pomeranja svih podataka iza tog mesta. U nastavku ovog poglavlja videćemo da se skupovi i mape efikasno mogu implementirati korišćenjem binarnih uređenih stabala. Umetanje, brisanje i pretraga onda mogu da budu logaritamske složenosti (po broju elemenata), ali samo ako implementacija binarnog stabla omogućava dodavanje i brisanje logaritamske složenosti. Jednostavna implementacija binarnog stabla koja se koristi u nastavku nema to svojstvo ali ipak može da posluži kao dobra ilustracija (jer bi i u efikasnijoj implementaciji deklaracije funkcija koje se koriste za skup i mapu bile iste).

Ako čvor binarnog uređenog stabla sadrži samo jedan podatak, stablom se implementira skup (tj. multiskup, u slučaju da je dopušteno da više čvorova sadrži istu vrednost). Kada se pomoću stabla implementira mapa, u svakom čvoru stabla se čuva ključ i njemu pridružena vrednost, pri čemu se poređenje čvorova vrši na osnovu ključeva (a ne njima pridruženih vrednosti).

6.7.1 Primer primene mape: brojanje reči

Razmotrimo sledeći zadatak: sa standardnog ulaza čitaju se reči i broje se njihova pojavljivanja. Zadatak se može rešiti korišćenjem mape koja sadrži parove reči i broja pojavljivanja. U narednoj implementaciji, nije definisana nova struktura za mapu, ali je raspoloživa implementacija generičkog stabla (poglavlje 6.6.4) korišćena u duhu mape. Koriste se funkcije za traženje ključa u stablu, dodavanje novog para, za ispis svih elemenata i za brisanje svih elemenata. U programu se vrši traženje datog ključa, pa ako on ne postoji – onda se vrši dodavanje novog elementa u stablo. Te dve funkcije mogle bi da budu objedinjene u jednu funkciju koja samo jednom prolazi kroz stablo. Složenost nalaženja i dodavanja reči zavisi od implementacije tih operacija u stablu. Korišćena implementacija (poglavlje 6.6.4) je jednostavna i ne obezbeđuje željeno logaritamsko vreme (u odnosu na broj elemenata) za ove operacije, ali i dalje dobro ilustruje način korišćenja uređenog binarnog stabla.

Naredna datoteka `stablo_podatak_r.h` treba da bude uključena u datoteku `stablo_genericko.h` (iz poglavlja 6.6.4). Ona definiše tip para koji se čuva u „mapi“ (tj. u odgovarajućem stablu).

```
#ifndef __STABLO_PODATAK_R_H
#define __STABLO_PODATAK_R_H

#define MAX_REC 100

typedef struct
{
  char rec[MAX_REC + 1];
  int broj_pojavljivanja;
} Data;
```

```
int poredi(const Data *d1, const Data *d2);
int dodeli(Data *d1, const Data *d2);
void ispisi_podatak(const Data *d);

#endif
```

U datoteci `stablo_podatak_r.c` definisano je nekoliko osnovnih operacija nad ovim parovima. Na primer, poređenje se vrši po prvom elementu (koji je niz karaktera).

```
#include <stdio.h>
#include <string.h>
#include "stablo_podatak_r.h"

int poredi(const Data *d1, const Data *d2)
{
  return strcmp(d1->rec, d2->rec);
}

int dodeli(Data *d1, const Data *d2)
{
  strcpy(d1->rec,d2->rec);
  d1->broj_pojavljivanja = d2->broj_pojavljivanja;
}

void ispisi_podatak(const Data *d)
{
    printf("[%s,%i] ", d->rec, d->broj_pojavljivanja);
}
```

Datoteka sa funkcijom `main` koja koristi podršku za „mape“ i koja broji pojavljivanja reči sa standardnog ulaza.

```
#include <stdio.h>
#include <string.h>
#include "stablo_genericko.h"

int main() {
  char rec[MAX_REC + 2];
  Stablo s;
  if (inicijalizuj_stablo(&s) != OK)
    return -1;

  while (fgets(rec, MAX_REC + 2, stdin) != NULL) {
    rec[strcspn(rec, "\n")] = '\0';
```

```
    Data trazeno, *nadjeno;
    strcpy(trazeno.rec, rec);
    if (pronadji(s, &trazeno, &nadjeno, poredi) == OK)
      nadjeno->broj_pojavljivanja++;
    else {
      trazeno.broj_pojavljivanja = 1;
      if (dodaj_u_stablo(&s, &trazeno, poredi) != OK) {
        fprintf(stderr, "Greska u radu sa stablom.\n");
        return -1;
      }
    }
  }
  ispisi_stablo(s);
  obrisi_stablo(&s);
  printf("\n");
  return 0;
}
```

Navedeni program može da se kompilira na sledeći način:

```
gcc main_prebrojavanje_reci.c stablo_podatak_r.c
  stablo_genericko.c -o prebrojavanje_reci
```

Ako se programu sa ulaza zada:

```
Ceca
Boban
Aca
Aca
Aca
Boban
```

dobija se izlaz (primetimo da su reči ispisane u sortiranom poretku):

```
[Aca,3] [Boban,2] [Ceca,1]
```

Pitanja i zadaci za vežbu

Pitanje 6.23. *Šta je to uređeno binarno stablo?*

Pitanje 6.24. *Ako binarno stablo ima* 15 *čvorova, koja je njegova najmanja moguća a koja najveća moguća dubina?*

Pitanje 6.25. *Ako binarno stablo ima dubinu* 9*, koliko najviše čvorova ono može da ima?*

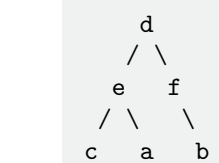
Pitanje 6.26.
(a) *Koja je složenost dodavanja novog elementa u uređeno binarno stablo (po analizi najgoreg slučaja)?*
(b) *Koja je složenost pronalaženja elementa u uređenom binarnom stablu (po analizi najgoreg slučaja)?*
(c) *Da li se i kako te složenosti mogu poboljšati?*

Pitanje 6.27. *Složenost operacije pronalaženja elementa u uređenom binarnom stablu zavisi od strukture stabla. Kada je ta složenost najmanja a kada najveća (i kolika je ona tada)?*

Pitanje 6.28. *Koja je složenost dodavanja elementa u uređeno binarno stablo koje ima k elemenata, a n nivoa?*

Pitanje 6.29. *Koja dodatna struktura podataka i na koji način se koristi prilikom nerekurzivne implementacije obilaska stabla u širinu? A u dubinu?*

Pitanje 6.30. *Dato je stablo:*



Ispisati elemente stabla koristeći:
(a) *Prefiksni obilazak (K-L-D);*
(b) *Infiksni obilazak stabla (L-K-D);*
(c) *Postfiksni obilazak stabla (L-D-K);*
(d) *Obilazak stabla u širinu.*

Pitanje 6.31. *Da li se postfiksnim ispisivanjem elemenata binarnog stabla uvek dobija isti niz elemenata koji se dobija prefiksnim obilaskom, ali u obratnom poretku? Da li se to ponekad može dogoditi?*

Pitanje 6.32. *Kako se predstavljaju izrazi u vidu stabala?*

Pitanje 6.33. *Binarnim stablom predstaviti izraz* $2 + 3 * (4 + 2)$.
Prikazati u vidu stabla izraz koji ima sledeći prefiksni zapis: $*+* a\ 3\ +\ b\ 4\ c$.
Prikazati u vidu stabla izraz koji ima sledeći prefiksni zapis: $+** x\ 2\ +\ x\ 3\ 7$.

Pitanje 6.34. *Nacrtati stablo za izraz* $y = x \cdot 5 < 3 + 4$ *u skladu sa prioritetom operacija jezika C i napisati šta se dobija njegovim postfiksnim obilaskom.*

Pitanje 6.35. *Opisati strukturu podataka kojom se može predstaviti struktura direktorijuma i datoteka u operativnom sistemu.*

Zadatak 6.36. *Definisati funkciju koja:*
(a) *izračunava zbir svih elemenata u binarnom stablu koje sadrži cele brojeve;*
(b) *određuje broj elemenata u stablu;*
(c) *određuje visinu stabla;*
(d) *određuje najveći element u stablu (ono ne mora biti uređeno);*
(e) *pravi kopiju tog binarnog stabla;*
(f) *transformiše stablo tako što svakom čvoru razmenjuje levo i desno podstablo;*
(g) *ispisuje sve elemente na datom nivou stabla (koren je na nivou 0, njegovi sinovi na nivou 1, njihovi sinovi na nivou 2 itd.);*
(h) *određuje broj čvorova koji su po sadržaju veći od svih svojih potomaka;*
(i) *ispisuje čvorove koji su veći od zbira vrednosti svih svojih potomaka;*
(j) *proverva da li su dva data binarna stabla identična (po strukturi i sadržaju);*
(k) *ispisuje sve listove (to su čvorovi koji nemaju ni jednog potomka);*
(l) *određuje čvor koji je najbliži korenu i pri tom je deljiv sa 3 (ako ima više takvih čvorova, vratiti bilo koji);*
(m) *proverava da li dato binarno stablo ima simetričnu strukturu (postoji vertikalna osa simetrije).*

Zadatak 6.37. *Definisati funkciju koja ispisuje sve putanje od korena do lista u datom binarnom stablu.*

Zadatak 6.38. *Definisati funkciju koja briše datu vrednost iz datog binarnog stabla.*

Zadatak 6.39. *Definisati funkciju koja u uređenom binarnom stablu pronalazi najmanju vrednost koja je veća od ili jednaka datoj.*

Zadatak 6.40. *Definisati funkciju koja za dva data čvora binarnog stabla određuje njihovog najnižeg zajedničkog pretka.*

Definisati funkciju koja određuje najkraće rastojanje (broj grana) između dva data čvora binarnog stabla.

Zadatak 6.41. *Definisati funkciju kojom se proverava da je jedno dato binarno stablo podstablo drugog datog binarnog stabla.*

Zadatak 6.42.
(a) *Definisati funkciju koja ispisuje elemente binarnog stabla po nivoima (prvo koren, pa njegova dva sina, pa njihove sinove itd).*
(b) *Definisati funkciju koja ispisuje elmente binarnog stabla po nivoima, ali tako da se elementi na parnim nivoima ispisuju sleva nadesno, a na neparnim zdesna nalevo.*
(c) *Definisati funkciju koja ispisuje elemente binarnog stabla po nivoima, tako što se prvo ispiše poslednji nivo, zatim pretposlednji itd., dok se koren ispisuje poslednji.*
(d) *Definisati funkciju koja ispisuje čvorove koji bi se videli kada bi se binarno*

stablo posmatralo s leve strane (to su početni čvorovi svakog nivoa).
(e) *Potpuno binarno stablo je ono stablo u kom su svi nivoi, osim eventualno poslednjeg potpuno popunjeni, dok je poslednji nivo takav da mu je popunjen neki prefiks iza koga nema više elemenata. Definisati funkciju koja proverava da li je dato binarno stablo potpuno.*
(f) *Potpuno binarno stablo se može predstaviti nizom, tako što se elementi tog niza popunjavaju redom, po nivoima stabla. Definisati funkciju koja na osnovu takvog niza gradi stablo predstavljeno čvorovima i pokazivačima i funkciju koja na osnovu potpunog stabla predstavljenog čvorovoima i pokazivačima formira niz.*

Zadatak 6.43. *Napisati program koji za uređeno binarno stablo ispisuje elemente na najvećoj dubini (elemente najdalje od korena).*

Zadatak 6.44. *Definisati funkciju linearne složenosti koja proverava da li je dato binarno stablo uređeno. Obratiti pažnju na to da nije dovoljno samo uporediti da li je levi sin manji, a desni veći od oca.*

Zadatak 6.45. *Napisati program koji formira uređeno binarno stablo bez ponavljanja elemenata. Elementi stabla su celi brojevi i unose se sa ulaza, a oznaka za kraj unosa je* 0.
(a) *Definisati funkciju linearne složenosti koja proverava da li je uneto stablo* uravnotežene visine. *Stablo je uravnotežene visine ako za svaki čvor stabla važi da mu se visina levog i desnog podstabla razlikuju najviše za jedan.*
(b) *Definisati funkciju linearne složenosti koja proverava da li je uneto stablo* uravnotežene težine. *Stablo je uravnotežene težine ako za svaki čvor stabla važi da mu se broj čvorova u levom i desnom podstablu razlikuju najviše za jedan.*

Zadatak 6.46. *Definisati funkciju linearne složenosti koja određuje dijametar binarnog stabla (najveći broj grana na putu između dva lista).*

Zadatak 6.47. *Korišćenjem binarnog stabla implementirati program koji određuje koliko ima različitih među n učitanih celih brojeva.*

Zadatak 6.48. *Napisati program koji određuje 10 peči koje se najčešće javljaju u datoj datoteci.*

Zadatak 6.49. *U datoteci* `kupci.txt` *se nalaze podaci o kupcima oblika* `ime kolicina` *gde je* `ime` *ime kupca koji je kupio proizvod, a* `kolicina` *količina proizvoda. Pretpostavlja se da je datoteka dobro strukturirana. Isti kupac se može više puta pojaviti u datoteci. Kreirati uređeno binarno stablo prema imenima kupaca. Na standarni izlaz ispisati ime onog kupca koji je kupio najviše proizvoda.*

Zadatak 6.50. *Napisati program koji formira uređeno binarno stablo bez ponavljanja elemenata čiji elementi su imena studenata i brojevi njihovih indeksa (struktura). Pretpostavka je da ime studenta nije duže od* 30 *karaktera i da je indeks dat kao ceo broj. Korisnik programa u svakom koraku može da unese podatak o novom studentu, proveri da li postoji student sa datim brojem indeksa i ako postoji ispiše njegovo ime i ispiše trenutni broj studenata.*

Zadatak 6.51. *Sa standardnog ulaza se unosi ispravno zapisan aritmetički izraz koji sadrži prirodne brojeve, operatore sabiranja i množenja i zagrade. Definiši funkciju koja tehnikom rekurzivnog spusta učitava izraz i predstavlja ga binarnim stablom. Definiši funkciju koja na osnovu stabla izračunava vrednost izraza. Ispistati izračunatu vrednost na standardni izlaz.*

Zadatak 6.52. *Celobrojni aritmetički izraz predstavljen je binarnim stablom. Izraz može da sadrži konstante, promenljive i binarne operatore* + *i* ∗*. Definisati funkciju koja izračunava vrednost izraza, pri čemu se vrednosti svih promenljivih prosleđuju u mapi (mapa je predstavljena binarnim stablom).*

Zadatak 6.53.
(a) *Definisati funkciju u izrazu opisanom stablom svako pojavljivanje date promenljive zamenjuje datom konstantom.*
(b) *Definisati funkciju koja ispituje da li izraz opisan stablom sadrži promenljive.*
(c) *Definisati funkciju izračunava vrednost izraza opisanog stablom (ako izraz ima promenljivih, funkcija treba da signalizira grešku).*
Podrazumevati da je stablo izraza koje se prosleđuje funkcijama ispravno konstruisano.

Zadatak 6.54. *Definisati funkciju koja uprošćava stablo kojim se predstavlja aritmetički izraz nad celim brojevima tako što primenjuje algebarske identitete* $x + 0 = x$, $x \cdot 0 = 0$, $x \cdot 1 = x$, *izračunava vrednost konstantnih izraza i slično (na primer,* $x \cdot (-3 + 3)$ *se uprošćava u* 0*).*

Deo III

Osnove programskih jezika

Glava 7

Programski jezici i paradigme

Programski jezici nastaju, razvijaju se i nestaju već više od sedamdeset godina. Neki izvori navode da je u realnoj upotrebi bilo do sada oko 250 jezika, a neki izvori koji pretenduju da popišu sve programske jezike koji su ikad postojali navode skoro 9000 konkretnih jezika. I dalje nema „najboljeg programskog jezika“, i dalje se za različite potrebe biraju različiti jezici, i dalje se razvijaju jezici sa novim svojstvima.

U daljem tekstu, pomenuti su samo neki od mnogih do sada razvijenih programskih jezika – oni koji su ostavili najdublji trag u programiranju, kao i najznačajnije programske paradigme. Pored programskih jezika opšte namene, postoje i namenski jezici – jezici specijalne namene, kao, na primer, jezici za rad sa bazama podataka, za rad sa raširenim tabelama, za statističke analize, za matematička izračunavanja, itd, ali oni nisu opisivani u ovom pregledu.

