Programiranje II

Beleške za predavanja

Smer *Informatika*
Matematički fakultet, Beograd

Filip Marić i Predrag Janičić

2011.

Sadržaj

Deo I

Osnovi algoritmike

Glava 1

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:

- Roditelj osobe je predak te osobe (bazni slučaj)
- Roditelj bilo kog pretka neke osobe je takođe predak te osobe (rekurzivni korak)

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].

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 imaju sličan oblik.

- 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

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

slučaju izlaska iz rekurzije razrešavaju slučaj nule, dok u slučaju nekog broja većeg od nule rekruzivno 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, niska čini zapis prirodnog broja ako ili (i) dekadna cifra (bazni slučaj) ili (ii) dobija se nadovezivanjem dekande cifre sa desne strane zapisa prirodnog broja (induktivni korak). U tom slučaju primitivnom rekurzijom je moguće definisati funkcije tako da izlaz iz rekurzije predstavljaju jednocifreni brojevi, dok se u slučaju višecifrenih brojeva `n` rekurzivno razrešava slučaj broja sa odsečenom poslednjom cifrom (`n/10`), najčešće kombinujući dobijeni rezultat sa poslednjom cifrom (`n%10`). Dualni pristup razlaganja broja na prvu cifru i ostale nije pogodan, zbog toga što ovakvo razlaganje nije moguće izračunati jednostavnim matematičkim operacijama (kakvo je deljenje sa ostatkom sa 10).

Dalje, na primer, niz je moguće definisati tako što (i) prazan niz predstavlja niz (bazni slučaj) i (ii) niz dobijen dodavanjem elementa na kraj nekog niza predstavlja niz. Pri ovom razmatranju, primitivnom rekurzijom je moguće definisati funkcije tako da pri izlasku iz rekurzije obrađuju slučaj praznog niza, dok slučaj nepraznog niza dužine `n` rešavaju tako što rekurzivno razreše njegov prefiks dužine `n-1` i onda rezultat iskombinuju sa poslednjim elementom niza. Dualno razlaganje na prvi element i sufiks niza je takođe moguće, pri čemu je onda prilikom svakog rekruzivnog poziva potrebno prosleđivati pored indeksa 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, namesto primitivne rekurzije, dozvoljeno je pisati funkcije koje su *totalno (generalno) rekurzivne*. Tako je npr. 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, često razmatranje niza kao unije dva njegova duplo kraća podniza vodi boljoj efikasnosti nego primitivno-rekurzivno razmatranje niza kao unije jednog njegovog elementa i podniza za jedan kraćeg.

1.1 Primeri rekurzivnih funkcija

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.

1.1.1 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:

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

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

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

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).

1.1.2 Sumacija niza

Sumacija niza brojeva može biti izražena (primitivno) rekurzivno.

- $\sum_{i=1}^{0} a_i = 0$ (bazni slučaj)
- 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 sum(float a[], unsigned n) {
  if (n == 0)
     return 0.0f;
  else
     return sum(a, n-1) + a[n-1];
}
```

1.1.3 Stepenovanje

Stepenovanje broja može biti izraženo (primitivno) rekurzivno.

- $x^0 = 1$
- za $n > 0$ važi: $x^n = x \cdot x^{n-1}$.

Vrednost x^k za nenegativne celobrojne izložioce može se jednostavno izračunati sledećom funkcijom:

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

Iterativna varijanta prethodne funkcije je:

```
float stepen(float x, unsigned k) {
    unsigned i; float m = 1.0f;
    for(i=0; i<k; i++)
        m *= x;
    return m;
}
```

Sledeće rekurzivno rešenje je znatno brže (zahteva mnogo manje množenja, što će kasnije i biti dokazano):

```
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);
}
```

Iterativnu verziju prethodne funkcije nije trivijalno napisati.

1.1.4 Fibonačijev niz

Fibonačijev niz (0,1,1,2,3,5,8,13,...) može se definisati u vidu (generalne) rekurzivne funkcije F:

- $F(0) = 0$ i $F(1) = 1$ (bazni slučaj)
- za $n > 1$ važi: $F(n) = F(n-1) + F(n-2)$ (rekurzivni korak)

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);
}
```

1.1.5 NZD

Euklidov algoritam za izračunavanje najvećeg zajedničkog delioca dva broja se može izračunati narednom rekurzivnom funkcijom.

```
unsigned nzd(unsigned a, unsigned b) {
  if (b == 0)
     return a;
  else
     return nzd(b, a % b);
}
```

1.1.6 Kule Hanoja

Problem kula Hanoja[2] glasi ovako: data su tri tornja i na prvom od njih n diskova opadajuće veličine; zadatak je prebaciti sve diskove sa prvog na treći toranj (koristeći i drugi) ali tako da nikada nijedan disk ne stoji iznad manjeg.

Iterativno rešenje ovog problema je veoma 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 pomoćni toranj (korišćenjem dolaznog tornja kao pomoćnog), prebaci najveći disk sa polaznog na dolazni toranj i, konačno, prebaci (rekurzivno) $n-1$ diskova sa pomoćnog na dolazni disk (korišćenjem polaznog tornja kao pomoćnog). U nastavku je implementacija ovog rešenja:

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

```
void tower(unsigned n, char start, char finish, char spare) {
  if (n > 0) {
     tower(n-1,start,spare,finish);
     printf("Prebaci disk sa kule %c na kulu %c\n",start,finish);
     tower(n-1,spare,finish,start);
  }
}
```

Poziv navedene funkcije `tower(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
```

1.2 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*, kao u narednom primeru:

```
void A(int n) {
  if (n <= 0)
    return;
  n--;
  B(n);
}

void B(int n) {
  if (n <= 0)
    return;
  n--;
  A(n);
}
```

1.3 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, ovo može

biti veoma memorijski i vremenski zahtevno, te je poželjno rekurzivno rešenje zameniti iterativnim.

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

Razmotrimo, na primer, izvršavanje navedene funkcije koja izračunava elemente Fibonačijevog niza za $n = 5$. U okviru tog izvršavanja, funkcija f je 3 puta pozvana za $n = 0$, 5 puta za $n = 1$, itd. Naravno, na primer, poziv $f(1)$ je svaki put izvršavan iznova i nije korišćena vrednost koju je vratio prethodni takav poziv. 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.

```
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);
}
```

Drugi pristup rešavanja problema suvišnih izračunavanja naziva se *dinamičko programiranje*. Ključna ideja je da se, umesto da se analizirajući problem on svodi na niže ka jednostavnijim potproblemima, od jednostavnih potproblema naviše konstruiše rešenje glavnog problema. Rešenja dinamičkim programiranjem obično ne uključuju rekurziju.

Na primer, gore navedena funkcija `fib` može se zameniti iterativnom funkcijom koja od od početnih elemenata niza postepeno sintetiše sve dalje i dalje elemente niza. Primetimo da za izračunavanje n-tog elementa niza nije neophodno pamtiti sve elemente niza do indeksa `n` već samo dva prethodna:

```
int fib(int n) {
    int fpp, fp;

    if (n == 0 || n == 1)
        return n;

    fpp = 0;
    fp = 1;

    for(i = 3; i <= n; i++) {
        int f = fpp + fp;
        fpp = fp;
        fp = f;
    }

    return fp;
}
```

1.4 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[3].

Međutim, takav postupak postoji za neke specijalne slučajeve. Naročito je zanimljiv slučaj *repne rekurzije* jer se u tom slučaju rekurzija može jednostavno eliminisati[4]. Rekurzivni poziv je *repno rekurzivni* ukoliko je vrednost rekurzivnog poziva upravo i konačan rezultat funkcije, tj. nakon rekurzivnog poziva funkcija ne vrši nikakvo dodatno izračunavanje. U tom slučaju, nakon rekruzivnog poziva nema potrebe vraćati se u kôd pozivne funkcije, te nema potrebe na stek smeštati trenutni kontekst (vrednosti lokalnih promenljivih). 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

[3] Opšti postupak bi uključivao implementaciju sturkture podataka na koju bi eksplicitno bile smeštani podaci koji se inače smeštaju na programski stek.

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

izlasku iz rekurzije neophodno još pomnožiti rezultat sa `x`).

Pokažimo na narednom kôdu kako je moguće eliminisati repnu rekurziju.

```
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 vrednost parametara funkcije (u ovom slučaju promenljive `x`) postavi na vrednost parametra rekruzivnog 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);
}
```

Demonstrirajmo tehniku ulanjanja 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 promenljivih `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.