7.1 Najznačajniji programski jezici kroz istoriju

Programiranje prvih elektronskih računara 1940-tih godina vršeno je isključivo korišćenjem mašinskog i asemblerskog jezika. Iako su prvi računari bili skupoceni, mnoge kompanije su još više novca trošile na razvoj softvera, zbog kompleksnosti programiranja na asembleru. Već u to vreme postojala je ideja o razvoju apstraktnijih programskih jezika koji bi automatski bili prevođeni na mašinski jezik.

U nastavku navodimo kratki pregled istorijski najznačajnijih programskih jezika. Nisu svi oni sada najpopularniji jezici, neki od njih nikada i nisu bili, ali su ostvarili snažan uticaj na razvoj programiranja i razvoj programskih jezika.

Plankalkül: Konrad Cuze (nem. Konrad Zuse) je 1948. godine definisao prvi viši programski jezik – *Plankalkül*. Jezik je trebalo da se koristi na njegovom računaru Z3, ali odgovarajući prevodilac nikada nije implementiran.

FORTRAN: Prvi viši programski jezik koji je stekao širok krug korisnika je programski jezik FORTRAN (od „FORmula TRANslating System“), na-

stao u periodu 1953-1957 godine. Vođa tima u okviru kompanije IBM koji je projektovao ovaj jezik, bio je Džon Bakus (engl. John Backus). Prvi interpretator za FORTRAN bio je razvijen 1953. godine. Programiranje je postalo brže, ali novi programi su se izvršavali 10-20 puta sporije nego programi napisani na asembleru. Početna verzija kompilatora za FORTRAN I objavljena je nekoliko godina kasnije – 1956. godine i imala je oko 25000 linija asemblerskog koda. Kompilirani programi izvršavali su se skoro jednako brzo kao programi ručno pisani na asembleru. Pisanje programa ubrzano je i po 40 puta. Znatno je olakšano i održavanje programa zbog bolje čitljivosti i omogućena je prenosivost između različitih računara (za koje su postojali razvijeni FORTRAN kompilatori). Već 1958. više od polovine svih programa pisano je na FORTRAN-u. Ovaj jezik se, uz velike izmene u odnosu na prvobitne verzije, i danas koristi i namenjen je, pre svega, za numerička i naučna izračunavanja.

LISP: Programski jezik *LISP* razvio je Džon Makarti (engl. John McCarthy) 1958. godine na univerzitetu MIT. Ime LISP nastalo je od *LISt Processing*, jer jezik podržava liste i stabla kao osnovne strukture podataka. LISP je smatran jezikom veštačke inteligencije. Zasnovan je na Čerčovom formalizmu λ-računa i spada u grupu funkcionalnih programskih jezika. Od LISP-a su se dalje razvili svi savremeni funkcionalni programski jezici. Neke njegove varijante se i danas koriste, ali je u polju funkcionalnih jezika potisnut od modernijih rođaka poput jezika Scheme, ML, OCaml i Haskell.

COBOL: Programski jezik COBOL (od „COmmon Business Oriented Language“) razvijen je 1959. godine zajedničkom inicijativom nekoliko vodećih kompanija i univerziteta da se napravi jezik pogodan za izradu poslovnih, finansijskih i administrativnih aplikacija. Mnoge aplikacije pisane u COBOL-u su i danas u upotrebi, ali se COBOL više ne koristi za razvoj novih aplikacija.

ALGOL i Pascal: Jedan od najuticajnijih programskih jezika je imperativni, proceduralni, strukturirani programski jezik ALGOL (od „ALGOrithmic Language“), nastao i razvijan od kraja pedesetih do početka sedamdesetih godina prošlog veka. U njegov razvoj bili su uključeni mnogi znameniti informatičari iz Evrope i Amerike. Jedan od naslednika jezika ALGOL je jezik Pascal, koji je 1970. godine dizajnirao švajcarski informatičar Niklaus Virt (nem. Niklaus Wirth) kao mali i efikasan jezik koji ohrabruje korišćenje strukturiranog programiranja i drugih dobrih praksi programiranja. Jezik Pascal bio je veoma popularan tokom osamdesetih i devedesetih godina prošlog veka, ali je ipak smatrano da je njegovo glavno polje nastava programiranja (a ne i industrijske primene). Zbog svojih slabosti, jezik je bio česta meta kritika, posebno zagovornika jezika C. Unapređenja i modifikacije jezika Pascal dovele su do jezika Modula, Oberon i Modula-

2 (koje je, osamdesetih godina prošlog veka, razvio takođe Virt), kao i do objektno-orijentisanog jezika Object Pascal.

BASIC: Jezic BASIC (od „Beginner's All-Purpose Symbolic Instruction Code") inicijalno je razvijen (1964. godine) za početnike u programiranju, za koje su FORTRAN i ALGOL bili previše kompleksni. Jezik je bio ekstremno jednostavan – prva verzija imala je samo četrnaest naredbi. Popularnost jezika porasla je sa pojavom mikro-računara i personalnih računara. U međuvremenu je razvijeno mnogo njegovih modifikacija i proširenja, od kojih je trenutno najpopularniji Visual Basic.

C i C++: Programski jezik C je programski jezik opšte namene koji je 1972. godine razvio Denis Riči[1] u Belovim telefonskim laboratorijama (engl. Bell Telephone Laboratories) u SAD. Ime C dolazi od činjenice da je jezik nastao kao naslednik jezika B (a koji je bio pojednostavljena verzija jezika BCPL). C je jezik koji je bio namenjen prevashodno pisanju sistemskog softvera i to u okviru operativnog sistema Unix. Međutim, vremenom je počeo da se koristi i za pisanje aplikativnog softvera na velikom broju drugih platformi. C je danas prisutan na širokom spektru platformi — od mikrokontrolera do superračunara. Jezik C uticao je i na razvoj drugih programskih jezika. Najznačajniji direktni naslednik jezika C je jezik C++ koji se, u trenutku nastanka, mogao smatrati njegovim objektno-orijentisanim proširenjem. Kreirao ga je Bjern Stroustrup (danski informatičar) 1986. godine. C++ je i dalje jedan od najpopularnijih jezika i koristi se za razvoj zahtevnih aplikacija, s jedne strane zbog svojih objektno-orijentisanih svojstava, a s druge zbog bliske veze sa mašinom, u duhu jezika C. U izvesnom smislu, potomkom i unapređenjem jezika C++ može se smatrati jezik C#, koji je razvijen 2000. godine.

Perl: Jezik Perl razvijen je sredinom osamdesetih godina prošlog veka. Jedna od njegovih ključnih karakteristika su moćne funkcije za obradu teksta, a koristio se u radu sa tekstualnim podacima, sa bazama podataka, u mrežnom programiranju i slično, ali i kao skript jezik (jezik za izvršno okruženje na kojem se zadaje automatizovano izršavanje zadataka) za Linux sisteme.

Python: Pajton (eng. Python) je multiparadigmatski jezik jednostavne i izražajne sintakse koji se interpretira (prva verzija je objavljena 1991. godine). Ima elemente objektno-orijentisanih, imperativnih, funkcionalnih jezika itd. Raspolaže ugrađenim strukturama podataka visokog nivoa. Pošto se programi interpretiraju, osnovni ciklus razvoja programa (pisanje-testiranje-debagovanje) odvija se izuzetno brzo. Pajton je najpopularniji jezik u oblastima kao što su istraživanje podataka i mašinsko učenje, ali

[1] Dennis Ritchie (1941–2011), američki informatičar, dobitnik Tjuringove nagrade 1983. godine.

se koristi i za veb-aplikacije, za mobilne aplikacije, kao i za ugrađene sisteme. U nastavi je Pajton na mnogim mestima kao prvi programski jezik zamenio Pascal, Javu i C.

Java: Java je objektno-orijentisani programski jezik razvijen 1995. godine sa motivom da bude što manje zavisnosti za fazu izvršavanja i da se kompilirani Java kôd (takozvani bajtkod) može izvršavati na bilo kojoj platformi koja podržava Javu (tj. koja raspolaže Java virtuelnom mašinom) bez ponovnog kompiliranja. Sintaksa jezika Java slična je jezicima C i C++ ali ima manje operacija niskog nivoa. Java omogućava modifikacije koda u fazi izvršavanja. Java je trenutno jedan od najpopularnijih programskih jezika.

C#: Jezik C# je razvijen oko 2000. godine u kompaniji Micorosoft kao deo njihove .NET inicijative. Po svojim karakteristikama je donekle sličan programskom jeziku Java. Zamišljen kao moderni objektno-orijentisani programski jezik opšte namene koji se karakteriše relativnom jednostavnošću programiranja (poput jezika Java i ovaj jezik vrši automatsko upravljanje memorijom korišćenjem sakupljača otpadaka, pre pristupa elementima niza vrši se provera da li je indeks unutar granica niza, sve promenljive se inicijalizuju na podrazumevane vrednosti, disciplina tipova je prilično striktna i slično). Jezik se stalno obogaćuje i unapređuje i i dalje je veoma popularan izbor za programiranje aplikacija za Windows, veb, pa i mobilne aplikacije.

Internet skript jezici JavaScript, PHP, ASP: Skript jezik PHP (od „Hypertext Preprocessor“) kreiran je 1994. godine kao internet jezik koji može biti ugrađen u HTML kôd. PHP kôd se izvršava na serveru i dinamički generiše HTML sadržaj koji se šalje i prikazuje klijentu. Više od polovine svih veb sajtova kao jezik na strani servera koriste PHP u nekom obliku.

JavaScript je skript jezik, kreiran 1995. godine, koji se obično izvršava na strani klijenta (na primer, u okviru pregledača veba) i omogućava da se sadržaj veb strana menja interaktivno, na primer, u zavisnosti od akcija korisnika. JavaScript danas koristi većina veb strana. JavaScript i PHP mogu da se koriste skladno – prvi na strani klijenta, a drugi na strani servera.

ASP.NET, kreiran 2002. godine, je skript jezik za veb aplikacije i dinamičko kreiranje veb sadržaja kompanije Microsoft. Ima dosta sličnosti sa jezikom PHP, ali može da se izvršava samo na Windows serverima.

Kao što je već rečeno, ne postoji „najbolji programski jezik“, pa ni „najzastupljeniji programski jezik“ jer su polja i načini primene za neke jezike skoro neuporedivi. Indeks TIOBE (eng. TIOBE Programming Community index) još od davne 1985. godine meri popularnost nekog jezika kroz broj programera u svetu koji vladaju nekim jezikom, broj univerzitetskih kurseva, broj kompanija koje koriste neki jezik i slično. Za rangiranje se koristi više popularnih

pretraživača veba. Jezik C je uvek na prvom ili drugom mestu, od 2000. godine jezik Java je takođe uvek na prvom ili drugom mestu, C++ je uvek u prvih pet. U januaru 2021. godine poredak prvih osam jezika bio je ovakav: C, Java, Pajton, C++, C#, Visual Basic, JavaScript, PHP. Po zastupljenosti koda u određenom programskom jeziku na portalu `github`, poredak je skoro identičan. Među korisnicima foruma `stackoverflow` koji okuplja hiljade informatičara širom sveta, spisak najčešće korišćenih jezika u godini 2020, izgleda ovako: JavaScript, HTML/CSS, SQL, Python, Java, Bash/Shell, C#, PHP, TypeScript, C++, C. Ove liste, kao i svaku drugu treba, naravno, shvatiti vrlo uslovno.

7.2 Programske paradigme

U ovom poglavlju biće opisane neke klasifikacije programskih jezika. Ovakve klasifikacije nikada ne mogu biti oštre i isključive jer mnogi programski jezici dele osobine nekoliko različitih klasa jezika.

U odnosu na način rešavanja problema programski jezici mogu biti:

Proceduralni jezici, koji zahtevaju od programera da precizno zada algoritam kojim se problem rešava navodeći naredbe oblika: „uradi ovo, uradi ono“. Značajni proceduralni programski jezici su, na primer, C, Pascal, Java, ...

Deklarativni jezici, koji ne zahtevaju od programera da opisuje korake rešenja problema, već samo da opiše problem, dok se programski jezik sâm stara o pronalaženju rešenja korišćenjem nekog od ugrađenih algoritama. Ovo u mnogome olakšava proces programiranja, međutim, zbog nemogućnosti automatskog pronalaženja efikasnih algoritama koji bi rešili široku klasu problema, domen primene deklarativnih jezika je često ograničen. Najznačajniji deklarativni programski jezici su, na primer, Prolog i SQL. S obzirom da se programi u krajnjoj instanci moraju izvršavati na računarima fon Nojmanove arhitekture, oni se moraju prevesti na mašinski programski jezik koji je imperativan.

S obzirom na stil programiranja razlikujemo nekoliko *programskih paradigmi*. Paradigme se razlikuju po konceptima i apstrakcijama koje se koriste da bi se predstavili elementi programa (na primer, promenljive, funkcije, objekti, ograničenja) i koracima od kojih se sastoje izračunavanja (dodele, sračunavanja vrednosti izraza, tokovi podataka, itd.). Najznačajnije grupe jezika na osnovu programske paradigme su:

Imperativni jezici, koji razmatraju izračunavanje kao niz iskaza (naredbi) koje menjaju stanje programa određeno tekućom vrednošću promenljivih. Vrednosti promenljivih i stanje programa se menja naredbom dodele, a kontrola toka programa se vrši korišćenjem sekvencijalnog, uslovnog

i cikličkog izvršavanja programa. Imperativni jezici su obično izrazito proceduralni.

Najznačajniji imperativni programski jezici su FORTRAN i ALGOL i njihovi naslednici, Pascal, C, Basic, ...

Imperativni jezici, uz objektno orijentisane jezike, se najčešće koriste u industrijskom sistemskom i aplikativnom programiranju.

Funkcionalni jezici, koji razmatraju izračunavanje kao proces izračunavanja matematičkih funkcija i tretiraju stanja i promenljive potpuno drugačije nego imperativni jezici („promenljivama“ se ne može menjati vrednost). Koreni funkcionalnog programiranja leže u λ-računu razvijenom 1930-tih kako bi se izučavao proces definisanja i primene matematičkih funkcija i rekurzija. Mnogi funkcionalni programski jezici mogu se smatrati nadogradnjama λ-računa.

Ključna razlika između matematičkog pojma funkcije i pojma funkcija u imperativnim programskim jezicima je u tome što funkcije u imperativnim programima mogu imati *propratne (bočne) efekte* — uz izračunavanje rezultata one mogu menjati i nešto sem svojih parametara, tj. mogu uticati na tekuće stanje programa. Ovo dovodi do toga da poziv iste funkcije sa istim argumentima može da proizvede različite rezultate u zavisnosti od stanja u kojem je poziv izvršen, što pojam funkcije u imperativnim jezicima čini bitno drugačijim od matematičkih funkcija (koje za iste vrednosti ulaznih argumenata uvek imaju istu vrednost). U funkcionalnim programskim jezicima nisu mogući propratni efekti i u njima funkcija uvek daje isti rezultat za iste ulazne argumente. Zbog tog svojstva mnogo je jednostavnije razumeti, analizirati i predvideti ponašanje programa, kao i izbeći greške koje često nastaju u razvoju imperativnih programa. Takođe, to svojstvo omogućava mnoge automatske optimizacije izračunavanja što današnje funkcionalne jezike čini veoma efikasnim. Jedna od takvih optimizacija je mogućnost automatske paralelizacije i izvršavanja na višeprocesorskim sistemima.

Kako se ispravnost funkcionalnih programa dokazuje mnogo lakše nego ispravnost imperativnih programa, funkcionalno programiranje često se koristi za razvoj bezbednosno kritičnih sistema (na primer, softver u nuklearnim elektranama, svemirskim programima, avionima, ...).

Najznačajniji funkcionalni programski jezici su LISP, Scheme, ML, Haskell, Erlang, Elixir, ...

Logički jezici, koji, po najširoj definiciji ove paradigme, koriste matematičku logiku za programiranje računara. Na osnovu ove široke definicije koreni logičkog programiranja leže još u radovima Makartija u oblasti veštačke inteligencije 1950-tih godina. Ipak, najznačajniji logički jezik je Prolog razvijen 1970-tih godina. Logički jezici su obično deklarativni.

Objektno-orijentisani jezici, koji koriste *objekte* - specijalizovane strukture podataka koje uz polja podataka sadrže i metode kojima se manipuliše podacima. Podaci se mogu obrađivati isključivo primenom metoda što smanjuje zavisnosti između različitih komponenata programskog koda i čini ovu paradigmu pogodnu za razvoj velikih aplikacija uz mogućnost saradnje većeg broja programera. Najčešće korišćene tehnike programiranja u OOP uključuju sakrivanje informacija, enkapsulaciju, apstraktne tipove podataka, modularnost, nasleđivanje i polimorfizam. Najznačajniji objektno-orijentisani jezici su SmallTalk, Eiffel, C++ i Java.

Način zadavanja i određivanja tipova promenljivih bitno utiču na brzinu i pouzdanost izvršavanja sa jedne, kao i na jednostavnosti kodiranja, sa druge strane. Prema ovom pitanju, programski jezici mogu biti:

Statički tipizirani jezici, u kojima je potrebno deklarisati promenljivu (navesti njeno ime i tip) pre prve upotrebe. Drugim rečima, tip svake promenljive poznat je u fazi izvršavanja. Provera tipova (na primer, u izrazima i pozivima funkcija) vrši se u fazi kompilacije, čime se omogućava rano otklanjanje mnogih bagova, kao i efikasnije izvršavanje kasnije. Primeri ovakvih jezika su C, C++, Java i Scala.

Dinamički tipizirani jezici, u kojima nije potrebno deklarisati promenljivu pre prve upotrebe. Tip promenljive određuje se na osnovu njene prve upotrebe. Kodiranje je jednostavnije jer nije potrebno navoditi tipove promenljivih i jer se razne provere (na primer, provera tipova) odlažu za fazu izvršavanja. To znači da program može proći kompilaciju čak i ako nije moguće da se kasnije izvrši, što obično može da stvara probleme. Odsustvo deklaracija omogućava u dinamički tipiziranim jezicima pravljenje i grešaka poput sledeće:

```
kamata = 2.1
...
kamta = (kamata + 1.5) / 2
```

U navedenom kodu ime `kamata` navedeno je pogrešno (`kamta` umesto `kamata`), ali to komplilator neće registrovati kao grešku i kreiraće novu promenljvu sa imenom `kamta`). Primeri ovakvih jezika su Perl, Pajton, PHP i JavaScript.

Hibridno tipizirani jezici, U kojima je moguće korišćenje dinamičkog tipiziranja unutar funkcija, ali je neophodno statičko tipiziranje funkcija (tj. zadavanje tipiziranih deklaracija). Primer ovakvih jezika je Rascal.

Glava 8

Uvod u kompilaciju programskih jezika

Prevođenje (kompilacija) programskog jezika je transformiranje teksta programa na jednom računarskom jeziku u tekst programa na drugom jeziku. Obično je polazni jezik programski jezik visokog nivoa, a ciljni jezik je asembler ili mašinski jezik, ili jezik neke „virtualne mašine“. *Kompilatori* prevode čitav program na jeziku višeg nivoa u program na mašinskom ili nekom drugom ciljnom jeziku. Ukoliko je ciljni jezik mašinski jezik, onda, očigledno, kompilacija mora zavisiti od mašine na kojoj će se program izvršavati. *Interpretatori* program na jeziku višeg nivoa čitaju i odmah izvršavaju liniju po liniju (bez generisanja prevoda na asembleru tj. mašinskom jeziku). Interpretiranje se koristi i u alatima za debagovanje. Kompilatori troše značajno vreme za prevođenje pre samog izvršavanja, prevođenje je potrebno ponoviti prilikom svake izmene izvornog koda, ali je izvršavanje programa obično daleko brže nego kada se program interpretira.

Naglasimo da programski jezik sâm ne određuje da li će programi na njemu biti prevođeni na jedan, drugi ili neki treći način. Zaista, za mnoge jezike postoji raspoloživo više raznorodnih sistema za prevođenje i izvršavanje.

Već odavno se prevođenje ne svodi uvek na kompiliranje ili interpretiranje, već se u realnom softveru koristi neka njihova kombinacija. Na primer, neki kompilirani programi koriste u toku izvršavanja servise koji nisu zasnovani na mašinskom kodu i obratno. Naime, današnji softver je često sastavljen od komponenti koje su implementirane u različitim jezicima i saradnja između kompiliranog mašinskog koda i koda koji se na neki način interpretira nije retkost.

Neke platforme i programski jezici zasnovani su na ideji da se programi kompiliraju u specifičan „poluprevedeni kôd“ (često se naziva bajtkod, engl. bytecode) a zatim se taj kôd prilikom izvršavanja interpretira ili kompilira na neki specifičan način. Bajtkod se može shvatiti kao asemblerski tj. mašinski jezik neke „virtualne mašine“ koji je mnogo nižeg nivoa nego originalni program

na višem programskom jeziku, ali koji za razliku od realnog asemblera ne pokriva detalje konkretne arhitekture na kojoj će se program izvršavati. Veoma popularna je i takozvana JIT kompilacija (engl. just in time compilation), koja podrazumeva da se bajtkod izvršava tako što se pojedinačne naredbe programa pre svog izvršavanja prevode u mašinske instrukcije za ciljnu mašinu, koje se onda izvršavaju. Moguća je i takozvana AOT kompilacija (engl. ahead of time compilation) koja podrazumeva da se pre svog izvršavanja ceo bajtkod prevede na mašinski jezik ciljnog računara. I JIT i AOT kompilacijom se dobija mnogo brže izvršavanje programa nego kod klasičnog interpretiranja bajtkoda, a zadržavaju se prednosti koje prevođenje na bajtkod donosi. To je pre svega jednostavna prenosivost programa prevedenih na bajtkod na različite platforme — za svaku novu platformu potrebno je izgraditi samo interpreter, JIT ili AOT kompilator koji obrađuju bajtkod relativno blizak mašinskom jeziku, a to je mnogo jednostavnije nego razviti kompilator za polazni izvorni jezik visokog nivoa (prevođenje do nivoa bajtkoda je izrazito složen proces, koji uključuje kako različite analize, tako i različite napredne optimizacije izvornog programa).

Možda najznačajniji jezici danas koji koriste bajtkod su Java i C#. Java programi mogu da se kompiliraju na takozvani JAVA bajtkod, koji se onda može interpretirati ili JIT kompilirati na bilo kakvom računaru (koji ima raspoloživu takozvanu Java virtualnu mašinu — JVM). Postoje prevodioci i za druge programske jezike koji koriste ovaj pristup i generišu Java bajtkod (jedan od takvih je i funkcionalni programski jezik Scala). C# je deo Microsoft-ove .NET platforme i on se, kao i Visual Basic i F#, prevodi na bajtkod platforme .NET. Iako dominantno vezana za operativni sistem Windows, platforma .NET je podržana i na drugim operativnim sistemima (na primer, Mono je implementacija otvorenog koda platforme .NET koja može da se koristi i na Linux sistemima).

Kako bi se izvršila standardizacija i olakšala izgradnja jezičkih prevodioca potrebno je precizno definisati šta su ispravne sintaksičke konstrukcije programskog jezika. Opisi na govornom, prirodnom jeziku, iako mogući, obično nisu dovoljno precizni i potrebno je korišćenje preciznijih formalizama. Ovi formalizmi se nazivaju *metajezici* dok se jezik koji se opisuje korišćenjem metajezika naziva *objektni jezik*. Metajezici obično rezervišu neke simbole kao specijalne, obično za svoj zapis koriste samo ASCII karaktere i imaju svoju posebnu sintaksu pogodnu za jednostavnu obradu na računaru. U okviru ove glave biće ukratko opisano i nekoliko metajezika.

8.1 Struktura kompilatora

Kako bi bilo moguće prevođenje programa u odgovarajuće programe na mašinskom jeziku nekog konkretnog računara, neophodno je precizno definisati sintaksu i semantiku programskih jezika. Generalno, *leksika* se bavi opisivanjem osnovnih gradivnih elemenata jezika, a *sintaksa* načinima za kombinovanje tih

osnovnih elemenata u ispravne jezičke konstrukcije. Pitanjem značenja ispravnih jezičkih konstrukcija bavi se *semantika*.

Struktura kompilatora se, suštinski, nije mnogo promenila od vremena prvog Bakusovog kompilatora za FORTRAN, ali je svaki njihov fragment daleko napredniji od te pionirske verzije. Današnji kompilatori obično imaju tri ključne komponente:

- *prednji sloj* (eng. front-end): koji čita program zapisan na višem programskom jeziku, obrađuje ga i pohranjuje u obliku interne reprezentacije tj. međukoda. Sastoji se od sledećih komponenti (kojima u nekim slučajevima prethodi pretprocesor):
  - Leksički analizator;
  - Sintaksički analizator;
  - Semantički analizator;
  - Generator međukoda.
- *srednji sloj* (eng. middle-end): koja optimizije međukod i priprema ga za prevođenje na ciljni jezik. Čini ga jedna komponenta:
  - Optimizator međukoda.
- *zadnji sloj* (eng. back-end): koji prevodi internu reprezentaciju (međukod) u ciljni jezik (često, ali ne nužno mašinski jezik). Čini ga jedna komponenta:
  - Generator i optimizator ciljnog koda.

Razlaganje kompilatora na navedene komponente omogućava korišćenje jednog istog prednjeg i srednjeg sloja (i jednog jezika za međukod) sa različitim zadnjim slojevima (koji su prilagođeni različitim ciljnim arhitekturama).

U nastavku su ukratko opisani samo neki načini za zadavanje leksike, sintakse i semantike programskih jezika (ali ne i drugi detalji o svim fazama prevođenja)

8.2 Leksička analiza

Leksička analiza je proces izdvajanja „tokena“, osnovnih jezičkih elemenata, iz niza ulaznih karaktera (na primer, karaktera koji čine program). U analogiji sa prirodnim jezikom, leksička analiza bi odgovarala podeli rečenice na reči i određivanju vrste svake reči. Razmotrimo naredni kôd:

```
if (a == b);
    x = 1.5;
else
    x = a + 0.5;
```

```
endif;
```

Navedeni kôd je, zapravo, samo niz karaktera: `if_(a==b);\n\tx=1.5;\nelse \n\tx=a+0.5;\nendif;` i zadatak leksičkog analizatora („leksera“) je da razloži ovaj niz na tokene kao što su identifikator, broj u pokretnom zarezu, matematički operator, itd. Token je sintaksička klasa, kategorija, kao što su u prirodnom jeziku kategorije, na primer, imenice, glagoli ili prilozi. Leksema je konkretan primerak, konkretna instanca jednog tokena. Na primer, za token `IDENTIFIER`, primeri leksema bi mogli da budu `a`, `b`, za token `OPERATOR` primer lekseme mogao bi da bude `+`, itd.

Pored izdvajanja tokena („tokenizacije“), leksički analizator može imati i druge zadatke, kao što je, na primer, eliminacija komentara (ako nema pretprocesora). Tokom leksičke analize u specijalnu tabelu („tabelu simbola“) upisuju se prepoznati identifikatori i pridružuju im se određene relevantne informacije (na primer, vrsta i kolona u kodu gde je taj identifikator pronađen). Ova tabela dopunjuje se tokom narednih faza kompilacije (na primer, informacijama o tipovima).

Leksička analiza može da otkrije neke (jednostavne) vrste grešaka u kodu kao, na primer, u programskom jeziku C:

```
int a = 09;  /* error: invalid digit "9" in octal constant */
printf("Hi); /* error: missing terminating " character */
```

Leksički analizatori obično prosleđuju spisak izdvojenih leksema (i tokena kojima pripadaju) drugom programu koji nastavlja analizu teksta programa.

Regularni izrazi kao metajezik za opis leksike. Token, kao skup svih mogućih leksema, opisuje se formalno pogodnim obrascima koji mogu da uključuju cifre, slova, specijalne simbole i slično. Za te obrasce za opisivanje tokena (tj. za opis leksike programskog jezika) kao metajezik se obično koristi formalizam *regularnih izraza (engl. regular expressions)*.

Osnovne konstrukcije koje se koriste prilikom zapisa regularnih izraza su:

karakterske klase: navode se između `[` i `]` i označavaju jedan od navedenih karaktera. Na primer, klasa `[0-9]` označava cifru;

alternacija: navodi se sa `|` i označava alternativne mogućnosti. Na primer, `a|b` označava slovo `a` ili slovo `b`;

opciono pojavljivanje: navodi se sa `?`. Npr. `a?` označava da slovo `a` može, a ne mora da se javi;

ponavljanje: navodi se sa `*` i označava da se nešto javlja nula ili više puta. Npr. `a*` označava niz od nula ili više slova `a`;

pozitivno ponavljanje: navodi se sa `+` i označava da se nešto javlja jedan ili više puta. Npr. `[0-9]+` označava neprazan niz cifara.

Primer 8.1. Razmotrimo identifikatore u programskom jeziku C. Govornim jezikom, identifikatore je moguće opisati kao „neprazne niske koje se sastoje od slova, cifara i podvlaka, pri čemu ne mogu da počnu cifrom“. Ovo znači da je prvi karakter identifikatora ili slovo ili podvlaka, za čim sledi nula ili više slova, cifara ili podvlaka. Na osnovu ovoga, moguće je napisati regularni izraz kojim se opisuju identifikatori:

```
([a-zA-Z]|_)([a-zA-Z]|_|[0-9])*
```

Ovaj izraz je moguće zapisati još jednostavnije kao:

```
[a-zA-Z_][a-zA-Z_0-9]*
```

Algoritam za izdvajanje leksema iz ulaznog teksta zasniva se na takozvanim konačnim automatima. Postoje programi koji na osnovu opisa tokena u vidu regularnih izraza generišu leksere na izabranom programskom jeziku, na primer, na jeziku C. Primer takvog programa je `lex`.

8.3 Sintaksička analiza

Sintaksička analiza, poznata i kao *parsiranje*, je proces organizovanja leksema izdvojenih u fazi leksičke analize u ispravnu jezičku konstrukciju. U programskim jezicima ispravne jezičke konstrukcije mogu da uključuju dodele, petlje, uslovne naredbe itd. U analogiji sa prirodnim jezikom, sintaksička analiza odgovara proveri da li su reči u rečenici složene u skladu sa gramatičkim pravilima jezika, kao i određivanju gramatičke strukture rečenice (određivanje subjekta, predikata, itd).

Rezultat sintaksičke analize za ispravnu ulaznu jezičku konstrukciju je *sintaksičko stablo* (ili *stablo parsiranja*). U tom stablu, unutrašnji čvorovi su *neterminalni*, a listovi *terminalni* simboli. Programi koji vrše parsiranje zovu se *sintaksički analizatori* ili *parseri*.

Sintaksička analiza može da otkrije raznovrsne greške u kodu kao, na primer, u programskom jeziku C:

```
x = x +* y;  /* error: invalid type argument of unary ‘*’
                (have ‘int’) */
if x x++;    /* error: expected ‘(’ before ‘x’ */
```

Načini opisa sintakse programskih jezika. Za opisivanje sintakse jezika obično se koriste kontekstno-slobodne gramatike (jer izražajna snaga regularnih izraza nije dovoljno velika za to) i drugi odgovarajući metajezici: *BNF (Bakus-Naurova forma)*, *EBNF (proširena Bakus-Naurova forma)* i *sintaksički dijagrami*. BNF je pogodna notacija za zapis kontekstno-slobodnih gramatika, EBNF proširuje BNF operacijama regularnih izraza čime se dobija pogodniji

zapis, dok sintaksički dijagrami predstavljaju slikovni metajezik za predstavljanje sintakse. Dok je BNF veoma jednostavan metajezik, precizna definicija EBNF zahteva više truda i ona je data kroz ISO 14977 standard.

Sintaksička analiza na osnovu zadate sintakse jezika zasniva se na takozvanim potisnim automatima. Postoje programi koji na osnovu opisa sintakse jezika (na primer, u vidu EBNF-a) generišu parsere na izabranom programskom jeziku, na primer, na jeziku C. Primer takvog programa je program `Yacc` („yet another compiler compiler“).