```
unsigned nzd(unsigned a, unsigned b) {
  while (b != 0)
     unsigned tmp = a % b; a = b; b = tmp;
  }
  return a;
}
```

Pitanja i zadaci za vežbu

Pitanje 1. *1. Napisati rekurzivnu funkciju koja za dato n iscrtava trougao dimenzije n:*

```
(a)          (b)          (c)          (d)
+             +            +           +
++           +++          +++          ++
+++         +++++        +++++         +
```

2. *Napisati rekurzivnu funkciju koja prikazuje sve cifre datog celog broja i to: (a) s leva na desno; (b) s desna na levo.*

3. *Napisati rekurzivnu funkciju koja određuje binarni (oktalni, heksadekadni) zapis datog celog broja.*

4. *Napisati rekurzivnu funkciju koja izračunava zbir cifara datog celog broja.*

5. *Napisati rekurzivnu funkciju koja izračunava broj cifara datog celog broja.*

6. *Napisati rekruzivnu funkciju koja izračunava broj parnih cifara datog celog broja.*

7. *Napisati rekurzivnu funkciju koja izračunava najveu cifru datog broja.*

8. *Napisati rekurzivnu funkciju koja uklanja sva pojavljivanja date cifre iz datog broja.*

9. *Napisati rekurzivnu funkciju koja kreira niz cifara datog celog broja.*

10. *Napisati rekurzivnu funkciju koja obrće cifre datog celog broja.*

11. *Napisati rekurzivnu funkciju koja obrće niz brojeva.*

12. *Napisati rekurzivnu funkciju koja ispituje da li su elementi nekog niza brojeva poređani palindromski (isto od napred i od pozadi).*

13. *Napisati rekruzivnu funkciju koja obrće nisku.*

14. *Napisati rekurzivnu funkciju koja izbacuje sve parne cifre datog celog broja.*

15. *Napisati rekurzivnu funkciju koja iz datog broja izbacuje svaku drugu cifru.*

16. *Napisati rekurzivnu funkciju koja posle svake neparne cifre datog broja dodaje 0.*

17. *(a) Napisati rekurzivnu funkciju koja štampa brojeve između* 0 *i* n.

    (b) Napisati rekurzivnu funkciju koja štampa brojeve između n *i* 0.

18. *Napisati rekurzivnu funkciju koja izračunava sumu prvih* n *prirodnih brojeva.*

19. *Korišćenjem identiteta* $\sqrt{4 \cdot x} = 2 \cdot \sqrt{x}$ *napisati rekurzivnu funkciju koja izračunava ceo deo korena datog broja.*

20. *Napisati rekurzivnu funkciju koja izračunava vrednost binomnog koeficienta* $\binom{n}{k}$.

21. *Napisati rekurzivnu funkciju koja određuje maksimum niza celih brojeva.*

22. *Napisati rekurzivnu funkciju koja izračunava skalarni proizvod dva data vektora (predstavljena nizovima dužine* n*).*

23. *Napisati repno-rekurzivnu funkciju koja izračunava* $n!$.

Glava 2

Složenost algoritama

Pored svojstva ispravnosti programa, veoma je važno i pitanje koliko program zahteva vremena (ili izvršenih instrukcija) i memorije za svoje izvršavanje. Često se algoritmi ne izvršavaju isto niti zahtevaju isto memorije za sve ulazne vrednosti, pa je potrebno naći način za opisivanje i poređenje *efikasnosti* (ili *složenosti*) različitih algoritama. Obično se razmatraju:

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

2.1 O notacija i red algoritma

Neka je funkcija $f(n)$ jednaka broju instrukcija koje zadati algoritam izvrši za ulaz veličine n. Naredna tabela prikazuje potrebno vreme izvršavanja algoritma ako se pretpostavi da jedna instrukcija traje jednu nanosekundu.

Vremenska složenost algoritma određuje njegovu praktičnu upotrebljivost tj. najveću dimenziju ulaza za koju je moguće da će se algoritam izvršiti u nekom razumnom vremenu. Analogno važi i za prostornu složenost. Iz navedene tabele je jasno da vodeći član u funkciji $f(n)$ skoro potpuno određuje potrebno vreme izvršavanja. Tako, ako je broj instrukcija $n^2 + 2n$, onda za ulaz dimenzije 1000000, član n^2 odnosi 16.7 minuta dok član $2n$ odnosi samo dodatne dve mikrosekunde. 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 (multiplikativni i aditivni) konstantni faktori u broju potrebnih instrukcija, koliko asimptotsko ponašanje broja instrukcija u zavisnosti od veličine ulaza. Ovaj koncept se preciznije uvodi sledećom definicijom.

Definicija 2.1. *Ako postoje pozitivna konstanta c i prirodan broj N takvi da za funkcije f i g nad prirodnim brojevima važi*

$$f(n) \leq c \cdot g(n) \text{ za sve vrednosti } n \text{ veće od } N$$

$n \setminus f(n)$	$\log n$	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	$0.006\,\mu s$	$0.05\,\mu s$	$0.282\,\mu s$	$2.5\,\mu s$	$13\,dan$	
100	$0.007\,\mu s$	$0.1\,\mu s$	$0.644\,\mu s$	$10\,\mu s$	$4 \times 10^{13}\,god$	
1,000	$0.010\,\mu s$	$1.0\,\mu s$	$9.966\,\mu s$	$1\,ms$		
10,000	$0.013\,\mu s$	$10\,\mu s$	$130\,\mu s$	$100\,ms$		
100,000	$0.017\,\mu s$	$0.10\,\mu s$	$1.67\,ms$	$10\,s$		
1,000,000	$0.020\,\mu s$	$1\,ms$	$19.93\,ms$	$16.7\,min$		
10,000,000	$0.023\,\mu s$	$0.01\,s$	$0.23\,s$	$1.16\,dan$		
100,000,000	$0.027\,\mu s$	$0.10\,s$	$2.66\,s$	$115.7\,dan$		
1,000,000,000	$0.030\,\mu s$	$1\,s$	$29.9\,s$	$31.7\,god$		

onda pišemo

$$f = O(g)$$

i čitamo "f je veliko 'o' od g"[1].

Naglasimo da O nije funkcija — O označava klasu funkcija. Lako se pokazuje da aditivne i multiplikativne konstante ne utiču na klasu kojoj funkcija pripada.

Primer 2.1. *Važi:*

- $n^2 = O(n^2)$
- $n^2 + 10 = O(n^2)$
- $10 \cdot n^2 + 10 = O(n^2)$
- $10 \cdot n^2 + 8n + 10 = O(n^2)$
- $n^2 = O(n^3)$
- $2^n = O(2^n)$
- $2^n + 10 = O(2^n)$
- $10 \cdot 2^n + 10 = O(2^n)$
- $2^n + n^2 = O(2^n)$
- $3^n + 2^n = O(3^n)$
- $2^n + 2^n n = O(2^n n)$

Često se algoritmi ne izvršavaju isto za sve ulaze istih veličina, pa je potrebno naći način za opisivanje i poredjenje 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). Ta procena može da bude varljiva, ali ne postoji bolji opšti način za poredjenje efikasnosti algoritama. Čak i kada bi uvek bilo moguće izračunati prosečno vreme izvršavanja algoritma, i takva procena bi često mogla da bude

[1] S obzirom da će oznaka O biti korišćena samo u kontekstu nizova, podrazumevaće se da $n \to +\infty$

varljiva. Analiziranje najboljeg slučaja, naravno, ima još manje smisla. U nastavku će, ako nije rečeno drugačije, biti podrazumevana analiza najgoreg slučaja.

Definicija 2.2. *Ako je $T(n)$ vreme izvršavanja algoritma A (čiji ulaz karakteriše prirodan broj n) i ako važi $T = O(g)$, onda kažemo da je algoritam A* složenosti *(ili* reda*) g i da pripada klasi $O(g)$.*

Neki autori u prethonoj definiciji podrazumevaju da je funkcija g asimptotski najmanja takva (zanemarujući aditivne i multiplikativne konstante) da važi $T = O(g)$. Takav dodatni uslov obezbeđuje da algoritam koji zahteva, na primer, $5n$ koraka jeste složenosti $O(n)$, ali ne i složenosti $O(n^2)$. S druge strane, na osnovu navedene definicije, za taj algoritam može da se smatra da je složenosti $O(n)$ ali i složenosti $O(n^2)$ (mada ne i složenosti, na primer, $O(\log n)$.

Kao što je rečeno, aditivne i multiplikativne konstante nisu relevantne u asimptotskoj analizi složenosti algoritama. Zato se može reći da je algoritam složenosti $O(n^2)$, ali ne i $O(5n^2 + 1)$, jer aditivna konstanta 1 i multiplikativna 5 nisu od suštinske važnosti. Zaista, ako jedan algoritam zahteva $5n^2 + 1$ instrukcija, a drugi n^5, 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 da se prvi algoritam izvršava brže od drugog za svaku veličinu ulaza.

Za algoritme složenosti $O(n)$ kažemo da imaju linarnu složenost, za $O(n^2)$ kvadratnu, za $O(n^3)$ kubnu, za $O(n^k)$ za neko k polinomijalnu, a za $O(\log n)$ logaritamsku.

2.2 Izračunavanje složenosti funkcija

Izračunavanje (vremenske i prostorne) složenosti funkcija se zasniva na određivanju tačnog ili približnog broja instrukcija koje se izvršavaju i memorijskih jedinica koje se koriste. Tačno određivanje tih vrednosti je najčešće veoma teško ili nemoguće, te se obično koriste razna pojednostavljivanja. Na primer, u ovom kontekstu pojam „jedinična instrukcija“ se obično pojednostavljuje, te se može smatrati da, na primer, i poredjenje i sabiranje i množenje i druge pojedinačne naredbe troše po jednu vremensku jedinicu. Ono što je važno je da takva pojednostavljivanja ne utiču na klasu složenosti kojoj algoritam pripada (jer, kao što je rečeno, konstantni faktori ne utiču na red algoritma).

2.2.1 Rekurentne jednačine

Kod rekurzivnih funkcija, vreme $T(n)$ potrebno za izračunavanje vrednosti funkcije za ulaz dimenzije n se može izraziti kao zbir vremena izračunavanja za sve rekurzivne pozive za ulaze manje dimenzije i vremena potrebnog za pripremu rekurzivnih poziva i objedinjavanje rezultata. Tako se, obično jednostavno, može napisati veza oblika

$$T(n) = T(n_1) + \ldots + T(n_k) + c,$$

gde rekurzivna funkcija za ulaz dimenzije n vrši k rekurzivnih poziva za ulaze (ne obavezno različitih) dimenzija $n_1, \ldots, n_k$, dok je vreme potrebno za pripremu poziva i objedinjavanje rezultata c.

U nekim slučajevima iz ovakve *linearne rekurentne relacije* može se eksplicitno izračunati nepoznati niz $T(n)$. U nekim slučajevima eksplicitno rešavanje jednačine nije moguće, ali se može izračunati asimptotsko ponašanje niza $T(n)$.

Homogena 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 $T(n) = ca^n$.

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 da i nula niz (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 vrednosti 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 rekurentntu jednačinu drugog reda.*

Ako su t_1 i t_2 različiti korenovi ove jednačine, 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 rešesnja. 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 $n = 1$.

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

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

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

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

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

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, $\ldots$, k).

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.

Nehomogena rekurentna jednačina prvog reda. Razmotrimo jednačinu oblika

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

za $n > 0$, pri čemu je data vrednost $T(0) = c$.

Definišimo niz $T'(n) = T(n+1) - T(n)$ za $n \geq 0$. Za $n > 0$ važi $T'(n) = (aT(n)+b) - (aT(n-1)+b) = aT'(n-1)$. Za $n = 0$ važi $T'(0) = T(1) - T(0) = ac + b - c$. Dakle, niz $T'(n)$ zadovoljava homogenu jednačinu prvog reda čije je rešenje $T'(n) = ((ac + b - c)a^n$.

Za $a \neq 1$ važi

$$\begin{aligned} T(n) &= T(0) + \sum_{k=0}^{n-1} T'(k) = c + \sum_{k=0}^{n-1} ((ac+b-c)a^k \\ &= c + (ac+b-c)\frac{a^n-1}{a-1} = b\frac{a^n-1}{a-1} + ca^n \end{aligned}$$

Drugi 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 $T(n) = aT(n-1)+b$ i $T(n+1) = aT(n)+b$, 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 novodobijeme homogene jednačine se može dobiti na gore opisani način (jer su poznate i početne vrednosti $T(0) = c$ i $T(1) = ac+b$).

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

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

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

Nehomogena rekurentna jednačina oblika $T(n) = aT(n/b) + cn^k$. Naredna teorema govori o asimptotskom ponašanju rešenja nehomogene rekurentne jednačine oblika $T(n) = aT(n/b) + cn^k$.

Teorema 2.1 (Master teorema)**.** *Rešenje rekurentne relacije*

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

gde su a i b celobrojne konstante ($a \geq 1$, $b \geq 1$) i c i k pozitivne konstante je

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

2.2.2 Faktorijel

Izračunajmo vremensku složenost naredne rekurzivne funkcije:

```
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n*factorial(n-1);
}
```

Neka $T(n)$ označava broj instrukcija koje zahteva poziv funkcije `f` za ulaznu vrednost `n`. Za $n = 0$, važi $T(n) = 2$ (jedno poređenje i jedna naredba `return`). Za $n > 0$, važi $T(n) = 3 + T(n-1)$ (jedno poređenje, jedno množenje i broj instrukcija koje zahteva funkcija `f` za argument `n-1`. Dakle,

$$T(n) = 3+T(n-1) = 3+3+T(n-2) = \ldots = \underbrace{3+3+\ldots+3}_{n}+T(0) = 3n+2 = O(n)$$

Dakle, navedena funkcija ima linearnu vremensku složenost.

Nehomogena jednačina $T(n) = 3+T(n-1)$ je mogla biti rešena i svođenjem na homogenu jednačinu. Iz $T(n) = 3 + T(n-1)$ i $T(n+1) = 3 + T(n)$ sledi $T(n+1)-T(n) = T(n)-T(n-1)$ i $T(n+1) = 2T(n)-T(n-1)$. Karakteristična jednačina $t^2 = 2t - 1$ ima dvostruko rešenje $t_1 = 1$, pa je opšte rešenje oblika $T(n) = a \cdot 1^n + b \cdot n \cdot 1^n = a + nb = O(n)$.

2.2.3 Fibonačijev niz

Za elemente Fibonačijevog niza važi $F(0) = 0$, $F(1) = 1$ i $F(n) = F(n-1) + F(n-2)$, 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

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

Koristeći početne uslove, može se izračunati opšti član niza:

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

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

```
int fib(int n) {
    if(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 $T(n) = 2$ (jedno poređenje i jedna naredba `return`). Za $n > 1$, važi $T(n) = T(n-1) + T(n-2) + 3$ (jedno poređenje i broj instrukcija koje zahtevaju pozivi funkcije za $n-1$ i $n-2$). Iz $T(n) = T(n-1) + T(n-2) + 3$ i $T(n+1) = T(n) + T(n-1) + 3$, 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 $T(n) = O(\left(\frac{1+\sqrt{5}}{2}\right)^n)$.

2.2.4 Stepenovanje

Izračunajmo vremensku složenost naredne rekurzivne funkcije za stepenovanje u zavisnosti od vrednosti `k` (jer je ona ključna u ovom primeru):

```
int stepen_brzo(int n, int 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);
}
```

Neka $T(k)$ označava broj instrukcija koje zahteva poziv funkcije `stepen_brzo` za ulaznu vrednost `k` (za `k`≥ 0). Za $k = 0$ važi $T(k) = 2$ (jedno poređenje i jedna naredba `return`). Za $k > 1$, ako je n neparan važi $T(k) = T(k-1) + 4$, a ako je paran, važi $T(k) = T(k/2) + 4$. Ukoliko se uvek koristi druga grana, onda na osnovu teoreme 2.1 važi $T(k) = O(\log k)$. Međutim, ovo je za algoritam najbolji slučaj. Najgori je ako se uvek koristi prva grana pa iz veze $T(k) = T(k-1) + 4$ sledi da je $T(k) = O(k)$. No, analizom kôda može se zaključiti da je scenario da se uvek koristi prva grana nemoguć (jer ako je k neparan broj, onda je $k-1$ paran). Dublja analiza pokazuje da je navedeni algoritam složenosti $O(\log k)$.

2.2.5 Kule Hanoja

Izračunajmo broj prebacivanja diskova kod naredne funkcije:

```
void tower(int n, char start, char finish, char spare) {
  if (n > 0) {
     tower(n-1,start,spare,finish);
     printf("Prebaci disk sa kule %c na kulu %c\n",start,finish);
     tower(n-1,spare,finish,start);
  }
}
```

Važi $T(0) = 0$ i $T(n) = 2T(n-1) + 1$ (i $T(1) = 2T(0) + 1 = 1$). Iz $T(n) - 2T(n-1) = 1 = T(n-1) - 2T(n-2)$ (za $n > 1$) sledi $T(n) = 3T(n-1) - 2T(n-2)$. Karakteristična jednačina ove veze je $t^2 = 3t - 2$ i njeni koreni su 2 i 1. Iz sistema

$$\alpha \cdot 1 + \beta \cdot 1 = 0$$

$$\alpha \cdot 2 + \beta \cdot 1 = 1$$

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

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

2.2.6 Uzajamna rekurzija

Složenost algoritama u kojima se javlja uzajamna rekurzija može se izračunati svođenjem na prikazane tehnike. To će biti ilustrovano primerom.

Algoritam A izvršava se za vrednost n ($n > 1$) primenom istog algoritma za vrednost $n-1$, pri čemu se za svodjenje problema koristi algoritam B za vrednost $n-1$. Algoritam B izvršava se za vrednost n ($n > 1$) trostrukom primenom istog algoritma za vrednost $n-1$, pri čemu se za svodjenje problema koristi algoritam A za vrednost $n-1$. Algoritmi A i B se za $n = 1$ izvršavaju jednu vremensku jedinicu. Izračunati vreme izvršavanja algoritma A za ulaznu vrednost n.

Neka je $A(n)$ vreme izvršavanja algoritma A za ulaznu vrednost n i neka je $B(n)$ vreme izvršavanja algoritma B za ulaznu vrednost n. Na osnovu uslova zadatka važi:

$$A(1) = B(1) = 1 \tag{2.1}$$

$$A(n) = A(n-1) + B(n-1) \tag{2.2}$$

$$B(n) = 3B(n-1) + A(n-1) \ \ (n > 1) \tag{2.3}$$

Iz jednakosti (2) imamo:

$$B(n) = A(n+1) - A(n) \tag{2.4}$$

$$B(n-1) = A(n) - A(n-1) \tag{2.5}$$

pa, uvrštavanjem u jednakost (3), za $n > 1$ važi:

$$A(n+1) - 4A(n) + 2A(n-1) = 0 \tag{2.6}$$

Pomoću ove rekurentne veze i početnih uslova: $A(1) = 1$ i $A(2) = A(1) + B(1) = 2$, možemo odrediti vrednost $A(n)$. Karakteristična jednačina relacije (6) je:

$$x^2 - 4x + 2 = 0$$

Njeni koreni su $x_1 = 2 + \sqrt{2}$ i $x_2 = 2 - \sqrt{2}$, pa je opšte rešenje rekurentne jednačine (6) oblika:

$$A(n) = c_1(2 + \sqrt{2})^n + c_2(2 - \sqrt{2})^n$$

za neke $c_1, c_2 \in \mathbf{R}$. Konstante c_1 i c_2 odredjujemo pomoću početnih uslova, rešavajući sledeći sistem linearnih jednačina po c_1 i c_2:

$$1 = A(1) = c_1(2 + \sqrt{2}) + c_2(2 - \sqrt{2})$$

$$2 = A(2) = c_1(2 + \sqrt{2})^2 + c_2(2 - \sqrt{2})^2$$

Dobija se da je $c_1 = \frac{1}{4}(2 - \sqrt{2})$, a $c_2 = \frac{1}{4}(2 + \sqrt{2})$, pa je konačno:

$$A(n) = \frac{1}{2}((2 + \sqrt{2})^{n-1} + (2 - \sqrt{2})^{n-1})$$

2.3 Klase složenosti P i NP

Neka od najvažnijih otvorenih pitanja matematike i informatike vezana su za složenost izračunavanja i, takozvane, NP-kompletne probleme. 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 *true* ili *false*, tj. $\top$ ili $\bot$), navedeni problem može biti formulisan na sledeći način: da li je *zadovoljiv* logički iskaz

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

Zadovoljivost navedenog iskaza može biti odredjena tako što bi bile ispitane sve moguće interpretacije – sve moguće dodele varijablama p, q i r. Ako je u nekoj interpretaciji vrednost datog logičkog iskaza *true*, onda je dati iskaz zadovoljiv. Za izabranu, fiksiranu interpretaciju može se u konstantnom vremenu utvrditi da li je istinitosna vrednost *true*. Za tri promenljive ima 2^n interpretacija, pa je red ovog algoritma za ispitivanje zadovoljivosti logičkih iskaza reda $O(2^n)$ (algoritam je eksponencijalne složenosti). Pitanje je da li postoji algoritam koji navedeni problem rešava u polinomijalnom vremenu.

Definicija 2.3. *Za algoritam sa ulaznom vrednošću n kažemo da je* polinomijalne složenosti *ako je njegovo vreme izvršavanja $O(P(n))$ gde je $P(n)$ polinom po n. Klasu polinomijalnih algoritama označava se sa P.*

Definicija 2.4 (Pojednostavljena definicija klase NP)**.** *Ako neki problem može da se predstavi u vidu najviše eksponencijalno mnogo instanci i ako za bilo koju instancu može da bude rešen u polinomijalnom vremenu, onda kažemo da problem pripada klasi NP.*

Definicija 2.5 (Formalna definicija klase NP)**.** *Ako je U skup svih mogućih ulaznih vrednosti za neki problem odlučivanja (za koji je potrebno dati odgovor* da *ili* ne*) i $L \subseteq U$ skup svih ulaznih vrednosti za koje je odgovor na problem potvrdan, onda L zovemo* jezik koji odgovara problemu.

Kažemo da nedetrministički algoritam *prepoznaje jezik L ako važi: za zadate ulazne vrednosti x, moguće je pretvoriti svaki nd-izbor koji se koristi tokom izvršavanja algoritma u stvarni izbor takav da će algoritam prihvatiti x ako i samo ako je $x \in L$.*

Klasu svih problema za koje postoje nedeterministički algoritmi čije je vreme izvršavanja je polinomijalne složenosti zovemo NP klasa.

Očigledno je da važi $P \subseteq NP$, ali se još uvek ne zna[2] da li važi $P = NP$.

[2]Matematički institut Klej nudi nagradu od milion dolara za svaki od izabranih sedam

Ako bi se pokazalo da neki problem iz klase NP nema polinomijalno rešenje, onda bi to značilo da ne važi $P = NP$. Ako neki problem iz klase NP ima polinomijalno rešenje, onda to još ne znači da važi $P = NP$. Za probleme iz posebne potklase klase NP (to je klasa NP-kompletnih problema) važi da ako neki od njih ima polinomijalno rešenje, onda važi $P = NP$.

Primer 2.2. *Problem ispitivanja zadovoljivosti logičkih iskaza (SAT) pripada klasi NP, ali se ne zna da li pripada klasi P.*

Definicija 2.6. *Za problem X kažemo da je NP-težak problem ako je svaki NP problem polinomijalno svodljiv na X.*

Definicija 2.7. *Za problem X kažemo da je NP-kompletan problem ako pripada klasi NP i ako je NP-težak.*

Teorema 2.2. *Ako bilo koji NP-težak problem pripada klasi P, onda važi $P = NP$.*

Teorema 2.3. *Problem X je NP-kompletan ako*

- *X pripada klasi NP*
- *Y je polinomijalno svodljiv na X, gde je Y neki NP-kompletan problem.*

Dakle, ako znamo da je neki problem NP-kompletan, onda na osnovu prethodne teoreme možemo da pokažemo da su i svi NP-problemi na koje ga je moguće svesti takodje NP-kompletni. Dugo se tragalo za pogodnim NP-kompletnim problemom za koji bi moglo da se pronadje polinomijalno rešenje, što bi značilo da važi $P = NP$. Zato je bilo važno otkriti više raznorodnih NP-kompletnih problema. Postavljalo se pitanje da li uopšte postoji ijedan NP-kompletan problem (korišćenjem prethodne teoreme može da se utvrdi da je neki problem NP-kompletan samo ako se za neki drugi problem već zna da je NP-kompletan). Steven Kuk je 1971. godine dokazao (neposredno, koristeći formalizam Tjuringove mašine) da je problem SAT problem NP-kompletan. U godinama koje su sledile za mnoge probleme je utvrdjeno da su takodje NP-kompletni (najčešće svodjenjem problema SAT na njih), ali ni za jedan od njih nije pokazano da pripada klasi P, pa se još uvek ne zna da li važi $P = NP$ (mada većina istraživača veruje da ne važi).

$PSPACE$ je klasa problema koji mogu biti rešeni korišćenjem memorijskog prostora koji je polinomijalna funkcija ulaza. Važi $NP \subseteq PSPACE$, ali se ne zna da li važi $NP = PSPACE$ (rasprostranjeno je uverenje da ne važi).

Tokom poslednje decenije, pored napora da se dokaže da ne važi $P = NP$, radi se i na ispitivanju raspodela najtežih problema u pojedinim klasama NP-kompletnih problema. Dosta se radi i na primeni novih pristupa u efikasnijem rešavanju nekih instanci NP-kompletnih problema.

najznačajnijih otvorenih matematičkih i informatičkih problema. Problem da li su klase P i NP jednake je prvi na toj listi.

Pitanja i zadaci za vežbu

Pitanje 2. *ako za vreme izvršavanja $T(n)$ algoritma A (gde n odredjuje ulaznu vrednost za algoritam) važi $T(n) = T(n-1) + n/2$ i $T(1) = 1$, odrediti složenost algoritma A.*

Rešenje:

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

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

pa je algoritam A kvadratne složenosti.

Pitanje 3. *Ako za vreme izvršavanja $T(n)$ algoritma A (gde n odredjuje ulaznu vrednost za algoritam) važi $T(n+2) = 4T(n+1) - 4T(n)$ (za $n \geq 1$) i $T(1) = 6, T(2) = 20$, odrediti složenost algoritma A.*

Rešenje: Karakteristična jednačina za navedenu homogenu rekurentntu vezu je

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

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

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

tj.

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

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

$$\alpha \cdot 2 + \beta \cdot 2 = 6$$

$$\alpha \cdot 4 + \beta \cdot 8 = 20$$

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

Pitanje 4. *Problem P ima parametar n (n je prirodan broj) i rešava se primenom algoritama A i B. Algoritam A rešava problem za vrednost n ($n > 1$). primenom algoritma B za vrednost $n - 1$, pri čemu se za svodjenje problema troši n vremenskih jedinica. Algoritam B rešava problem za vrednost n ($n > 1$). primenom algoritma A za vrednost $n - 1$, pri čemu se za svodjenje problema troši n vremenskih jedinica. Problem za $n = 1$, algoritam A rešava trivijalno za jednu vremensku jedinicu, a algoritam B za dve vremenske jedinice. Izračunati vreme izvršavanja algoritma A za ulaznu vrednost n.*

Rešenje:

Označimo sa a_n vreme izvršavanja algoritma A, a sa b_n vreme izvršavanja algoritma B za ulaznu vrednost n. Na osnovu uslova zadatka je:

$a_1 = 1$

$b_1 = 2$

$a_n = b_{n-1} + n \quad (n > 1)$

$b_n = a_{n-1} + n \quad (n > 1)$

Kako je $b_{n-1} = a_{n-2} + n - 1$, zaključujemo da je

$a_n = a_{n-2} + n - 1 + n$

Primenom matematičke indukcije dokažimo da za neparne vrednosti n važi $a_n = \frac{n(n+1)}{2}$.

Tvrdjenje važi za $n = 1$ jer je $a_1 = 1 = \frac{1 \cdot 2}{2}$.