Kontekstno-slobodne gramatike. Formalizam regularnih izraza je obično dovoljan da se opišu leksički elementi programskog jezika (na primer, skup svih identifikatora, skup brojevnih literala, i slično). Međutim nije moguće konstruisati regularne izraze kojim bi se opisale neke konstrukcije koje se javljaju u programskim jezicima. Tako, na primer, nije moguće regularnim izrazom opisati skup reči $\{a^n b^n, n > 0\} = \{ab,\ aabb,\ aaabbb,\ \ldots\}$. Takođe, nije moguće napisati regularni izraz kojim bi se opisali svi ispravni aritmetički izrazi, tj. skup $\{a,\ a+a,\ a*a,\ a+a*a,\ a*(a+a),\ \ldots\}$.

Sintaksa jezika se obično opisuje gramatikama. U slučaju prirodnih jezika, gramatički opisi se obično zadaju neformalno, koristeći govorni jezik kao metajezik u okviru kojega se opisuju ispravne konstrukcije, dok se u slučaju programskih jezika, koriste znatno precizniji i formalniji opisi. Za opis sintaksičkih konstrukcija programskih jezika koriste se uglavnom kontekstno-slobodne gramatike (engl. context free grammars).

Kontekstno-slobodne gramatike su izražajniji formalizam od regularnih izraza. Sve što je moguće opisati regularnim izrazima, moguće je opisati i kontekstno-slobodnim gramatikama, tako da je kontekstno-slobodne gramatike moguće koristiti i za opis leksičkih konstrukcija jezika (doduše regularni izrazi obično daju koncizniji opis).

Kontekstno-slobodne gramatike su određene skupom pravila. Sa leve strane pravila nalaze se takozvani pomoćni simboli (neterminali), dok se sa desne strane nalaze niske u kojima mogu da se javljaju bilo pomoćni simboli bilo takozvani završni simboli (terminali). Svakom pomoćnom simbolu pridružena je neka sintaksička kategorija. Jedan od pomoćnih simbola se smatra istaknutim, naziva se početnim simbolom (ili aksiomom). Niska je opisana gramatikom ako ju je moguće dobiti krenuvši od početnog simbola, zamenjujući u svakom koraku pomoćne simbole desnim stranama pravila.

Primer 8.2. Pokazano je da je jezik identifikatora programskog jezika C regularan i da ga je moguće opisati regularnim izrazom. Sa druge strane, isti ovaj jezik je moguće opisati i formalnom gramatikom. Razmotrimo naredna pravila (simbol ε označava praznu reč):

$$
\begin{array}{rcl}
I & \to & XZ \\
X & \to & S \mid P \\
Z & \to & YZ \mid \varepsilon \\
Y & \to & S \mid P \mid C \\
S & \to & \texttt{a} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \ldots \mid \texttt{Z} \\
P & \to & \texttt{\_} \\
C & \to & \texttt{0} \mid \ldots \mid \texttt{9}
\end{array}
$$

Na primer, identifikator `x_1` moguće je izvesti na sledeći način:

$$I \Rightarrow XZ \Rightarrow SZ \Rightarrow \texttt{x}Z \Rightarrow \texttt{x}YZ \Rightarrow \texttt{x}PZ \Rightarrow \texttt{x\_}Z \Rightarrow \texttt{x\_}YZ \Rightarrow \texttt{x\_}CZ \Rightarrow \texttt{x\_1}Z \Rightarrow \texttt{x\_1}.$$

Neterminal S odgovara slovima, neterminal P podvlaci, neterminal C ciframa, neterminal X slovu ili podvlaci, neterminal Y slovu, podvlaci ili cifri, a neterminal Z nizu simbola koji se mogu izvesti iz Y tj. nizu slova, podvlaka ili cifara.

Primer 8.3. Neka je gramatika određena skupom pravila:

$$E \to E + E \mid E * E \mid (E) \mid a\,.$$

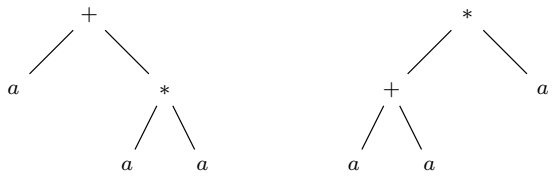
Ova gramatika opisuje ispravne aritmetičke izraze u kojima su dopuštene operacije sabiranja i množenja. Npr. izraz $a + a * a$ se može izvesti na sledeći način:

$$E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E \Rightarrow a + a * E \Rightarrow a + a * a.$$

Međutim, isti izraz je moguće izvesti na sledeći način:

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E \Rightarrow a + a * E \Rightarrow a + a * a.$$

Prvo izvođenje odgovara levo, a drugo desno prikazanom sintaksičkom stablu:



Primer 8.4. Neka je gramatika zadata sledećim pravilima:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid a \end{aligned}$$

I ova gramatika opisuje ispravne aritmetičke izraze. Na primer, niska $a+a*a$ se može izvesti na sledeći način:

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + T * F \Rightarrow \\ a + F * F \Rightarrow a + a * F \Rightarrow a + a * a.$$

Neterminal E odgovara izrazima, neterminal T sabircima (termima), a neterminal F činiocima (faktorima).

Primetimo da je ova gramatika u određenom smislu preciznija od gramatike date u prethodnom primeru, s obzirom da je njome jednoznačno određen prioritet i asocijativnost operatora, što sa prethodnom gramatikom nije bio slučaj.

Kontekstno-slobodne gramatike čine samo jednu specijalnu vrstu formalnih gramatika. U kontekstno-slobodnim gramatikama, sa leve strane pravila uvek se nalazi tačno jedan neterminalni simbol a sa desne strane pravila može se, u opštem slučaju, nalaziti proizvoljan niz terminalnih i neterminalnih simbola.

BNF. BNF je metajezik pogodan za zapis pravila kontekstno-slobodnih gramatika. Prvu verziju jezika kreirao je Džon Bakus, a ubrzo zatim poboljšao je Piter Naur i ta poboljšana verzija je po prvi put upotrebljena da se formalno definiše sintaksa programskog jezika ALGOL 60. BNF je u početku označavala skraćenicu od „Bakusova normalna forma“ (engl. Backus Normal Form), međutim na predlog Donalda Knuta, a kako bi se naglasio doprinos Naura, ime je izmenjeno u Bakus-Naurova forma (engl. Backus Naur Form) (čime je i jasno naglašeno da BNF nije *normalna forma* u smislu normalnih formi gramatika Čomskog).

U BNF notaciji, sintaksa objektnog jezika se opisuje pomoću konačnog skupa *metalingvističkih formula (MLF)* koje direktno odgovaraju pravilima kontekstno-slobodnih gramatika.

Svaka metalingvistička formula se sastoji iz leve i desne strane razdvojene specijalnim, takozvanim univerzalnim metasimbolom (simbolom metajezika koji se koristi u svim MLF) `::=` koji se čita „po definiciji je“, tj. MLF je oblika `A ::= a`, gde je `A` metalingvistička promenljiva, a `a` metalingvistički izraz. Metalingvističke promenljive su fraze prirodnog jezika u uglastim zagradama (<, >), i one predstavljaju pojmove, tj. sintaksičke kategorije objektnog jezika. Ove promenljive odgovaraju pomoćnim (neterminalnim) simbolima formalnih gramatika. U programskom jeziku, sintaksičke kategorije su, na primer, `<program>`, `<ceo broj>` i `<identifikator>`. U prirodnom jeziku, sintaksičke

kategorije su, na primer, `<rec>` i `<recenica>`. Metalingvističke promenljive ne pripadaju objektnom jeziku. U nekim knjigama se umesto uglastih zagrada `<` i `>` metalingvističke promenljive označavaju korišćenjem podebljanih slova.

Metalingvističke konstante su simboli objektnog jezika. To su, na primer, `0`, `1`, `2`, `+`, `-`, ali i rezervisane reči programskog jezika, na primer, `if`, `then`, `begin`, `for`, itd.

Dakle, uglaste zagrade razlikuju neterminalne simbole tj. imena sintaksičkih kategorija od terminalnih simbola objektnog jezika koji se navode tačno onako kakvi su u objektnom jeziku.

Metalingvistički izrazi se grade od metalingvističkih promenljivih i metalingvističkih konstanti primenom operacija konkatenacije i alternacije (`|`).

Metalingvistička formula `A ::= a` ima značenje: ML promenljiva `A` po definiciji je ML izraz `a`. Svaka metalingvistička promenljiva koja se pojavljuje u metalingvističkom izrazu `a` mora se definisati posebnom MLF.

Primer 8.5. Jezik celih brojeva u dekadnom brojnom sistemu može se opisati sledećim skupom MLF:

```
<ceo broj>           ::= <neoznacen ceo broj> |
                         <znak broja><neoznacen ceo broj>
<neoznacen ceo broj> ::= <cifra> |
                         <neoznacen ceo broj><cifra>
<cifra>              ::= 0|1|2|3|4|5|6|7|8|9
<znak broja>         ::= +|-
```

Primer 8.6. Gramatika aritmetičkih izraza može u BNF da se zapiše kao:

```
<izraz>  ::= <izraz> + <term> | <term>
<term>   ::= <term> * <faktor> | <faktor>
<faktor> ::= ( <izraz> ) | <ceo broj>
```

EBNF. EBNF (od „Extended Backus-Naur Form“ tj. „Proširena Bakus-Naurova forma“) je formalizam (opisan standardom ISO 14977) koji je iste izražajnosti kao BNF, ali uvodi skraćene zapise koji olakšavaju zapis gramatike i čine je čitljivijom. Nakon korišćenja BNF za definisanje sintakse ALGOL-a 60, pojavile su se mnoge njene modifikacije i proširenja.

EBNF proširuje BNF nekim elementima regularnih izraza:

- Vitičaste zagrade `{..}` okružuju elemente izraza koji se mogu ponavljati proizvoljan broj (nula ili više) puta. Alternativno, moguće je korišćenje i sufiksa `*`.

- Pravougaone zagrade `[..]` okružuju opcione elemente u izrazima, tj. elemente koji se mogu pojaviti nula ili jedan put. Alternativno, moguće je korišćenje i sufiksa `?`.

- Sufiks + označava elemente izraza koji se mogu pojaviti jednom ili više puta,

Svi ovi konstrukti se mogu izraziti i u BNF metajeziku, ali uz korišćenje znatno većeg broja MLF, što narušava čitljivost i jasnost (na primer, u BNF metajeziku opis izraza koji se ponavljaju odgovara rekurziji, a u EBNF metajeziku – iteriranju). Izražajnost i BNF metajezika i EBNF metajezika jednaka je izražajnosti kontekstno-slobodnih gramatika, tj. sva tri ova formalizma mogu da opišu isti skup jezika.

Ponekad se usvaja konvencija da se terminali od jednog karaktera okružuju navodnicima " kako bi se razlikovali od metasimbola EBNF.

Primer 8.7. Korišćenjem EBNF identifikatori programskog jezika C se mogu definisati sa:

```
<identifikator> ::= <slovo ili _> { <slovo ili _> | <cifra> }
<slovo ili _>   ::= "a"|...|"z"|"A"|...|"Z"|"_"
<cifra>         ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

Primer 8.8. Korišćenjem EBNF, jezik celih brojeva u dekadnom brojnom sistemu može se opisati sledećim skupom MLF:

```
<ceo broj> ::= ["+"|"-"]<cifra>{<cifra>}
<cifra>    ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

Primer 8.9. Gramatika aritmetičkih izraza može u EBNF da se zapiše kao:

```
<izraz>  ::= <term> {"+" <term>}
<term>   ::= <faktor> {"*" <faktor>}
<faktor> ::= "(" <izraz> ")" | <ceo broj>
```

Primer 8.10. Naredba grananja `if` sa opcionim pojavljivanjem `else` grane može se opisati sa:

```
<if_naredba>  ::= if "(" <bulovski_izraz> ")"
                     <niz_naredbi>
                  [ else
                     <niz_naredbi> ]
<niz_naredbi> ::= "{" <naredba> ";" { <naredba> ";" } "}"
```

S obzirom na veliki broj različitih proširenja BNF notacije, u jednom trenutku je postalo neophodno standardizovati notaciju. Verzija koja se danas standardno podrazumeva pod terminom EBNF je verzija koju je definisao i upotrebio Niklaus Virt u okviru definicije programskog jezika Pascal, 1977. Međunarodni komitet za standardizaciju definiše ovu notaciju u okviru standarda *ISO 14977*.

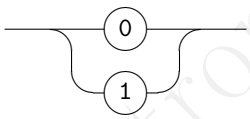
- Završni simboli objektnog jezika se navode pod navodnicima kako bi se svaki karakter, uključujući i one koji se koriste kao metasimboli u okviru EBNF, mogli koristiti kao simboli objektnog jezika.
- Pravougaone zagrade `[` i `]` ukazuju na opciono pojavljivanje.
- Vitičaste zagrade `{` i `}` ukazuju na ponavljanje.
- Svako pravilo se završava eksplicitnom oznakom kraja pravila.
- Obične male zagrade se koriste za grupisanje.

Sintaksički dijagrami. Sintaksički dijagrami ili sintaksički grafovi ili pružni dijagrami (engl. railroad diagrams) su grafička notacija za opis sintakse jezika koji su po izražajnosti ekvivalentni sa kontekstno-slobodnim gramatikama tj. sa BNF. Primeri sintaksičkih dijagrama:

- Sintaksički dijagram za `BinarnaCifra`:

```
<BinarnaCifra> ::= "0" | "1"
```

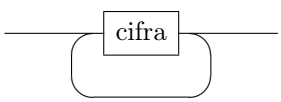
BinarnaCifra



- Sintaksički dijagram za `NeoznacenCeoBroj`:

```
<neoznacen_ceo_broj> ::= <cifra>+
```

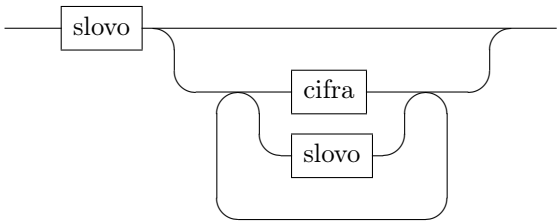
NeoznacenCeoBroj



- Sintaksički dijagram za `identifikator`:

```
<identifikator> ::= <slovo> { <slovo> | <cifra> }
```

identifikator



8.4 Semantička analiza

Semantička analiza je proces u kojem se proveravaju semantički uslovi i primenjuju pravila koja nije pogodno opisati sintaksičkim pravilima ni primenjivati u fazi sintaksičke analize. Semantička analiza vrši se nad sintaksičkim stablom izgrađenim tokom faze sintaksičke analize i neznatno ga modifikuje. Semantička analiza obično uključuje proveru tipova, implicitne konverzije, kao i provere koje se odnose na vidljivost promenljivih, to jest na njihov domen (jer kontekstno-slobodne gramatike nisu dovoljno izražajne da opišu pravila domena).

Semantička analiza može da otkrije raznovrsne greške u kodu kao, na primer, u programskom jeziku C:

```
int x, x;           /* error: redeclaration of ‘x’ with
                       no linkage */
char *s = x * "s"; /* error: invalid operands to binary *
                       (have ‘int’ and ‘char *’) */
```

8.5 Generisanje međukoda

Generisanje međukoda je proces u okviru kojeg se izlaz iz faze semantičke analize (u vidu označenog sintaksičkog stabla) prevodi u linearnu reprezentaciju, nezavisnu od konkretnih mašina. Na primer, složeni izrazi mogu se svoditi u međukodu samo na pojedinačne operacije sa po dva argumenta, koje lako mogu da se prevedu na asembler. To svojstvo je očuvano i tokom optimizacije međukoda. Time je omogućeno da se (mnogi) izrazi u međukodu mogu čuvati kao nizovi jednostavnih četvorki koje čine operator, dva argumenta i rezultat.