Pretpostavimo da je tvrdjenje tačno za neki neparan broj n i dokažimo da važi i za sledeći neparan broj - $n + 2$:

$a_{n+2} = b_{n+1} + n + 2 = a_n + n + 1 + n + 2$

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

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

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

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

$a_{n+2} = \frac{(n+2)(n+3)}{2}$

Dakle, na osnovu principa matematičke indukcije tvrdjenje važi za sve neparne brojeve.

Dokažimo da za parne vrednosti n važi $a_n = \frac{n(n+1)}{2} + 1$.

Za $n = 2$ tvrdjenje je tačno: $a_2 = b_1 + 2 = 2 + 2 = \frac{2 \cdot 3}{2} + 1$.

Pretpostavimo da je tvrdjenje tačno za neki paran broj n i dokažimo da je tačno i za $n + 2$, tj. za sledeći paran broj:

$a_{n+2} = b_{n+1} + n + 2 = a_n + n + 1 + n + 2$

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

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

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

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

$a_{n+2} = \frac{(n+2)(n+3)}{2} + 1$

Dakle, tvrdjenje je tačno za sve parne brojeve.

Konačno, važi:

$$a_n = \begin{cases} \frac{n(n+1)}{2} + 1 & \text{za } n \text{ parno} \\ \frac{n(n+1)}{2} & \text{za } n \text{ neparno} \end{cases}$$

Glava 3

Ispitivanje ispravnosti 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. 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. kako bi merio klimu uzrokovan činjenicom da je deo softvera koristio metričke, a deo softvera engleske jedinice su katastrofe koje su godinama opštepoznate. Međutim, fatalne sofverske greške se i dalje neprestano javljaju i one koje koštaju svetsku ekonomiju milijarde dolara. Evo nekih od najzanimljivijih:

- 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).

- 14. Septembra 2004. godine, više od četiristo aviona u blizini aerodroma u Los Anđelesu je 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 ranije otkrivena, 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.

- Pad satelita *Kriosat (eng. Cryosat)* 2005. koštao je Evropsku Uniju oko 135 milona 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 strane umesto sa leve kako je pravilno.

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

3.1 Osnovni pristupi verifikaciji

Postupak pokazivanja da je program ispravan naziva se *verifikacija*. 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 *postulsova* 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 korektnosti u fazi izvršavanja programa, najčešće putem testiranja i *statička verifikacija* koja podrazumeva analizu programskog koda korišćenjem formalnih metoda i matematičkog aparata.

U okviru ispitivanja ispravnosti, 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.

Proces verifikacije može biti neformalan i formalan (kada se koristi precizno definisana semantika programskog jezika i kada je precizno definisan logički okvir u kome se dokazi korektnosti izvode). U praksi se formalni dokaz ispravnosti programa retko izvodi i to često dovodi do neispravnog softvera i mnogih problema.

Veliki izazov za verifikaciju predstavlja činjenica da semantika uobičajenih tipova podataka i operacija u programima značajno razlikuje od uobičajene semantike matematičkih operacija (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 koji 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 se 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 stalno vršene.

3.2 Testiranje

Izuzetno značajan pristup obezbeđivanju višeg stepena pouzdanosti programa je testiranje, najznačajnija vrsta dinamičkog ispitivanja ispravnosti programa.

Neka tvrđenja o programu je moguće testirati, dok neka nije. Na primer, tvrdenje ,,program ima prosečno vreme izvršavanja 0.5 sekundi“ je (u principu) proverivo testovima kao i tvrđenje ,,prosečno vreme između dva pada programa je najmanje 8 sati sa verovatnocom 95%“. 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, trebalo bi 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 moguć. 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 nano sekundu (što je prilično optimistično), iscrpno testiranje bi uzelo približno $1.8 \cdot 10^{10}$ sekundi što je oko 570 godina. Dakle, testiranjem nije moguće dokazati korektnost netrivijalnih programa. S druge strane, testiranjem je moguće dokazati da program nije korektan tj. pronaći greške u programima.

S obzirom da iscrpno testiranje nije moguće, 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 brojeva 0, 1, -1, najmanji nega-

tivan broj, najveći pozitivan broj i slično.

Ukoliko se program sastoji od više celina (funkcija, modula), poželjno je nezavisno testirati svaki pojedinačni modul. Ova tehnika se naziva *testiranje zasebnih jedinica* (eng. unit testing). U današnje vreme postoje specijalizovani softverski alati i biblioteke koji omogućavaju jednostavnije kreiranje i održavanje ovakvih testova.

3.3 Funkcionalni programi

U principu, najjednostavnije je dokazivanje korektnosti rekurzivno definisanih funkcija koje ne koriste elemente imperativnog programiranja (dodele vrednosti pomoćnim promenljivim, petlje i slično). Glavni razlog za ovo je činjenica da se ovakve funkcije mogu jednostavno modelovati klasičnim matematičkim funkcijama. Zbog pretpostavke da nema eksplicitne dodele vrednosti promenljivima, nema potrebe za razmatranjem tekućeg stanja programa (okarakterisanog tekućim vrednostima promenljivih) i čini funkcije u programima veoma nalik matematičkim — prilikom poziva, za isti ulazni argument uvek se dobija ista vrednost, bez obzira na kontekst u kome je poziv nastupio. Dokazivanje korektnosti ovakvih programa teče nekim oblikom matematičke indukcije.

Primer 3.1. *Razmotrimo primer funkcije koja vrši množenje svođenjem na sabiranje. Dokažimo korektnost ove funkcije pod pretpostavkom da je* $x \geq 0$.

```
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 3.2. *Dokažimo korektnost algoritma brzog stepenovanja:*

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

S obzirom da je u ovom slučaju funkcija definisana opštom rekurzijom (za razliku od prethodnog skučaja kada je rekurzija bila primitivna) i dokaz mora da prati drugačiju shemu indukcije. Moguće je pokazati da za ovu funkciju važi naredna shema indukcije: ,,Da bi se pokazalo da vrednost $stepen(x, n)$ *zadovoljava neko svojstvo, može se pretpostaviti da za* $n \neq 0$ *i* n *parno vrednost* $stepen(x \cdot x, n/2)$ *zadovoljava to svojstvo, kao i da za* $n \neq 0$ *i* n *neparno vrednost* $stepen(x, n-1)$ *zadovoljava to svojstvo, pa tvrđenje 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 $stepen(x, n) = x^n$ *(za nenegativnu vrednost* n*).*

Slučaj: $n = 0$**.** *Tada je* $stepen(x, n) = stepen(x, 0) = 1 = x^0$.

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

- **Slučaj:** n **je paran.** *Tada je* $stepen(x, n) = stepen(x \cdot x, n/2)$. *Na osnovu prve induktivne pretpostavke, važi da je* $stepen(x \cdot x, n/2) = (x \cdot x)^{n/2}$. *Dalje, elementarnim aritmetičkim transformacijama sledi da je* $stepen(x, n) = (x \cdot x)^{n/2} = (x^2)^{n/2} = x^n$.
- **Slučaj:** n **je neparan.** *Tada je* $stepen(x, n) = x \cdot stepen(x, n-1)$. *Na osnovu druge induktivne pretpostavke važi da je* $stepen(x, n-1) = x^{n-1}$. *Dalje, elementarnim aritmetičkim transformacijama sledi da je* $stepen(x, n) = x \cdot x^{n-1} = x^n$.

3.4 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 izmedju stanja i dokazivanje korektnosti

zahteva dokazivanje da će program na kraju stići u neko stanje u kome su zadovoljeni uslovi zadati specifikacijom. Dodatnu otežavajuću okolnost čine bočni efekti 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, te je dokaz korektnosti složenog programa teže razložiti na elementarne dokaze korektnosti pojedinih funkcija.

Kao najkompleksniji programski konstrukt, petlje predstavljaju najveći izazov i 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 (npr. $x \cdot 0 = 0$ 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 3.3. *Razmotrimo program koji vrši množenje dva broja 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 $z = n \cdot y$. Ako je x nenegativan ceo broj (što je u ovom slučaju bitno za korektnost algoritma), 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. 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 3.4. *Razmotrimo program koji vrši deljenje uzastopnim oduzimanjem.*

```
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 3.5. *Razmotrimo program koji vrši pronalaženje minimuma niza brojeva.*

```
m = a[0];
for (j = 1; j < n; j++)
    if (a[j] < m)
         m = a[j];
```

Ovaj program ima smisla samo za neprazne nizove, pri čemu se još podrazumeva da je n dužina niza a. Invarijanta petlje je da promenljiva m sadrži minimum dela niza između pozicija 0 i j (tj. minimum elemenata od $a[0]$ do $a[j-1]$). Nakon završetka petlje, važi da je $j = n$, i pošto je n dužina niza iz invarijante sledi korektnost programa.

3.4.1 Formalni dokazi i Horova logika

Sva dosadašnja razmatranja o korektnosti programa vršena su neformalno, tj. nije postojao precizno opisan formalni sistem u kome bi se vršilo dokazivanje korektnosti imperativnih programa. Jedan od najznačajnijih formalnih sistema ovog tipa opisao je Toni Hor (Tony Hoare) u svom radu ,,An Axiomatic Basis for Computer Programming“. Istraživači koji su dali doprinos na ovom polju

pre i nakon Hora su, između ostalih, Robert Flojd (Robert Floyd), Džon Fon Nojman (John Von Neumann), Herman Goldštajn (Herman Goldstine), Alen Tjuring (Alan Turing) i Džon Makarti (John McCarthy).

Formalni dokazi (u jasno preciziranom logičkom okviru) su interesantni 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. Takođe, kada se program i dokaz njegove korektnosti istovremeno razvijaju, programer bolje razume sam program i proces programiranja uopšte. Informatičari su zainteresovani za formalne dokaze jer metodologija dokaza utiče na razmatranje preciznosti, konzistentnosti i kompletnosti specifikacije, na jasnoću implementcije i konzistentnost implementacije i specifikacije. Zahvaljujuci 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` programski kôd, a $\{\varphi\}$ i $\{\psi\}$ su logičke formule koje opisuju veze između promenljivih koje se javljaju u programu. Trojku $(\varphi, \texttt{P}, \psi)$ nazivamo *Horova trojka*. Interpretacija trojke je sledeća: „Ako izvršenje programa P počinje sa vrednostima ulaznih promenljivih (kažemo i ‚u stanju') koje zadovoljavaju uslov $\{\varphi\}$ i ako se program P završi u konačnom broju koraka, tada vrednosti programskih promenljivih (‚stanje') zadovoljavaju uslov $\{\psi\}$". Uslov $\{\varphi\}$ naziva se *preduslov* za algoritam (program, iskaz) P, a uslov $\{\psi\}$ naziva se *postuslov (pauslov, posleuslov)* za algoritam (program, naredbu) P.

Na primer, trojka $\{x = 1\}\texttt{y:=x}\{y = 1\}$[1], 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. Sa druge strane, trojka $\{x = 1\}\texttt{y:=x}\{y = 2\}$ 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 koje važe za ulazne promenljive, dok postuslov 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 $\{\top\}\texttt{P}\{z = x \cdot y\}$. Ako se zadovoljimo time da program može da množi samo nenegativne brojeve, specifikacija se može oslabiti u $\{x \geq 0 \wedge y \geq 0\}\texttt{P}\{z = x \cdot y\}$.

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 3.1). Za svaku sintaksnu konstrukciju programskog jezika koji se razmatra formiraju se aksiome i/ili pravila izvođenja koji aksiomatski daju opis semantike odgovarajućeg konstrukta programskog jezika.[2]

[1] U ovom poglavlju, umesto C-ovske, biće korišćena sintaksa slična sintaksi korišćenoj u

Pravilo posledice (Cons)

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

Pravilo kompozicije (Comp)

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

Aksioma dodele (assAx)

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

Pravilo grananja (ifAx)

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

Pravilo petlje (whileAx)

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

Slika 3.1: Horov formalni sistem

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

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

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

originalnom Horovom radu.

[2]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.).

slobodna 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 bočne efekte do samog efekta dodele (tj. izmene vrednosti promenljive sa leve strane dodele) koji je implicitno i opisan navedenom aksiomom.

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 granaja 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 `grana` zna se da je uslov grananja ispunjen, dok, ukoliko se izvršava `else` grana, zna se da 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.

Primer 3.6. *Dokažimo u Horovom sistemu da je klasičan 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 3.7. *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 3.1.

(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; }` $\{n \geq x \wedge n \leq x \wedge z = n * y\}$
(whileAx)

(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 3.1: Primer dokaza u Horovoj logici

3.4.2 Dokazivanje zaustavljanja programa

S obzirom na da je halting problem neodlučiv, zaustavljanje svakog programa mora se pokazivati na zaseban način (tj. ne postoji opšti postupak koji bi se primenio na sve programe). U programima kakvi su do sada razmatrani jedine naredbe koje mogu da dovedu do nezaustavljanja su petlje, tako da je potrebno dokazati zaustavljanje svake pojedinačne petlje. Ovo se obično radi tako što se definiše dobro zasnovana relacija[3] 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 stanja u skup prirodnih brojeva i pokaže se da se svako susedno stanje preslikava u manji prirodan

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

broj. Pošto je relacija $>$ na skupu prirodnih brojeva dobro zasnovana, dobro će zasnovana biti i ovako definisana relacija na skupu stanja.

Primer 3.8. *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), tako da u jednom trenutku mora da dostigne vrednost* 0.

Glava 4

Fundamentalni algoritmi

U ovoj glavi biće predstavljeni neki od osnovnih algoritama za pretraživanje, sortiranje i izračunavanje.

4.1 Pretraživanje

Pod pretraživanjem za dati niz elemenata podrazumevamo određivanje indeksa elementa niza koji je jednak datoj vrednosti ili ispunjava neko drugo zadato svojstvo (npr. najveći je element niza).

4.1.1 Linearno pretraživanje

Linearno (ili *sekvencijalno*) pretraživanje niza je pretraživanje zasnovano na ispitivanju redom svih elemenata niza ili ispitivanju redom elemenata niza sve dok se ne naiđe na traženi element (zadatu vrednost ili element koji ima neko specifično svojstvo, na primer — maksimum niza). Linearno pretraživanje je vremenske linearne složenosti po dužini niza koji se pretražuje. Ukoliko se u nizu od n elemenata traži element koji je jednak zadatoj vrednosti, u prosečnom slučaju (ako su elementi niza slučajno raspoređeni), ispituje se $n/2$, u najboljem 1, a u najgorem n elemenata niza.

Primer 4.1. *Naredna funkcija vraća indeks prvog pojavljivanja zadatog celog broja* `x` *u zadatom nizu* `a` *dužine* `n` *ili vrednost -1, ako se taj 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 $O(\texttt{n})$ a njena korektnost se dokazuje jednostavno (pod pretpostavkom da nenegativan broj `n` *predstavlja broj elemenata niza* `a`*).*

Primer 4.2. *Linearna pretraga može biti realizovana i rekurzivno. U narednom kodu se poziv* `linearna_pretraga(a, i, n, x)` *nalazi indeks prvog pojavljivanja elementa* `x` *u nizu* `a[i, n-1]`*. Kako bi se izvršila pretraga celog niza, potrebno je izvršiti poziv* `linearna_pretraga(a, 0, n, x)`*, pri čemu je* `n` *dužina niza* `a`*.*

```
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);
}
```

Ukoliko se zadovoljimo pronalaženjem poslednjeg pojavljivanja elementa `x`*, kôd se može dodatno uprostiti.*

```
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, x);
}
```

U oba slučaja rekurzivni pozivi su repno rekurzivni. Eliminacijom repne rekurzije u prvom slučaju se upravo dobija prikazana iterativna implementacija. Eliminacijom repne rekurzije u drugom slučaju se dobija:

```
int linearna_pretraga(int a[], int n, int x) {
  int i;
  for (i = n; i > 0; i--)
     if (a[i - 1] == x)
         return i - 1;
  return -1;
}
```

Primer 4.3. *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 je linearna po dužini niske `s` *a njena korektnost se dokazuje jednostavno.*

Primer 4.4. *Naredna funkcija vraća indeks najvećeg elementa među prvih* `n` *elemenata niza* `a` *(pri čemu je* `n` *veće 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 $O(\mathtt{n})$ *a njena korektnost se dokazuje jednostavno.*

4.1.2 Binarno pretraživanje

Binarno pretraživanje (ne nužno niza) je pronalaženje zadate vrednosti u zadatom skupu objekata koje pretpostavlja da su objekti zadatog skupa sortirani i u svakom koraku, sve dok se ne pronađe tražena vrednost, taj skup se deli na dva dela i pretraga se nastavlja samo u jednom delu — odbacuje se deo koji sigurno ne sadrži traženu vrednost. U diskretnom slučaju, ako skup koji se pretražuje ima n (konačno mnogo) elemenata, binarno pretraživanje je vremenski logaritamske složenosti — $O(\log n)$. Pretraživanje se u tom, diskretnom slučaju, vrši na sledeći način: pronalazi se srednji (po postojećem uređenju) element skupa (obično niza), proverava se da li je on jednak zadatoj vrednosti i ako jeste vraća se njegov indeks, a ako nije pretraživanje se nastavlja nad skupom svih manjih (ako je srednji element veći od zadate vrednosti) ili svih većih elemenata (ako je srednji element manji od zadate vrednosti). Binarno pretraživanje je primer *podeli-i-vladaj* (*divide and conquer*) algoritama.

Primer 4.5. *Binarno pretraživanje se može 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. *Broj pitanja potrebnih za određivanje intervala pretrage je* $O(\log k)$, *gde je* k *širina polaznog intervala.*

Primer 4.6. *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 1 *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 i ukupna složenost pogađanja je ponovo* $O(\log k)$.

Binarno pretraživanje je daleko efikasnije nego linearno, ali zahteva da su podaci koji se pretražuju uređeni. Na primer, ovo je jedan od glavnih razloga zašto se reči u rečnicima, enciklopedijama, telefonskim imenicima i slično sortiraju. Ove knjige se obično pretražuju postupkom koji odgovara varijantama linearne pretrage[1]. Odnos složenosti, postaje još očigledniji ukoliko se zamisli koliko bi komplikovano bilo sekvencijalno pretraživanje reči u nesortiranom rečniku.

Binarno pretraživanje se može implementirati iterativno ili rekurzivno.