8.6 Optimizacija međukoda

Optimizacija međukoda je proces u okviru kojeg se na generisani međukod primenjuju raznovrsne optimizacije u cilju dobijanja efikasnijeg i kvalitetnijeg ciljnog koda, a bez promene njegovog vidljivog ponašanja (tj. uz čuvanje *semantike programa*). Te optimizacije mogu da uključuju eliminisanje koda koji se ne

koristi, propagaciju konstanti, promenu poretka naredbi, transformaciju petlji i slično. Optimizacije mogu da se odnose na procenjeno vreme izvšavanja, na procenjen prostor potreban za izvršavanje ili na procenjenu veličinu izvršivog programa. Proces optimizacije mora da uzima u obzir procenjeno ukupno vreme kompilacije jer se često ne isplati primenjivati najkompleksnije optimizacije.

Optimizacije mogu da budu lokalne i da se odnose na delove programa koji imaju jednostavnu, sekvencijalnu formu. Komplikovanije su optimizacije koje se odnose na razgranate delove programa i petlje, a najkomplikovanije optimizacije koje uključuju zavisnosti između različitih funkcija.

Optimizacija može da naredne komande na međujeziku

```
x := x + 0  // eliminisanje koda bez efekta
x := x * 1  // eliminisanje koda bez efekta
x := x * 0  // moze se uprostiti
z := x + u  // moguca je primena propagacije vrednosti x
y := 2 * x  // eliminisanje mrtvog koda
y := 2 * z  // moze se uprostiti
```

zameni narednim komandama:

```
x := 0
z := 0 + u;
y := z << 1
```

U oblasti prevođenja programskih jezika, ovo je faza koja je najčešći predmet inovacija i istraživanja (prethodne faze se već dugo sprovode na suštinski iste načine).

8.7 Generisanje i optimizacija ciljnog koda

Generisanje i optimizacija ciljnog koda je proces prevođenja međukoda na ciljni jezik, često jezik prilagođen nekoj konkretnoj računarskoj arhitekturi, kao što su konkretni mašinski jezici. Taj proces uključuje baratanje resursima niskog nivoa (na primer, da li će neka promenljiva da bude čuvana u registrima ili u drugoj memoriji). U okviru optimizacije koda, nizovi instrukcija na ciljnom jeziku (na primer, mašinskih instrukcija) mogu biti zamenjeni jednostavnijim fragmentima koda ili njihov poredak može biti promenjen radi kvalitetnijeg korišćenja registara.

8.8 Ilustracija sprovođenja faza kompilacije

Navedene faze ilustrovane su narednim pojednostavljenim primerom.

Primer 8.11. Izvorni kôd:

```
cur_time = start_time + cycles * 60
```

Leksička analiza:

```
ID(1) ASSIGN ID(2) ADD ID(3) MULT INT(60)
```

Sintaksička analiza:

```
    ASSIGN
ID(1)      ADD
      ID(2)   MULT
              ID(3) INT(60)
```

Semantička analiza:

```
    ASSIGN
ID(1)      ADD
      ID(2)   MULT
              ID(3) int2real
                    INT(60)
```

Generisanje međukoda:

```
temp1 = int2real(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Optimizacija međukoda:

Korak 1:

```
temp1 = 60.0
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Korak 2:

```
temp2 = id3 * 60.0
temp3 = id2 + temp2
id1 = temp3
```

Korak 3:

```
temp2 = id3 * 60.0
id1 = id2 + temp2
```

Korak 4:

```
temp1 = id3 * 60.0
id1 = id2 + temp1
```

Generisanje koda na ciljnom jeziku:

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

8.9 Načini opisa semantike programskih jezika

Semantika pridružuje značenje sintaksički ispravnim niskama jezika. Za prirodne jezike, ovo znači povezivanje rečenica sa nekim specifičnim objektima, mislima i osećanjima. Za programske jezike, semantika za dati program opisuje koje je izračunavanje određeno tim programom.

Dok većina savremenih jezika ima precizno i formalno definisanu sintaksu, formalna definicija semantike postoji samo za neke programske jezike. U ostalim slučajevima, semantika programskog jezika se opisuje neformalno, opisima zadatim korišćenjem prirodnog jezika. Čest je slučaj da neki aspekti semantike ostaju nedefinisani standardom jezika i prepušta se implementacijama kompilatora da samostalno odrede potpunu semantiku.

Formalno, semantika programskih jezika se zadaje na neki od naredna tri načina:

denotaciona sematika - programima se pridružuju matematički objekti (na primer funkcije koje preslikavaju ulaz u izlaz).

operaciona semantika - zadaju se korak-po-korak objašnjenja kako će se naredbe programa izvršavati na stvarnoj ili apstraktnoj mašini.

aksiomatska semantika - opisuje se efekat programa na tvrđenja (logičke formule) koja opisuju stanje programa. Najpoznatiji primer aksiomatske semantike je *Horova logika*.

Na primer, semantika UR mašina definiše značenje programa operaciono, dejstvom svake instrukcije na stanje registara apstraktne UR mašine kao što je navedeno u tabeli 8.1.

Pitanja i zadaci za vežbu

Pitanje 8.1. *U čemu je razlika između interpretatora i kompilatora?*

Pitanje 8.2. *Da li se, generalno, brže izvršava program koji se interpretira ili onaj koji je preveden kompilatorom? Zašto?*

oznaka	naziv	efekat
$Z(m)$	nula–instrukcija	$R_m \leftarrow 0$ (tj. $r_m := 0$)
$S(m)$	instrukcija sledbenik	$R_m \leftarrow r_m + 1$ (tj. $r_m := r_m + 1$)
$T(m,n)$	instrukcija prenosa	$R_n \leftarrow r_m$ (tj. $r_n := r_m$)
$J(m,n,p)$	instrukcija skoka	ako je $r_m = r_n$, idi na p-tu; inače idi na sledeću instrukciju

Tabela 8.1: Tabela URM instrukcija

Pitanje 8.3. *Prvim implementiranim kompilatorom se smatra onaj napravljen za jezik FORTRAN. Kada je on nastao?*

Pitanje 8.4. *Navesti primer logičkog jezika, primer funkcionalnog programskoj jezika i primer objektno-orijentisanog programskog jezika.*

Pitanje 8.5. *Kojoj grupi jezika pripada LISP? Kojoj grupi jezika pripada PROLOG? Kojoj grupi jezika pripada C++?*

Pitanje 8.6. *Za veliki broj savremenih programskih jezika kažu da su muliparadigmatični (engl. multi paradigm). Šta to znači.*

Pitanje 8.7. *Na kojem formalizmu izračunavanja je zasnovan jezik LISP?*

Pitanje 8.8. *Na kojoj logičkoj metodi je zasnovan jezik PROLOG?*

Pitanje 8.9. *U objektno-orijentisanoj paradigmi, šta čini jednu klasu?*

Pitanje 8.10. *Navesti osnovne faze u prevođenju programskih jezika.*

Pitanje 8.11. *Šta je to leksička analiza i koji se poslovi vrše tokom nje?*

Pitanje 8.12. *Navesti niz leksema i niz odgovarajućih tokena koji su rezultat rada leksičkog analizatora nad izrazom* `a=3;`

Pitanje 8.13. *Kako se zovu programi koji vrše leksičku analizu?*

Pitanje 8.14. *U kom vidu se opisuju pravila koja se primenjuju u fazi leksičke analize?*

Pitanje 8.15. *U opisu regularnih jezika, kako se označava ponavljanje jednom ili više puta?*

Pitanje 8.16. *Navesti nekoliko primera reči koje pripadaju regularnom jeziku* `[a-z][0-9]+[0-9]`.

Pitanje 8.17. *Zapisati jezik studentskih naloga (na primer,* `mi10123`, `mr10124`, `aa10125`*) na studentskom serveru Alas u vidu regularnog izraza.*

Pitanje 8.18. *Navesti regularni izraz koji opisuje:*
(a) *niske u kojima se na početku mogu (a ne moraju) pojavljivati cifre (od 0 do 9), zatim sledi neprazan niz karaktera* `a`*, a zatim sledi neprazan niz cifara.*
(b) *niske u kojima se na početku može (a ne mora) pojaviti neka cifra (od 0 do 9), zatim sledi neprazan niz karaktera* `a`*, a zatim sledi niz (moguće prazan) karaktera* `b`*.*
(c) *neprazne konačne niske u kojima se na početku može (a ne mora) pojaviti karakter* `a`*, a zatim sledi neprazan niz cifara.*
(d) *konačne niske u kojima se na početku može (a ne mora) pojaviti karakter* `a` *ili karakter* `b`*, a zatim sledi niz cifara (moguće i prazan).*
(e) *neprazne konačne niske u kojima se pojavljuju ili samo karakteri* `a` *ili samo karakteri* `b`*.*
(f) *neprazne konačne niske u kojima se pojavljuju samo karakteri* `a` *i* `b`*.*

Pitanje 8.19. *Koja analiza se u kompilaciji programa vrši nakon leksičke analize?*

Pitanje 8.20. *Šta je to sintaksička analiza?*

Pitanje 8.21. *Kako se zovu programi koji vrše sintaksičku analizu?*

Pitanje 8.22. *U kom obliku se opisuju pravila koja se primenjuju u fazi sintaksičke analize?*

Pitanje 8.23. *Da li su regularni jezici su obavezno i kontest-slobodni?*
Da li su kontekstno-slobodni jezici su obavezno i regularni?
Da li metajezik BNF ima istu izražajnost kao regularni jezici?
Da li metajezik BNF ima istu izražajnost kao kontekstno-slobodni jezici?
Da li metajezik BNF ima istu izražajnost kao metajezik EBNF?

Pitanje 8.24. *Od čega dolazi skraćenica EBNF?*

Pitanje 8.25. *Kako se u EBNF-u, zapisuje izraz koji se može ponavljati proizvoljan broj (nula ili više)?*

Pitanje 8.26. *Koji od sledećih metajezika je najizražajniji:*
(a) *kontekstno-slobodne gramatike;* (b) *BNF;* (c) *EBNF;* (d) *svi navedeni metajezici imaju istu izražajnost.*

Pitanje 8.27. *Zapisati jezik* $\{a^n b^n | n > 0\}$ *u vidu konteksno-slobodne gramatike i u vidu EBNF izraza.*

Deo IV

Osnove razvoja softvera

Glava 9

Životni ciklus razvoja softvera

Pod *razvojem softvera* često se ne misli samo na neposredno pisanje programa, već i na procese koji mu prethode i slede. U tom, širem smislu, razvoj softvera naziva se i *životni ciklus razvoja softvera*. Razvoj softvera razlikuje se od slučaja do slučaja, ali u nekoj formi obično ima sledeće faze i podfaze:

Planiranje: Ova faza obuhvata prikupljanje i analizu zahteva od naručioca softvera, razrešavanje nepotpunih, višesmislenih ili kontradiktornih zahteva i kreiranje precizne specifikacije problema i dizajna softverskog rešenja. Podfaze ove faze, opisane u poglavlju 9.1, su:

- Analiza i specifikovanje problema;
- Modelovanje rešenja;
- Dizajn softverskog rešenja.

Realizacija: Ova faza obuhvata implementiranje dizajniranog softverskog rešenja u nekom konkretnom programskom jeziku. Implementacija treba da sledi opšte preporuke, kao i preporuke specifične za realizatora ili za konkretan projekat. Analizom efikasnosti i ispravnosti proverava se pouzdanost i upotrebljivost softverskog proizvoda, a za naručioca se priprema i dokumentacija. Podfaze ove faze su:

- Implementiranje (kodiranje, pisanje programa) (o nekim aspektima ove podfaze govori glava 1);
- Evaluacija – analiza ispravnosti i analiza efikasnosti (o nekim aspektima ovih podfaza govore redom glava 2 i glava 3);
- Izrada dokumentacije (obično korisničke dokumentacije – koja opisuje korišćenje programa i tehničke dokumentacije — koja opisuje izvorni kôd);

Eksploatacija: Ova faza počinje nakon što je ispravnost softvera adekvatno proverena i nakon što je softver odobren za upotrebu. Puštanje u rad uključuje instaliranje, podešavanja u skladu sa specifičnim potrebama i zahtevima korisnika, ali i testiranje u realnom okuženju i sveukupnu evaluaciju sistema u stvarnim uslovima korišćenja. Organizuje se obuka za osnovne i napredne korisnike i obezbeđuje održavanje kroz koje se ispravljaju greške ili dodaju nove manje funkcionalnosti. U održavanje se obično uloži više od tri četvrtine ukupnog rada u čitavom životnom ciklusu softvera. Podfaze ove faze su:

- Obuka i tehnička podrška;
- Puštanje u rad;
- Održavanje.

Postoje međunarodni standardi, kao što su ISO/IEC 12207 i ISO/IEC 15504, koji opisuju životni ciklus softvera kroz precizno opisane postupke izbora, implementacije i nadgledanja razvoja softvera. Kvalitet razvijenog softvera često se ocenjuje prema nivou usklađenosti sa ovim standardima.

Kontrola kvaliteta softvera (eng. software quality assurance, SQA) pokriva kompletan proces razvoja softvera i sve njegove faze i podfaze. Proces kontrole kvaliteta, takođe opisan standardom ISO/IEC 15504, treba da osigura nezavisnu potvrdu da su svi proizvodi, aktivnosti i procesi u skladu sa predefinisanim planovima i standardima.

Faze razvoja softvera i moguće probleme na šaljiv način ilustruje čuvena karikatura prikazana na slici 9.1.

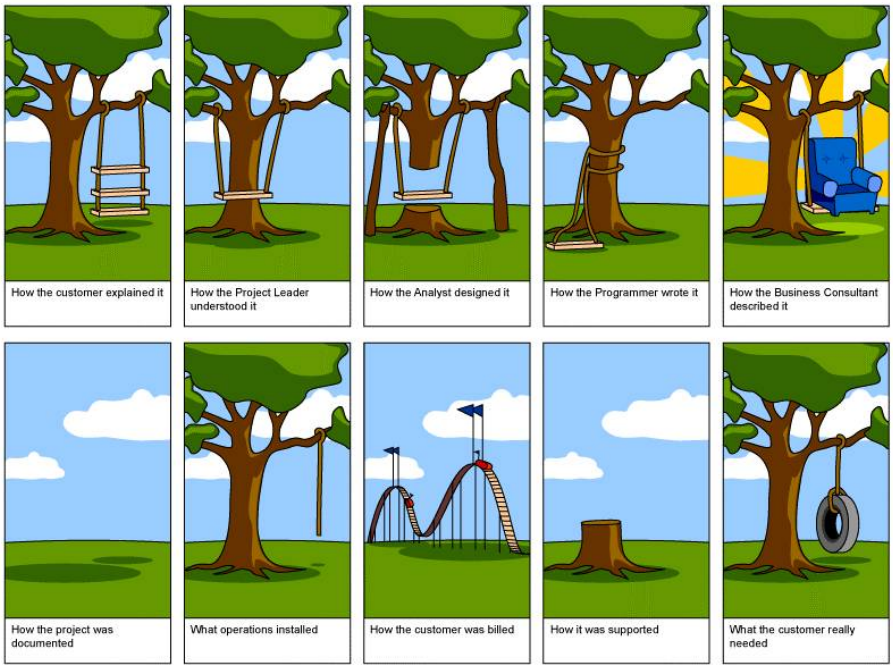
Za razvoj softvera relevantni su i procesi istraživanja tržišta, nabavke softvera, naručivanja softvera, tenderi, razmatranje ponuda i slični, ali u ovom tekstu neće biti reči o njima.

9.1 Planiranje

Poslovna analiza u fazi planiranja bavi se, pre svega, preciznom postavkom i specifikovanjem zahteva, dok se modelovanje i dizajn bave razradom projekta koji je definisan u fazi analize. U fazi planiranja često se koriste različite dijagramske tehnike i specijalizovani alati koji podržavaju kreiranje ovakvih dijagrama (takozvani CASE alati, engl. Computer Aided Software Engineering). Procesom planiranja strateški rukovodi arhitekta čitavog sistema (engl. enterprise architect, EA). Njegov zadatak je da napravi opšti, apstraktan plan svih procesa koji treba da budu softverski podržani.