Primer 4.7. *Naredna funkcija daje rekuriznu implementaciju binarnog pretraživanja. Poziv* `binarna_pretraga(a, l, d, x)` *vraća indeks elementa niza* `a` *između* `l` *i* `d` *(uključujući i njih) koji je jednak zadatoj vrednosti* `x` *ako takav postoji i* `-1` *inače. Dakle, ukoliko se želi pretraga celog niza, funkciju treba pozvati sa* `binarna_pretraga(a, 0, n-1, v)`.

[1] 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 (npr. 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).

```
int binarna_pretraga(int a[], int l, int d, int x) {
    int s;

    if (l > d)
        return -1;

    s = d + (d - l)/2;
    if (x > a[s])
       return binarna_pretraga(a, s+1, d, x);
    else if (x < a[s])
       return binarna_pretraga(a, l, s-1, x);
    else /* if (x == a[s]) */
       return s;
}
```

Primetimo da se za nalaženje središta, umesto izraza `d + (d - l)/2` *može koristiti i kraći izraz* `(l + d)/2`*. Ipak, upotreba prvog izraza je preporučena kako bi se smanjila mogućnost nastanka prekoračenja. Ovo jedan u nizu primera gde izrazi koji su matematički ekvivalentni, pokazuju različita svojstva u aritmetici fiksirane širine.*

Složenost ove funkcije je $O(\log \mathtt{n})$ *a njena korektnost se dokazuje jednostavno, pri čemu se pretpostavlja da je niz* `a` *sortiran.*

Primer 4.8. *Oba rekurzivna poziva u prethodnoj funkciji su bila 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 = d + (d - l)/2;
     if (x > a[s])
         l = s + 1;
     else if (x < a[s])
         d = s - 1;
     else /* if (x == a[s]) */
         return s;
   }
   return -1;
}
```

Sistemska implementacija binarnog pretraživanja

U standardnog 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. Prototip funkcije `bsearch` je:

```
void *bsearch(const void *key, const void *base,
              size_t num, size_t size,
              int (*compare)(const void *, const void *));
```

Argumenti funkcije imaju sledeću ulogu:

`key` je pokazivač na vrednost koja se traži; on je tipa `void*` kako bi mogao da prihti pokazivač na bilo koji konkretan tip;

`base` je pokazivač na početak niza koji se pretražuje; on je tipa `void*` kako bi mogao da prihti pokazivač na bilo koji konkretan tip;

`num` je broj elemenata niza koji će biti ispitani;

`size` je veličina jednog elementa u bajtovima; ona je potrebna kako bi funkcija mogla da prelazi sa jednog na sledeći element niza;

`compare` je pokazivač na funkciju (često definisanu od strane korisnika) koja vrši poređenje dve vrednosti zadatog tipa; 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 vrednost na koju ukazuje prvi argument manja od vrednosti na koju ukazuje drugi argument;
- 0 ako su vrednosti na koju ukazuju prvi i drugi argument jednake;
- > 0 ako je vrednost na koju ukazuje prvi argument veća od vrednosti na koju ukazuje drugi argument;

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).

Opštost funkcije `bsearch` ima i svoju cenu: zbog neophodnog kastovanja pokazivača nema provere tipova a za svaku proveru odnosa dve vrednosti poziva se druga funkcija (`compare`) što može da troši mnogo vremena i memorijskog prostora.

Primer 4.9. *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), 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 je mogla da bude implementirana i kao

```
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 prilikom poređenja.

Određivanje nula funkcije

Jedan od numeričkih metoda za određivanje nula neprekidne realne funkcije zasniva se na metodu polovljenja intervala koji odgovara binarnoj pretrazi nad neprikidnim domenom. Pretpostavka metoda je da je funkcija neprekidna i da su poznate vrednosti a i b u kojima funkcija ima različit znak. U svakom koraku se tekući interval 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 se dužina intervala smanjuje dvostruko i ako je dužina početnog intervala jednaka d, onda nakon n interacija 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.

Primer 4.10. *Naredna funkcija pronalazi, za zadatu tačnost, nulu zadate funkcije* `f`*, na intervalu* $[$`l`$,$`d`$]$*, pretpostavljajući da su vrednosti funkcije* `f` *različitog znaka u* `l` *i* `d`*.*

```
#include <math.h>

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;
    }
  }
}
```

4.2 Sortiranje

Sortiranje je jedan od fundamentalnih zadataka u računarstvu. Sortiranje podrazumeva uređivanje niza u odnosu na neko linearno uređenje (npr. uređenje niza brojeva po veličini — rastuće ili opadajuće, uređivanje niza niski leksikografski ili po dužini, uređivanje niza struktura na osnovu vrednosti nekog polja i slično). Mnogi zadaci nad nizovima se mogu jednostavnije odraditi u slučaju da je niz sortiran (npr. pretraživanje se može vršiti binarnom pretragom).

Postoji više različitih algoritama za sortiranje nizova. Neki algoritmi su jednostavni i intuitivni, dok su neki kompleksniji, ali izuzetno efikasni. Najčešće korišćeni algoritmi za sortiranje su:

- Bubble sort
- Selection sort
- Insertion sort
- Shell sort
- Merge sort
- Quick sort
- Heap sort

Neki od algoritama za sortiranje rade u mestu (eng. in-place), tj. sortiraju zadate elemente bez korišćenja dodatnog niza. Drugi algoritmi zahtevaju korišćenje pomoćnog niza ili nekih drugih struktura podataka.

Napomenimo da su prilikom sortiranja nizova koji sadrže podatke netrivijalnih tipova najkompleksnije operacije operacija poređenja dva elementa niza

i operacija razmene dva elemente niza. Zbog toga se prilikom izračunavanja složenosti algoritama obično u obzir uzimaju samo ove operacije.

Algoritmi za sortiranje se obično mogu podeliti u dve grupe, na osnovu njihove složenosti. Jednostavniji, sporiji algoritmi sortiranja imaju složenost najgoreg slučaja $O(n^2)$ i u tu grupu spadaju *bubble*, *insertion*, i *selection*. *Shell sort* je algoritam koji spada 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 $O(n^2)$, međutim, pošto je složenost prosečnog slučaja $O(n \log n)$ i pošto u praksi pokazuje dobre rezultate, ovaj se algoritam ubraja u grupu veoma brzih algoritama.

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 će biti prikazani na kanonskom primeru sortiranja niza celih brojeva u neopadajućem poretku. Započnimo izlaganje funkcijom koja proverava da li je niz već sortiran.

```
int sortiran(int a[], int n) {
  int i;
  for (i = 0; i < n - 1; i++)
      if (a[i] > a[i + 1])
         return 0;
  return 1;
}
```

U većini implementacija, osnovi korak će biti razmena elemenata niza i pretpostavićemo da je na raspolaganju 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 brojeva je

```
void sort(int a[], int n);
```

Preduslov funkcije je da nenegativna promenljiva `n` sadrži dimenziju niza `a`[2], dok je postuslov da su elementi niza[3] nakon primene funkcije sortirani kao i da se (mulit)skup elemenata niza nije promenio (ova invarijanta je trivijalno ispunjena kod svih algoritama zasnovanih na razmenama elemenata niza).

4.2.1 *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 se

[2] Preciznije, preduslov je da je `n` manje ili jednako od alociranog broja 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.

[3] Preciznije, elementi između pozicija `0` i `n-1` su sortirani, što je zaista ceo niz ukoliko `n` sadrži dimenziju niza.

ponavljaju sve dok se ne napravi prolaz u kome nije bilo razmena, što znači da je niz sortiran.

Primer 4.11. *Prikažimo rad algoritma na primeru sortiranja niza* (6 1 4 3 9)*:*

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 $4 > 3$
(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 *bubble sort* algoritmom 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 pre provere uslova izlaska iz spoljašnje petlje, onda je niz `a` sortiran (tj. njegov početni deo dužine `n`). Zaista, kada se spoljašnja petlja završi, vrednost promenljive `bilo_razmena` je 0, te prethodna implikacija obezbeđuje korektnost. Invarijanta unutrašnje petlje koja obezbeđuje pomenuti uslov je da ako promenljiva `bilo_razmena` ima vrednost 0, onda je deo niza `a[0, i]` sortiran. Zaista, po završetku unutrašnje petlje `i` ima vrednost `n-1`, te je pomenuti uslov obezbeđen. Pošto je algoritam zasnovan na razmenama, invarijanta algoritma je i da se (multi)skup elemenata niza ne menja tokom njegovog izvršavanja.