9.1.1 Analiza i specifikovanje problema

Proces analize i specifikovanja problema obično sprovodi *poslovni analitičar* (engl. business analyst, BA), koji nije nužno informatičar, ali mora da poznaje relevantne poslovne ili druge procese. Kada se softver pravi po narudžbini, za



Slika 9.1: Faze razvoja softvera ilustrovane na šaljiv način

poznatog kupca, u procesu analize i specifikovanja problema vrši se intenzivna komunikacija poslovnog analitičara sa naručiocima, krajnjim korisnicima ili njihovim predstavnicima. Kada se softver pravi za nepoznatog kupca, često u kompanijama ulogu naručioca preuzimaju radnici zaposleni u odeljenju prodaje ili marketinga (koji imaju ideju kakav proizvod bi kasnije mogli da prodaju).

U komunikaciji poslovnog analitičara sa naručiocima, često se najpre vrši analiza postojećih rešenja (na primer, postojećeg poslovnog procesa u kompaniji koja uvodi informacioni sistem) i razmatraju se mogućnosti njihovog unapređenja uvođenjem novog softvera. Naručioci često nemaju informatičko obrazovanje, pa njihovi zahtevi koje softver treba da zadovolji mogu da budu neprecizni ili čak i kontradiktorni. Zadatak poslovnog analitičara je da, u saradnji sa naručiocima, zahteve precizira i uobliči. Rezultat analize je opšta specifikacija problema koja opisuje problem (na primer, poslovne procese) i željene funkcionalnosti programa, ali i potrebnu efikasnost i druga svojstva.

Pored precizne analize zahteva, zadatak poslovne analize je i da proceni: obim posla[1] koji treba da bude urađen (potrebno je precizno definisati šta

[1]Obim posla često se izražava u terminima broja potrebnih čovek-meseci (jedan čovek-mesec podrazumeva da jedan čovek na projektu radi mesec dana).

projekat treba da obuhvati, a šta ne); rizike koji postoje (i da definiše odgovarajuće reakcije u slučaju da nešto pođe drugačije nego što je planirano); potrebne resurse (ljudske i materijalne); očekivanu cenu realizacije projekta i njegovih delova; plan rada (po fazama) koji je neophodno poštovati i slično.

Kada je problem precizno specifikovan, prelazi se na sledeće faze u kojima se modeluje i dizajnira rešenje specifikovanog problema.

9.1.2 Modelovanje rešenja

Modelovanje rešenja obično sprovodi *arhitekta rešenja* (engl. solution architect, SA), koji mora da razume specifikaciju zahteva i da je u stanju da izradi matematičke modele problema i da izabere adekvatna softverska rešenja, na primer, programski jezik, bazu podataka, relevantne biblioteke, strukture podataka, algoritamska rešenja, itd.

9.1.3 Dizajn softverskog rešenja

U procesu dizajniranja, *arhitekta softvera* (engl. software architect) vrši preciziranje rešenja i opisuje *arhitekturu softvera* (engl. software architecture) – celokupnu strukturu softvera i načine na koje ta struktura obezbeđuje integritet sistema i željeni ishod projekta (ispravan softver, dobre performanse, poštovanje rokova i uklapanje u planirane troškove). Dizajn razrađuje i pojmove koji su u ranijim fazama bili opisani nezavisno od konkretnih tehnologija, dajući opšti plan kako sistem da bude izgrađen na konkretnoj hardverskoj i softverskoj platformi. Tokom dizajna često se koriste neki unapred ponuđeni obrasci (engl. design patterns) za koje je praksa pokazala da predstavljaju kvalitetna rešenja za određenu klasu problema.

U jednostavnijim slučajevima (na primer kada softver treba da radi autonomno, bez korisnika i korisničkog interfejsa), dizajn može biti dat i u neformalnom tekstualnom obliku ili u vidu jednostavnog dijagrama toka podataka tj. tokovnika (engl. data flow diagram)[2]. U kompleksnijim slučajevima, koriste se standardizovane grafičke notacije (kaže se i *grafički jezici*), poput UML (Unified Modeling Language), koji omogućavaju modelovanje podataka, modelovanje poslovnih procesa i modelovanje softverskih komponenti.

Neke od osnovnih tema koje se razmatraju u okviru dizajna softvera su:

- Apstrahovanje (engl. abstraction) – apstrahovanje je proces generalizacije kojim se odbacuju nebitne informacije tokom modelovanja nekog entiteta ili procesa i zadržavaju samo one informacije koje su bitne za sâm softver. Na primer, apstrahovanjem se uočava da boja očiju studenta nema

[2]Ovi dijagrami ilustruju kako podaci teku kroz sistem i kako se izlaz izvodi iz ulaza kroz niz funkcionalnih transformacija, ali ne opisuju kako ih treba implementirati. Notacija koja se koristi u tokovnicima nije standardizovana, ali različite notacije su često veoma slične i intuitivne.

nikakvog značaja u informacionom sistemu fakulteta i ta informacija se onda odbacuje prilikom predstavljanja studenta u sistemu.

- Profinjavanje (engl. refinement) – profinjavanje je proces razvoja programa odozgo-naniže. Nerazrađeni koraci se tokom profinjavanja sve više preciziraju dok se na samom kraju ne dođe do sasvim preciznog opisa u obliku funkcionalnog programskog koda. U svakom koraku jedan zadatak razlaže se na sitnije zadatke. Na primer, u nekoj situaciji, zadatak koji obavlja funkcija `obradi_podatke_iz_datoteke()` razlože se na zadatke koje obavljaju funkcije `otvori_datoteku()`, `procitaj_podatke()`, `obradi_podatke()`, `zatvori_datoteku()`, itd. Apstrahovanje i profinjavanje međusobno su suprotni procesi.

- Dekompozicija (engl. decomposition) – cilj dekompozicije je razlaganje na komponente koje je lakše razumeti, realizovati i održavati. Njen proizvod nije implementacija, već opis arhitekture softverskog rešenja. Postoje različiti pristupi dekompoziji, obično u skladu sa programskom paradigmom koja će se koristiti (na primer, objektno-orijentisana, funkcionalna, itd). Većina pristupa teži razlaganju na komponente tako da se što više smanje njihove zavisnosti (tako što unutrašnje informacije jednog modula nisu dostupne iz drugih) i da se poveća kohezija (jaka unutrašnja povezanost) pojedinačnih komponenti. Na primer, u funkcijski-orijentisanom dizajnu, svaka funkcija odgovorna je samo za jedan zadatak i sprovodi ga sa minimalnim uticajem na druge funkcije. Rezultat dekompozicije često se prikazuje grafički, u vidu strukturnog modela sistema koji opisuje veze između komponenti i njihovu hijerarhiju (na svakom nivou hijerarhije, svakom čvoru koji nije list, odgovara nekoliko, obično između dva i sedam, podređenih čvorova).

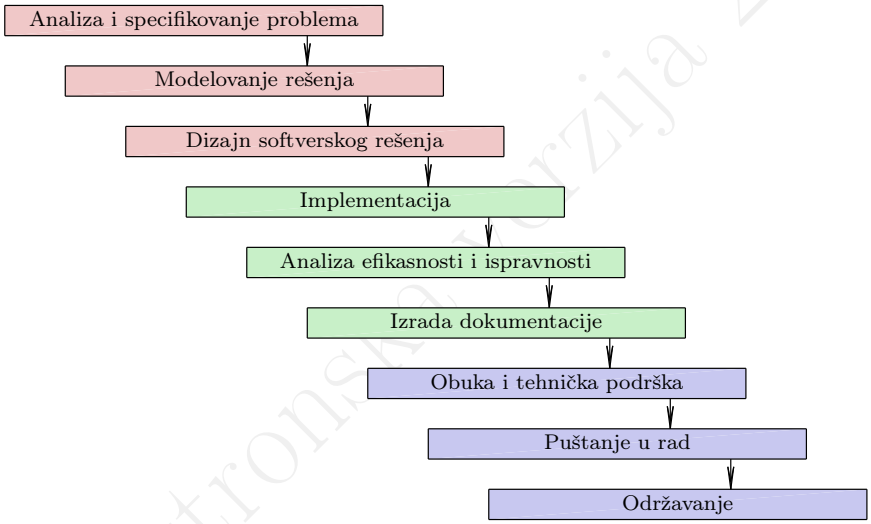
- Modularnost (engl. modularity) – softver se deli na komponente koje se nazivaju moduli. Svaki modul ima precizno definisanu funkcionalnost i poželjno je da moduli što manje zavise jedni od drugih kako bi mogli da se koriste i u drugim programima.

9.2 Metodologije razvoja softvera

I u teoriji i u praksi postoje mnoge metodologije razvoja softvera. U praksi su one često pomešane i često je teško striktno razvrstati stvarne projekte u postojeće metodologije. U nastavku je opisano nekoliko često korišćenih metodologija i ključnih ideja na kojima su zasnovane.

Metodologija vodopada. U strogoj varijanti ove tradicionalne metodologije, opisane prvi put još pedesetih godina prošlog veka, na sledeću fazu u razvoju softvera prelazi se tek kada je jedna potpuno završena (slika 9.2). Metodologija se smatra primenljivom ako su ispunjeni sledeći uslovi:

- svi zahtevi poznati su unapred i njihova priroda ne menja se bitno u toku razvoja;
- zahtevi su u skladu sa očekivanjima svih relevantnih strana (investitori, korisnici, realizatori, itd.);
- zahtevi nemaju nerazrešene, potencijalno rizične faktore (na primer, rizike koji se odnose na cenu, tempo rada, efikasnost, bezbednost, itd);
- pogodna arhitektura rešenja može biti opisana i podrobno shvaćena;
- na raspolaganju je dovoljno vremena za rad u etapama.

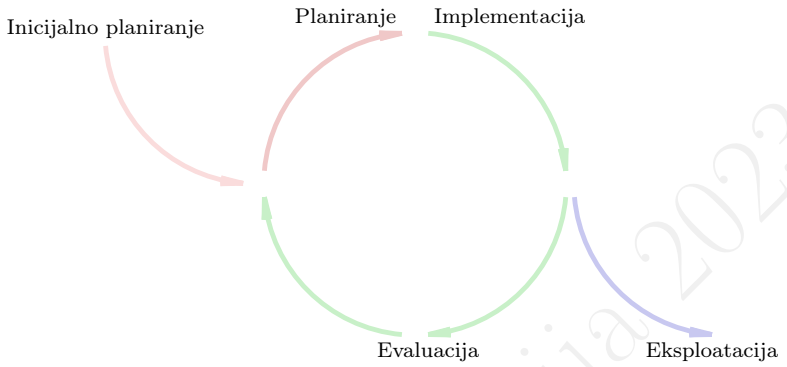


Slika 9.2: Ilustracija za metodologiju vodopada

Ova metodologija koristi se obično u veoma velikim timovima. Detaljna dokumentacija za sve faze je neophodna. Metodologija ne predviđa modifikovanje prethodnih faza jednom kada su završene i ova osobina, krutost metodologije, predmet je najčešćih kritika. Naime, izrada aplikacija često traje toliko dugo da se zahtevi promene u međuvremenu i završni proizvod više nije sasvim adekvatan a ponekad ni uopšte upotrebljiv.

Metodologija iterativnog i inkrementalnog razvoja. U ovoj metodologiji, opisanoj prvi put šezdesetih i sedamdesetih godina prošlog veka (ilustrovanoj slikom 9.3), razvoj se sprovodi u iteracijama i projekat se gradi inkrementalno. Iteracije obično donose više detalja i funkcionalnosti, a inkrementalnost podrazumeva dodavanje jednog po jednog modula, pri

čemu i oni mogu biti modifikovani ubuduće. U jednom trenutku, više različitih faza životnog ciklusa softvera može biti u toku. U ovoj metodologiji vraćanje unazad je moguće.



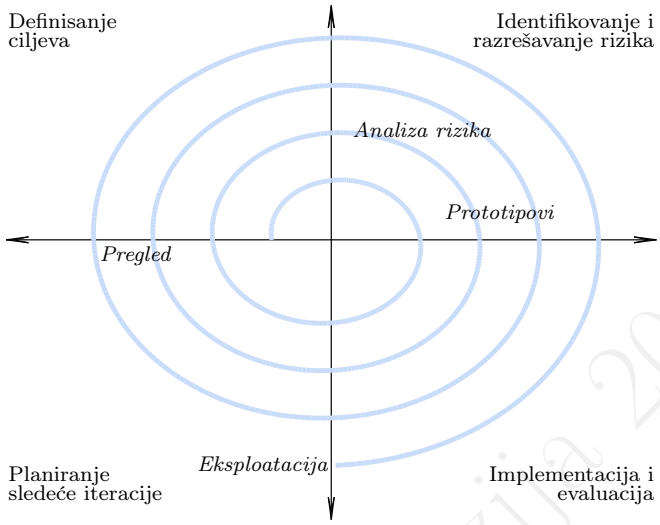
Slika 9.3: Ilustracija za iterativnu metodologiju

Metodologija rapidnog razvoja. U ovoj metodologiji (engl. rapid application development), opisanoj sedamdesetih i osamdesetih godina prošlog veka, faza planiranja svedena je na minimum zarad brzog dobijanja prototipova u iteracijama. Faza planiranja preklapa se sa fazom implementacije što olakšava izmene zahteva u hodu. Proces razvoja kreće sa razvojem preliminarnog modela podataka i algoritama, razvijaju se prototipovi na osnovu kojih se definišu, preciziraju ili potvrđuju zahtevi naručioca ili korisnika. Ovaj postupak ponavlja se iterativno, sve do završnog proizvoda. Aplikaciju prati vrlo ograničena dokumentacija.

Ova metodologija ponekad može dovesti do niza prototipova koji nikada ne dostižu do zadovoljavajuće finalne aplikacije. Čest izvor takvih problema su grafički korisnički interfejsi (engl. graphical user interface; GUI). Naime, korisnici napredak u razvoju aplikacije doživljavaju prvenstveno kroz napredak grafičkog korisničkog interfejsa. To podstiče programere, pa i vođe projekata, da se u prototipovima usredsređuju na detalje grafičkog interfejsa umesto na druge segmente aplikacije (kao što su, na primer, poslovni procesi i obrada podataka). Čak i mnogi razvojni alati privilegovano mesto u razvoju softvera daju razvoju grafičkih interfejsa. Ovakav razvoj aplikacije često dovodi do niza prototipova sa razrađenim korisničkim interfejsom, ali bez adekvatnih obrada koje stoje iza njega.

Ova metodologija pogodna je za razvoj softvera za sopstvene potrebe ili za potrebe ograničenog broja korisnika.

Spiralna metodologija. Ova metodologija (opisana prvi put krajem osamdesetih godina prošlog veka) kombinuje analizu rizika sa drugim meto-



Slika 9.4: Ilustracija za spiralnu metodologiju

dologijama kao što su metodologija vodopada i metodologije iterativnog razvoja. Spirala koja ilustruje ovu metodologiju (prikazana na slici 9.4) prolazi više puta kroz faze kao što su planiranje, implementacija i evaluacija tekućeg verzije, kao i analiza rizika. Različite faze ne sprovode se istovremeno, već jedna za drugom. Prvi prototip pravi se na osnovu preliminarnog, pojednostavljenog dizajna i predstavlja samo aproksimaciju finalnog proizvoda. Na kraju svake iteracije, prototip se evaluira, analiziraju se njegove dobre i loše strane, profinjuje specifikacija za sledeću iteraciju i analiziraju se rizici (rizici koji se odnose na bagove, na cenu, tempo rada, efikasnost, bezbednost, itd). Na primer, planiranje dodatnog testiranja smanjuje rizik od neispravnog proizvoda, ali može da uveća cenu ili da nosi rizik zakasnelog izlaska na tržište. Ako neki rizik ne može biti eliminisan, naručilac mora da odluči da li se sa projektom nastavlja ili ne. Ukoliko se sa projektom nastavlja, ulazi se u sledeću iteraciju.

Agilna metodologija razvoja. Agilne metodologije, u upotrebi od devedesetih godina prošlog veka a trenutno verovatno najpopularnije, teže minimizovanju rizika (kao što su bagovi, prekoračenje budžeta, izmena zahteva) razvijanjem softvera od strane visoko motivisanog tima, u iteracijama sa minimalnim dodavanjem funkcionalnosti i u kratkim vremenskim intervalima (obično od jedne do četiri nedelje). Agilne metodologije zahtevaju permanentnu komunikaciju, poželjno uživo (zbog čega, međutim, ne ostaje mnogo pisanog traga o progresu niti pisane dokumentacije). Zato su kompletni agilni timovi često locirani na jednom mestu: menadžeri, ana-

litičari, programeri, testeri, dizajneri, pisci dokumentacije, pa i predstavnici naručioca. *Manifest* agilne metodologije ima dvanaest jednostavnih principa, kao što su: glavna mera napretka je upotrebljivost raspoloživog softvera, održivi razvoj, neprekidna usredsređenost na dobar dizajn, itd.

Jedan od ciljeva ove metodologije je u ranom otkrivanju i ispravljanju propusta i neusklađenih očekivanja. Svaka iteracija odnosi se na minijaturni softverski proizvod sa svim uobičajenim fazama razvoja (koje se izvršavaju istovremeno). Svaku iteraciju potrebno je završiti na vreme i dobiti saglasnost naručioca. Za razliku od rapidne metodologije, u okviru koje se, u iteracijama, razvijaju nekompletni prototipovi, u agilnoj metodologiji, nakon nekih iteracija softver može biti isporučen naručiocu (ili na tržište) iako nema upotpunjenu funkcionalnost.

Agilna metodologija u mnogim je aspektima razvoja softvera uopštena, te postoji više vidova ove metodologije koji preciziraju neke njene aspekte, uključujući skram i ekstremno programiranje.

Skram (engl. scrum) je vid agilne metodologije u kojem se neposredna, praktična iskustva koriste u upravljanju izazovima i rizicima. Softverski proizvod sve vreme se održava u (integrisanom i testiranom) stanju koje se potencijalno može isporučiti. Vreme je podeljeno u kratke intervale, „sprintove“, obično duge samo jedan mesec ili kraće i na kraju svakog sprinta svi akteri i članovi time sastaju se da razmotre stanje projekta i planiraju dalje korake. Skram ima jednostavan skup pravila, zaduženja i sastanaka koji se, zarad jednostavnosti i predvidivosti, nikad ne menjaju. Postoje sastanci na početku i kraju svakog sprinta, ali i kratki, petnaestominutni dnevni sastanci („dnevni skram“). U skramu postoje tri uloge: vlasnik proizvoda (osoba sa vizijom i autoritetom koja usmerava članove tima), skram master (osoba koja olakšava komunikaciju između vlasnika proizvoda i tima) i razvojni tim (koji je sâm odgovoran za organizovanje i završavanje posla; tim obično ima tri do devet članova, idealno izdvojeno u jednoj prostoriji; tipičan tim uključuje programere, analitičare, testere, dizajnere za korisnički interfejs i slično).

Ekstremno programiranje je vid agilne metodologije u kojem su posebno važne jednostavnost, motivacija i kvalitetni odnosi unutar tima. Programeri rade u parovima (dok jedan programer piše kôd, drugi pokušava da pronađe i ukaže na eventualne greške i nedostatke) ili u većim grupama, na kodu jednostavnog dizajna koji se temeljno testira i unapređuje tako da odgovara tekućim zahtevima. U ekstremnom programiranju, sistem je integrisan i radi sve vreme (iako svesno nema potpunu funkcionalnost). Svi članovi tima upoznati su sa čitavim projektom i pišu kôd na konzistentan način, te svako može da razume kompletan kôd i da radi na svakom delu koda. U ekstremnom programiranju, faze se sprovode u veoma malim koracima i prva iteracija može da dovede, do svesno nepotpune ali funkcionalne celine, već za jedan dan ili nedelju. Zahtevi se obično ne mogu u potpunosti utvrditi na samom početku, menjaju se tokom vre-

mena, te naručilac treba da konstantno bude uključen u razvojni tim. Naručiocu se ne prikazuju samo planovi i dokumenti, već konstantno i (nekompletni, nesavršeni, ali funkcionalni) softver. Dokumentacija mora da postoji, ali se izbegava preobimna dokumentacija.

Pitanja i zadaci za vežbu

Pitanje 9.1. *Šta su sličnosti a koje razlike između projekata u građevinarstvu i informacionim tehnologijama?*

Pitanje 9.2. *Navesti faze razvoja softvera i ko ih obično sprovodi.*

Pitanje 9.3. *Koje dve vrste dokumentacije treba da postoje?*

Pitanje 9.4. *Nabrojati najznačajnije metodologije razvoja softvera. Istraži na internetu koje su metodoloje razvoja softvera danas najpopularnije.*

Deo V

Računari i društvo

Glava 10

Socijalni i etički aspekti računarstva

Socijalni i etički aspekti računarstva imaju ogroman značaj, ne samo za informatičare, već za čitavo društvo. Za razliku od programiranja i temelja na kojim se računarstvo gradi, njegovi socijalni i etički aspekti ne mogu se opisivati i analizirati na rigorozan, matematički način, te su za njihovo razumevanje i poznavanje potrebna drugačija sredstva.

10.1 Uticaj računarstva na društvo

Tokom prethodnih decenija računari su izvršili dramatičan uticaj na ljudsko društvo. U vreme prvih računara, njihove primene bile su ograničene na svega nekoliko tradicionalnih oblasti: vojne, naučne, bankarske primene i slično. Sa pojavom personalnih računara i, kasnije, sa pojavom interneta, računari ulaze u mnoge domove i danas su skoro sveprisutni (godine 2019, u Srbiji je desktop ili laptop računar imalo više od 73% domaćinstava, a raspoloživ internet više od 80% domaćinstava; u Severnoj Americi i u Evropskoj uniji desktop ili laptop računar imalo je oko 85% domaćinstava, a više od 90% je imalo raspoloživ internet; u svetu je desktop ili laptop računar imalo oko 50% domaćinstava, a više od 60% je imalo raspoloživ internet; ukupno, više od 60% osoba na svetu koristi internet svakodnevno ili često). Danas se računari koriste u skoro svim oblastima ljudskog života: u obrazovanju, u telekomunikacijama, u industriji zabave, za prognozu vremena, itd. Najnovije primene računara omogućavaju, na primer, automatsko dijagnostikovanje bolesti, analizu ljudskog genoma, automatsko navođenje automobila, *internet stvari* (sistem automatskog, efikasnijeg i jeftinijeg upravljanja elektronskim uređajima putem senzora, adekvatnog softvera i računarske mreže, na primer, za kontrolu temperature vazduha u nekom magacinu ili kući), a računari pobeđuju ljude u kvizovima opšteg znanja.

U nastavku su nabrojani neki od faktora zbog kojih računari sve više utiču na društvo u celini:

Sveprisutnost: Računari su sveprisutni, čak i onda kada se njihov rad ne primećuje.

Laka dostupnost informacija: Količina podataka koji su dostupni na internetu raste ogromnom brzinom, baš kao i broj ljudi kojima su ti podaci dostupni. Sa bilo kog mesta, u skoro bilo kojoj situaciji, dostupne su ogromne količine teksta, zvučnih i video zapisa. Internet pretraživači potpuno su potisnuli enciklopedije, ali i mnoge druge knjige.

Efekat umnožavanja: Porast raspoloživih informacija, omogućava mnoge nove, dodatne vidove obrada i dovodi do dalje, još veće proizvodnje podataka. Zbog opšte umreženosti, i korisne i nekorisne informacije, pa i softverski problemi stižu ogromnom brzinom do miliona ljudi.

Temporalnost: Zahvaljujući računarima i internetu, mnoge ljudske aktivnosti odvijaju se lakše i brže. To postavlja i sve viša očekivanja (na primer, za brzinu objavljivanja rezultata popisa ili rezultata izbora). Sve više elektronskih usluga raspoloživo je neprekidno, dvadeset četiri sata dnevno.

Prostor: Zahvaljujući računarima i internetu, fizička udaljenost više nije ograničenje za mnoge vrste poslova: može se istovremeno baratati ogromnim količinama podataka sa različitih mesta u svetu i mnogi poslovi mogu se raditi na daljinu.

Neuništivost: Zahvaljujući sveprisutnoj umreženosti, podaci se sve češće čuvaju na udaljenim računarima ili na mnogo njih. To obezbeđuje visok nivo pouzdanosti sistema, ali i do toga da se gomile beskorisnih podataka skladište i čuvaju zauvek.

10.2 Pouzdanost računarskih sistema i rizici po društvo

Dok rade ispravno, računarski sistemi često su neprimetni. Ali, ako dođe do greške u radu sistema, posledice mogu da budu katastrofalne. Računarski sistemi mogu da rade neispravno zbog neispravne specifikacije sistema, greške u dizajnu hardvera, hardverskog otkazivanja, greške u dizajnu softvera, baga u softveru, zbog neispravnog održavanja itd. U kompleksnim sistemima, greška može biti i neka kombinacija navedenih mogućih uzroka. Često su se dešavale greške u radu računarskih sistema koje su ugrozile ili mogle da ugroze živote velikog broja ljudi. Takva je, na primer, bila greška 1980. godine zbog koje je računarski sistem u SAD ukazivao na započeti nuklearni napad Sovjetskog Saveza što je moglo je da dovede do stvarnog nuklearnog rata i uništenja ljudske civilizacije.

Mnoge od primena računara neposredno su unapredile kvalitet ljudskog života. Međutim, postoje i oblasti u kojima je uticaj računara štetan ili, makar, upitan. Na primer, igranje računarskih igrica može da dovede do bolesti zavisnosti koje se teško leče. Mnogi smatraju da razvoj elektronskih društvenih mreža loše utiče na tradicionalne forme komunikacije. Na primer, upotreba

elektronske pošte elimiše mnoge atribute komunikacije uživo, ali to može biti ocenjeno i negativno i pozitivno — naime, elektronska komunikacija može da maskira atribute kao što je rasa, nacionalnost, pol, starost i slično (a koji u nekim situacijama mogu da dovedu do diskriminisanja neke osobe). Razvoj veštačke inteligencije budi kod ljudi nove strahove, ne samo od smanjenja radnih mesta za ljude, već i od stvaranja mašina koje čovek neće moći da kontroliše ili od inteligentnih mašina za ubijanje koje će se koristiti u ratovima kao najmodernije oružje. Ti strahovi nisu karakteristični samo za one koji ne poznaju računarstvo, već i za mnoge stručnjake u oblasti veštačke inteligencije (grupa naučnika objavila je 2015. pismo u kojem se poziva na pojačan oprez u razvoju sistema veštačke inteligencije).

10.3 Građanska prava, slobode i privatnost

Računari su omogućili, više nego ikad, uvid države, pa i raznih drugih organizacija u život pojedinaca. To često pojednostavljuje svakodnevicu, na primer, kada treba odrediti put od kuće do neke lokacije, ali istovremeno ostavlja prostor za mnoge zloupotrebe (jer, na primer, na internetu mogu da se nađu podaci o tačnom kretanju mnogih korisnika). Situacija je još komplikovanija zbog bezbednosnih (na primer, terorističkih) izazova sa kojima su suočene mnoge zemlje. Naime, tim izazovima opravdava se elektronski nadzor potencijalnih terorista, ali u okviru toga se pod nadzorom nađu i mnogi drugi. Nedavno je otkriveno da su, u okviru svoje borbe protiv terorizma, Sjedinjene Američke Države elektronski nadzirale ogroman broj svojih građana ali i građana drugih država, uključujući i političke lidere najbližih saveznika. Dok jedni smatraju da je to činjeno sa pravom i iz najboljih motiva, drugi smatraju da je to činjeno nelegalno i neetički. Osobe koje su objavile dokumenta koja govore o tom nadzoru, za jedne su heroji, a za druge – izdajnici. U mnogim državama, tokom prethodnih godina usvojeni su zakoni koji štite privatnost pojedinaca i dozvoljavaju elektronski nadzor samo u specijalnim situacijama.

Nisu samo države zainteresovane za prikupljanje privatnih podataka. Mnoge podatke, uz prećutnu saglasnost korisnika, prikupljaju pretraživači veba, društvene mreže i slično. Na osnovu postavljenih upita i poslatih poruka, kreira se precizan profil korisnika ka kojem se onda usmeravaju personalizovane reklame. Privatnost pojedinaca može da bude ugrožena i nehatom, greškom ili usled napada hakera. Više puta su provaljivani i zatim prodavani ili objavljivani spiskovi koji uključuju milione korisnika neke usluge, sa svim njihovim ličnim detaljima. Zbog takvih situacija, potrebno je preduzimati rigorozne mere koje minimizuju mogućnost gubljenja ili curenja podataka iz sistema. U kontekstu privatnosti, specifično je pitanje privatnosti zaposlenog i njegovih komunikacija u okviru kompanije u kojoj radi.

Jedno od osnovnih građanskih prava je pravo na slobodu govora – pravo da se, ukoliko one ne ugrožavaju druge, iznesu sopstvene ideje bez straha od kažnjavanja ili cenzure. Internet kao medij u principu omogućava potpunu slobodu izražavanja širokom krugu ljudi, ali istovremeno omogućava i promo-

visanje radikalnih ideja, mržnje, pa i terorističke borbe. Pored toga, internet je kroz društene mreže i forume postao i poligon za širenje svih vrsta poluinformacija, dezinformacija, „lažnih vesti“ (eng. „fake news“) i manipulacija, u tolikoj meri da mnogi smatraju da one mogu da utiču i na značajne političke odluke ili na ishod izbora. Pojedine države (u manjoj ili većoj meri) kontrolišu internet saobraćaj svojih građana ili pokušavaju da kontrolišu sadržaj na društvenim mrežama, a sa obrazloženjem da to čine u cilju očuvanja sopstvenih sloboda i nezavisnosti.

10.4 Etički aspekti računarstva

Zakoni su pravila koja omogućavaju ili zabranjuju neke vidove ponašanja. Oni su izvedeni iz etike, koja opisuje socijalno prihvatljivo ponašanje. Ključna razlika između zakona i etičkih pravila je u tome što vladajuća tela sprovode ova prva, ali ne i ova druga. Mnoga etička pravila su univerzalno prihvaćena, ali postoje i pravila zasnovana na stavovima i običajima neke uže zajednice. Na primer, zajednice kao što su lekarska ili advokatska, imaju detaljna etička pravila i članu zajednice koji ih krši može biti zabranjen dalji rad. Etička pitanja u informatici imaju dosta toga zajedničkog sa etičkim pitanjima u drugim oblastima, ali imaju i svojih specifičnosti. Etička pitanja mogu se ticati naručioca posla ili samog naručenog proizvoda (na primer, softver za ratne potrebe, softver koji se može koristiti za neovlašćeno prikupljanje podataka, itd). Etička pitanja postoje i u situacijama kada, na primer, zaposleni neovlašćeno koristi računarske resurse za lične potrebe, kada zaposleni primeti da se u njegovoj kompaniji koristi i neki nelegalno nabavljen softver, kada zaposleni zaključi da softver za medicinski uređaj svesno dozvoljava rizik po zdravlje korisnika, da je softver za merenje količine izduvnih gasova automobila napravljen da namerno radi neispravano (kao što je utvrđeno da je 2015. godine rađeno u kompaniji Volkswagen) itd. Etička pitanja mogu da postoje i u slučajevima prelaska od jednog poslodavca kod drugog koji mu predstavlja direktnu konkurenciju (što se često onemogućava ugovorom o zaposlenju).

Generalno, razmatranje i razrešavanje etičkih dilema može da bude veoma kompleksno i obično se zasniva na detektovanju svih aktera u konkretnoj situaciji, na detektovanju svih pojedinačnih etičkih pitanja, razmatranju na osnovu primera, analogije i kontraprimera. Upravo nedostatak situacija sa dovoljnim stepenom analogije često može da uzrokuje dileme u nekim situacijama koje se tiču računara. Na primer, sve aktuelnija su pitanja u vezi sa automobilima sa automatskom navigacijom. Kako taj softverski sistem treba da bude obučen – ukoliko se vozilo nađe u situaciji kada mora da riziku izloži ili vozača ili pešaka, šta će odlučiti?

Američki etički institut (engl. Computer Ethics Institute) objavio je 1992. *deset zapovesti računarske etike*:

1. Ne koristi računar da naudiš drugim ljudima.
2. Ne mešaj se nepozvan u tuđi rad na računaru.

3. Ne njuškaj kroz tuđe datoteke.
4. Ne koristi računar da ukradeš.
5. Ne koristi računar da svedočiš lažno.
6. Ne kopiraj i ne koristi softver za koji nisi platio.
7. Ne koristi tuđe računarske resurse bez odobrenja ili odgovarajuće nadoknade.
8. Ne prisvajaj tuđe intelektualne rezultate.
9. Misli o društvenim posledicama programa koji pišeš ili sistema koji dizajniraš.
10. Uvek koristi računar na načine koji osiguravaju brigu i poštovanje za druge ljude.

Navedene „zapovesti“ daju neke opšte smernice i razrešavaju neka, ali nikako sva moguća etička pitanja u vezi sa računarima i programiranjem.

Pitanja i zadaci za vežbu

Pitanje 10.1. *Koje sve aplikacije koriste „sisteme za preporučivanje“? Šta su moguće koristi a šta moguće štete od ovakvih sistema?*

Pitanje 10.2. *U kojim situacijama privatnost na internetu predstavlja moguću opasnost.*

Pitanje 10.3. *Šta opravdava a šta ne opravdava ograničavanje privatnosti na internetu?*

Pitanje 10.4. *Pronađi na internetu neke građanske grupe koje se zalažu za privatnost na internetu i kritički razmisli o njihovim stavovima.*

Pitanje 10.5. *Pronađi na internetu najznačajnije „uzbunjivače“ koji su ukazali na nelegalno prisluškivanje internet komunikacija.*

Glava 11

Pravni i ekonomski aspekti računarstva

Razni aspekti programiranja i primene računara pokrivene su zakonima i drugim sličnim normama. Ipak, i nakon istorije računarstva duge više od sedam decenija, mnoge situacije, na primer, u vezi sa autorskim pravima vezanim za softver, izazivaju dileme ili vode do kompleksnih sudskih procesa. To je samo jedan od mnogih pravnih i ekonomskih aspekata računarstva.

11.1 Intelektualna svojina i njena zaštita

Intelektualna svojina odnosi se na muzička, likovna, književna dela, simbole, dizajn, otkrića i pronalaske, itd. Intelektualna svojina štiti se raznovrsnim pravnim sredstvima, kao što su autorsko pravo ili kopirajt (engl. copyright), patenti, zaštićeni registrovani simboli (engl. trademarks), itd. Dok se autorsko pravo odnosi na lično, neotuđivo pravo autora i njegovog dela, kopirajt se odnosi na konkretno delo i može menjati vlasnika.

Autorska prava. Autorska prava na softver treba da spreče neovlašćeno korišćenje i kopiranje softvera. Nosilac autorskih prava ima pravo kopiranja, modifikovanja i distribuiranja softvera, što može odobriti i drugima. U Evropskoj uniji, na računarski softver polažu se autorska prava, obezbeđena zakonom, isto kao na, na primer, književna dela i to bez ikakve registracije ili formalne procedure. Takva autorska prava odnose se na sve aspekte kreativnosti autora, ali ne i ideje na kojima je program zasnovan, algoritme, metode ili matematičke pojmove koje koristi. Dakle, autorska prava štite samo program u formi (u izvornom kodu) u kojoj je napisan od strane programera. Funkcionalnost programa, programski jezik, format datoteka koje program koristi nisu zaštićeni autorskim pravom. S druge strane, grafika, zvuci i izgled programa mogu biti predmet

autorskih prava, a skup funkcionalnosti programa može biti zaštićen patentom.

U većini zemalja, podrazumeva se da autorska prava pripadaju autorima dela, pri čemu se pod autorima podrazumevaju poslodavci koji su svojim zaposlenim dali zadatak da naprave softver. Autorska prava na softver koji napravi zaposleni, ali ne po zadatku i uputstvima poslodavca (na primer, u slobodno vreme) pripadaju zaposlenom. Stvari postaju komplikovanije u slučaju kada poslodavac za razvoj softvera angažuje spoljašnjeg saradnika, a još komplikovanije ako je takav saradnik isporučio naručeni program a nije dobio dogovoreni honorar. Zbog takvih situacija, umesto oslanjanja na opšte zakone, bolje je unapred sklopiti namenski i precizan ugovor.

Patenti. U većini zemalja, računarski program ne može se registrovati patentom. Razlog je to što se svaki program, u manjoj ili većoj meri, oslanja na kumulativni razvoj i korišćenje tuđeg rada, ali i zbog toga što je teško ili nemoguće kontrolisati buduće korišćenje programa koji je patentiran. U Evropskoj uniji, patentom se ne može zaštititi računarski program kao takav, ali mogu pronalasci koji uključuju upotrebu računara i namenskog softvera.

Poverljivost. Ako se algoritam ne može zaštititi (od budućeg neovlašćenog korišćenja na navedene načine), onda se preporučuje da se on čuva u tajnosti i štiti na taj način. On se onda čini dostupnim za korišćenje ili modifikacije na vrlo ograničen način, ograničenom skupu osoba i u skladu sa ugovorom o neotkrivanju i o poverljivosti (engl. non-disclosure or confidentiality agreements; NDAs).

Razumevanje i primena pomenutih pravila i zakona često nije jednostavno i pravolinijsko, te su česti sudski sporovi o pojedinim programima, idejama i slično, a koji uključuju i softverske gigante kao što su Microsoft i Apple.

11.2 Kršenja autorskih prava

Kršenje autorskih prava (engl. copyright infringement) je korišćenje autorskog dela kao što je softver bez ovlašćenja, uključujući reprodukovanje, distribuiranje, modifikovanje, prodavanje i slično. Za kršenja autorskih prava često se koristi termin piraterija (engl. piracy). Neki od čestih razloga za kršenje autorskih prava su: cena, nedostupnost (na primer, u nekoj zemlji), pogodnost (na primer, ukoliko legalnu verziju nije moguće dobiti internetom), anonimnost (ukoliko je za legalnu verziju neophodno se identifikovati), itd. Slučajevi kršenja autorskih prava se često razrešavaju neposrednom pogodbom ili sudskim procesom.

Zbog digitalnog zapisa i interneta, kopiranje softvera i umetničkih dela često je veoma jednostavno i omogućava masovnu pirateriju. Procenjuje se da je u

2016. godini, čak oko 39% programa na personalnim računarima bilo nelicencirano. Na osnovu jedne velike ankete iz 2017. godine, čak 57% pojedinačnih korisnika barem ponekad koristi piratovani softver. Kompanija Google dobija dnevno oko dva i po miliona zahteva nosioca autorskih prava za uklanjanje linkova na piratske verzije njihovih proizvoda.

Autorska prava obično se mogu ignorisati i autorsko delo se može kopirati bez eksplicitnog ovlašćenja u nekim specijalnim nekomercijalnim situacijama koje se smatraju „fer korišćenjem“ (engl. fair use), na primer, u okviru predavanja (na primer, pravljenje nekoliko primeraka softvera za korišćenje na času), izveštavanja u medijima, naučnim istraživanjima i slično. Granica za fer korišćenje često nije jasna, pa se često pod maskom fer korišćenja distribuiraju čitava umetnička dela ili računarski programi.

11.3 Vlasnički i nevlasnički softver

Vlasnički softver (engl. proprietary software) je softver čiji je vlasnik pojedinac ili kompanija (obično neko ko je softver razvio), postoje oštra ograničenja za njegovo korišćenje i, gotovo isključivo, njegov izvorni kôd čuva se u tajnosti.

Softver koji nije vlasnički pripada obično nekoj od sledećih kategorija:

Šerver softver (engl. shareware software) distribuira se po niskoj ceni ili besplatno za svrhe probe i testiranje, ali zahteva plaćanje i registraciju za legalno, neograničeno korišćenje i neku vrstu tehničke podrške. Autorska prava na šerver softver zadržavaju originalni autori i nije dozvoljeno modifikovati ili dalje distribuirati softver.

Friver softver (engl. freeware software) se distribuira besplatno i to su obično mali pomoćni programi, bez obezbeđene tehničke podrške. Autorska prava na friver softver zadržavaju originalni autori.

Javni softver (engl. public software) ili softver javnog domena (engl. public domain software) ne podleže autorskim pravima, objavljuje se bez ikakvih ograničenja za njegovo korišćenje i nema nikakvu obezbeđenu tehničku podršku.

Softver otvorenog koda (engl. open source software) ili besplatni, slobodni softver (engl. free software)[1] je softver kojem se besplatno može pristupiti, koji se besplatno može koristiti, modifikovati i deliti (u originalnoj ili izmenjenoj formi) od strane bilo koga. Softver otvorenog koda obično je razvijan od strane velikog broja autora i distribuira se pod odgovarajućom licencom. Licence za otvoreni kôd oslanjaju se na autorska prava.

Softver otvorenog koda nije vlasnički softver (engl. proprietary softver), ali može biti komercijalan, tj. prodavati se. Ako je neko preuzeo neki

[1] Neki autori prave razliku između pojmova „open source software“ i „free software“, naglašavajući da se reč „free“ (koja ima više značenja) u ovom drugom odnosi na slobodu, slobodu upotrebe, a ne na cenu upotrebe.

softver pod licencom za otvoreni kôd, može da ga prodaje, ali je u principu u obavezi da ga distribuira pod istom tom licencom.

Licence za softver otvorenog koda odnose se na sledeća pitanja i kriterijume:

1. Slobodno redistribuiranje;
2. Raspoloživost programa u izvornom kodu;
3. Dozvola za dela izvedena iz originalne verzije;
4. Nepovredivost autorskog izvornog koda;
5. Nema diskriminisanja prema osobama ili grupama;
6. Nema diskriminisanja prema polju primene;
7. Redistribuiranje licence uz redistribuirani program;
8. Licenca ne može biti specifična za konkretan proizvod;
9. Licenca ne može da ograničava drugi softver;
10. Licenca mora da bude neutralna u odnosu na tehnologiju.

Neke od najčešće korišćenih okvira za licenciranje softvera otvorenog koda su GNU General Public License (GPL), Apache License, Creative Commons Licenses, itd.

11.4 Nacionalna zakonodavstva i softver

Iako postoje mnogi univerzalni principi u zakonskom regulisanju oblasti računarstva, za zakonodavstva mnogih zemalja postoje i brojne specifičnosti. U nekim zemljama zabranjene su određene internet usluge (na primer, u Kini je nedostupno internet pretraživanje putem pretraživača Google), u nekim zemljama zabranjen je uvoz ili izvoz nekih vrsta softvera (na primer, ograničen je izvoz kriptografskog softvera iz SAD), a u nekim zemljana zabranjen je izvoz nekih vrsta podataka ili njihova obrada u inostranstvu (u Evropskoj uniji, pravni akt GDPR, *The General Data Protection Regulation*, iz 2018. godine propisuje niz pravila o upravljanju ličnim podacima građana EU – o njihovoj zaštiti, kao i o strogoj kontroli izvoza ili obrade izvan zemalja EU). Zbog toga, u svim računarskim poslovima koji se sprovode u više zemalja neophodno je voditi računa o specifičnostima njihovih zakonodavstava.

11.5 Sajber kriminal

Sajber ili računarski kriminal (engl. cyber crime) je svaka protivzakonita aktivnost koja se sprovodi putem računara. U mnogim zemalja postoje zakoni koji se odnose na sajber kriminal i kazne mogu da budu novčane ili zatvorske u trajanju i do 20 godina. Zakoni obično tretiraju dela sajber kriminala učinjena za sticanje komercijalne prednosti, za lični novčani dobitak ili za pripremu

kriminalne radnje. Neka od sredstava kojima se radnje sajber kriminala mogu sprovesti su: krađa identiteta, zloupotreba mejla i spam, neovlašćeni upadi u sisteme, distribuiranje nelegalnih sadržaja, itd.

Sajber kriminal javlja se na ogromnoj skali i u raznim vidovima, počev od pojedinačnih prevara, pa do napada na najvišem nivou, kada jedna država vrši upade u računarske sisteme druge države.

Pored sajber kriminala, postoje i mnoge vrste sajber nasilja (engl. cyberbullying), kao što je vršnjačko sajber nasilje, sajber uhođenje i slično, koje se takođe tretiraju zakonima u mnogim zemljama, kao i pratećim policijskim jedinicima za sajber kriminal.

11.6 Računarstvo, produktivnost i nezaposlenost

Kao što je slučaj i sa mnogim drugim tehnologijama, pojava i razvoj računara i njihovih primena doveli su do otpuštanja mnogih ljudi i gubljenja mnogih poslova. Međutim, to je samo jedan aspekt i pravo pitanje je da li razvoj računara dovodi do ukupno većeg ili manjeg broja radnih mesta za ljude. Slična dilema postojala je i u vreme industrijske revolucije, kada su mašine počele da u proizvodnji zamenjuju ljude i kada su, početkom devetnaestog veka, u strahu od nezaposlenosti, pripadnici ludističkog pokreta uništavali proizvodne mašine. Vreme je, međutim, pokazalo da su, na duže staze, pojava i razvoj mašina omogućile veliki porast produktivnosti, porast proizvodnje i veliki porast radnih mesta. Zato mnogi smatraju da će računari, iako će na mnogim radnim mestima odmeniti ljude, zapravo stvoriti mnogo više novih radnih mesta – za razvoj tih računara i upravljanje njima, kao i za čitav niz usluga koje danas i ne postoje na tržištu.

11.7 Kriptovalute

Od pre nekoliko godina postoji još jedan način na koji računari utiču na globalnu ekonomiju – kripto valute. Kripto valute su digitalno sredstvo plaćanja. Za kontrolu stvaranja novih jedinica valute, kao i za sigurnost i registrovanje transakcija koriste se kriptografski algoritmi. Za razliku od klasičnih nacionalnih valuta koje izdaju centralne banke, kripto valute nemaju centralnog izdavača niti centralizovanu kontrolu toka. Kontrola svake kriptovalute odvija se kroz blokčejn (engl. blockchain) tehnologiju, kao javnu, distribuiranu bazu podataka. Ta baza podataka ne čuva se na jednoj lokaciji i ne postoji centralizovana verzija koja može biti oštećena ili uništena. Ta baza podataka zapravo je deljena između miliona računara i dostupna svakome putem interneta. U okviru blokčejna, registruje se svaka transakcija u vidu novog bloka i, zahvaljujući kriptografskoj zaštiti, ne može kasnije biti izbrisana.

Bitcoin je prva kripto valuta, kreirana 2009. godine. Od tada se pojavilo na stotine drugih kripto valuta. Uprkos mnogim turbulencijama na tržištu kripto valuta, njihova vrednost tokom prethodnih godina je uglavnom rasla. Teško je predvideti njihovu dugoročnu sudbinu.

Pitanja i zadaci za vežbu

Pitanje 11.1. *Za svoje omiljene aplikacije proveri pod kojom licencom se distribuiraju.*

Pitanje 11.2. *Istraži na internetu koje su kriptovalute trenutno najpopularnije.*

Pitanje 11.3. *Istraži na internetu najznačajnije sajber napade među različitim zemljama. Kakav je bio stav vlada tim zemalja u vezi sa tim napadima?*

Pitanje 11.4. *Istraži na internetu osnovne principie blokčejn tehnologije.*

Literatura

[1] ISO Committee. *Standardi jezika C.* http://www.open-std.org/JTC1/SC22/WG14/www/projects#9899 2021.

[2] Brian Kernighan, Dennis Ritchie. *The C Programming Language.* Prentice Hall, 1978.

[3] Brian Kernighan, Rob Pike. *The Practice of Programming.* Addison-Wesley, 1999.

[4] K. N. King. *C Programming: A Modern Approach (2nd edition).* W. W. Norton, 2008.

[5] Filip Marić, Predrag Janičić. *Programiranje 1 — Osnove programiranja kroz programski jezik C.* Matematički fakultet, 2015.

[6] Gordana Pavlović-Lažetić, Duško Vitas. *Osnovi programiranja/Programiranje 1/2, skripta.* Matematički fakultet, 2005.

[7] Milena Vujošević Janičić, Jelena Graovac, Nina Radojičić, Ana Spasić, Mirko Spasić, Anđelka Zečević. *Programiranje 2 — Zbirka zadataka sa rešenjima.* Matematički fakultet, 2016.