Svojstvo algoritma koje obezbeđuje zaustavljanje je da se nakon svake iteracije spoljašnje petlje sledeći najveći elemenat koji nije bio na svojoj poziciji

dolazi na nju. *Bubble sort* je na osnovu ovog svojstva i dobio ime (jer veliki elementi kao mehurići ,,isplivavaju" ka kraju niza). Ovo sa jedne strane obezbeđuje zaustavljanje algoritma, dok se sa druge strane može iskoristiti za optimizaciju. Ovom optimizacijom se može smanjiti broj poređenja, ali ne i broj razmena.

```
void bubblesort(int a[], int n) {
  do {
    int i;
    for (i = 0; i < n - 1; i++)
       if (a[i] > a[i + 1])
          razmeni(a, i, i+1);
    n--;
  } while (n > 1);
}
```

Isti algoritam, implementiran na stilski malo drugačiji način (koji se može ponekad sresti u literaturi).

```
void bubblesort(int a[], int n) {
  int i, j;
  for (i = n - 1; i > 0; i--)
    for (j = 0; j < i; j++)
       if (a[j] > a[j + 1])
          razmeni(a, j, j+1);
}
```

Optimizacija može otići i korak dalje, ukoliko se dužina ne smanjuje samo za 1, već ukoliko se posmatra koliko je zaista elemenata na kraju niza već postavljeno na svoje mesto (ovo se može odrediti na osnovu mesta gde se dogodila poslednja zamena).

```
int bubblesort(int a[], int n) {
  do {
    int nn = 1, i;
    for (i = 0; i < n - 1; i++)
       if (a[i] > a[i + 1]) {
          razmeni(a, i, i+1);
          nn = i + 1;
       }
    n = nn;
  } while (n > 1);
}
```

U ovom slučaju, uslov koji važi pre provere uslova spoljašnje petlje je da su elementi od pozicije `n-1` do kraja niza sortirani. Ovo obezbeđuje invarijanta unutrašnje petlje koja obezbeđuje da su svi elementi `a[nn-1, i]` sortirani.

Što se efikasnosti tiče, najgori slučaj nastupa kada su elementi početnog niza sortirani u opadajućem poretku. U tom slučaju vrši se n prolaska kroz niz, a u svakom prolasku se pravi n poređenja i n razmena, što ukupno daje $O(n^2)$ poređenja i $O(n^2)$ razmena. Primetimo da se optimizacijom broj poređenja

smanjuje otprilike dva puta.

Bubble sort algoritam se smatra veoma lošim algoritmom. Neki autori čak zagovaraju tezu da bi ga trebalo potpuno izbaciti iz nastave računarstva. U šali se čak navodi kako bi jedini algoritam gori od njega bio algoritam koji permutuje algoritam na slučajan način sve dok niz slučajno ne postane sortiran.

4.2.2 *Selection sort*

Selection sort je algoritam koji se može u jednoj rečenici opisati sa: „Ako niz ima više od jednog elementa, dovedi najmanji element na prvu poziciju i zatim rekurzivno sortiraj rep (elemente iza prve pozicije)". U iterativnoj implementaciji, niz se sortira tako što se u svakoj iteraciji na svoju poziciju dovodi sledeći po redu element niza, tj. u i-toj iteraciji se i-ti po redu 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 može da prestane sa radom u slučaju kada se pretposlednji po veličini element dovede na pretposlednju poziciju u nizu.

Primer 4.12. *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` se može 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;
}
```

U tom slučaju, *selection sort* algoritam izgleda ovako:

```
int 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 su svi oni manji od svih elemenata niza `a[i+1, n-1]` (eventualno je, poslednji, najveći element prvog dela 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`. Tako da je ukupan broj poređenja $\sum_{i=0}^{n-1} n-i = \sum_{i=0}^{n-1} i = \frac{n\cdot(n-1)}{2}$.

Ukoliko se ne koriste pomoćne funkcije, algoritam se može implementirati na sledeći način.

```
int 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.

```
int selectionsort(int a[], int n) {
  int i, j;
  for (i = 0; i < n; i++)
     for (j = i + 1; j < n; j++)
         if (a[i] < a[j])
            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. Naravno, iz ovog razloga bi ovaj način implementacije trebalo izbegavati.

Rekruzivna implementacija je donekle jednostavnija u slučaju da se umesto dovođenja najmanjeg elementa na početak vrši dovođenje najvećeg elementa na kraj.

```
int poz_max(int a[], int n) {
  if (n == 1)
     return 0;
  else {
     int m = poz_max(a, n-1);
     return a[m] > a[n-1] ? m : n-1;
  }
}

void selectionsort(int a[], int n) {
  if (n > 1) {
     razmeni(a, n-1, poz_max(a, n));
     selectionsort(a, n-1);
  }
}
```

Naravno, moguća je i originalna varijanta, uz slanje dodatnog indeksa kroz rekurzivne pozive.

```
int poz_min(int a[], int n, int i) {
  if (i == n-1)
     return n-1;
  else {
     int m = poz_min(a, n, i+1);
     return a[m] < a[i] ? m : i;
  }
}

void selectionsort(int a[], int n, int i) {
  if (i < n - 1) {
     razmeni(a, i, poz_min(a, n, i));
     selectionsort(a, n, i+1);
  }
}
```

Početni poziv je u tom slučaju `selectionsort(a, n, 0)`.

Naglasimo da se korišćenjem naprednijih struktura podataka, ideje *selection sort* algoritma mogu iskoristiti da se dobije algoritam složenosti $O(n \log n)$ (tzv. *heap sort* algoritam koji koristi strukturu podataka poznatu kao hip (eng. heap)).

4.2.3 *Insertion sort*

Insertion sort algoritam sortira niz tako što uzima jedan po jedan element niza i umeće ga na odgovarajuće mesto u sotrirani niz koji čuva krajnji rezultat. Konceptualno gledano, postoje dva niza — polazni niz iz kojeg se uklanjaju elementi i niz koji čuva rezultat i u koji se dodaju elementi. Međutim, obično implementacije koriste memorijski prostor polaznog niza za obe uloge — početni deo niza predstavlja rezultujući niz, dok krajnji deo predstavlja preostali deo

polaznog niza.

Primer 4.13. *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.
Masnim slovima su prikazani elementi umetnuti na svoju poziciju.

Dakle, *insertion sort* se može formulisati na sledeći način: „Ako niz ima više od jednog elementa, sortiraj rekurzivno sve elemente ispred poslednjeg, a zatim umetni poslednji u već sortirani prefiks." Ovim se dobija sledeća (rekurzivna) implementacija.

```
void insertionsort(int a[], int n) {
  if (n > 1) {
    insertionsort(a, n-1);
    umetni(a, n-1);
  }
}
```

Kada se eliminiše rekurzija, dolazi se 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. Kako bi se obezbedila korektnost, preduslov funkcije `umetni` je 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 od mogućnosti 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);
}
```

Prikažimo još i odgovarajuću rekurzivnu implementaciju.

```
void umetni(int a[], int j) {
  if (j > 0 && a[j] < a[j-1]) {
       razmeni(a, j, j-1);
       umetni(a, j-1);
  }
}
```

Funkcija `umetni` se poziva $n-1$ put i to za vrednosti i od 1 do n, dok u najgorem slučaju (obrnuto sortiranog niza) ona izvršava i razmena i i poređenja. Zbog toga je ukupan broj razmena, kao i broj poređenja $O(n^2)$ ($\sum_{i=1}^{n} i = \frac{n\cdot(n+1)}{2}$).

Efikasnija implementacija se može dobiti ukoliko se ne koriste razmene, već se element koji nije na svom mestu zapamti, zatim se pronađe pozicija na koju treba da se umetne, svi elementi od te pozicije se pomere za jedno mesto u desno da bi se na kraju zampćeni element upisao 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;
}
```

S obzirom da se prilikom razmena koriste tri dodele, ovim se broj dodela smanjuje 3 puta (broj poređenja ostaje nepromenjen).

Prikažimo i verziju `insertionsort` funkcije u kojoj je kod pomoćne funkcije 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 se može 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 značajno smanjiti. Zbog toga, nećemo prikazati implementaciju ovakvog rešenja.

Naglasimo da se korišćenjem naprednijih struktura podataka, ideje *insertion sort* algoritma mogu iskoristiti da se dobije algoritam složenosti $O(n \log n)$ (tzv. *tree sort* algoritam koji koristi binarna stabla).

4.2.4 *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*[4] 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

[4]Nazvan po D.L.Šelu koji ga je prvi opisao 1959. godine.

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 4.14. *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 (npr.* 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

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 da je 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).

4.2.5 *Merge sort*

Dva već sortirana niza se mogu objediniti u treći sortirani niz samo jednim prolaskom kroz nizove (tj. u linearno 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 nizove `a` dimenzije `m` i `b`

dimenzije `n`. Promenljiva `i` čuva tekuću pozicija 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 koga je taj manji element uzet. Prolazak se ne stigne do kraja jednog od nizova. Kada se kraći niz isprazni, eventualni preostali elementi iz dužeg niza se nadovezuju na kraj niza `c`.

Merge sort algoritam deli niz na dve polovine (čija se dužina razlikuje najviše za 1), rekurzivno sortira svaku od njih, i zatim objedinjuje sortirane polovine. Problematično je što je za objedinjavanje neophodno koristiti dodatni niz pomoćni niz. Na kraju se izvršava vraćanje objedinjenog niza iz pomoćnog u polazni. Izlaz iz rekurzije je slučaj jednočlanog niza (slučaj praznog niza ne može da nastupi).

Funkicija `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 se objedinjuju u pomoćni niz `tmp`. Primetimo na ovom mestu korišćenje pokazivačke aritmetike. Adresa početka prvog sortiranog podniza koji se objedinjava je `a+l`, dok je adresa početka drugog `a + s`.

Pomoćni niz se može pre početka sortiranja dinamički alocirati i koristiti kroz rekurzivne pozive.

```
void mergesort(int a[], int n) {
  int* tmp = (int*)malloc(n*sizeof(int));
  mergesort_(a, 0, n-1, tmp);
  free(tmp);
}
```

Provera da li je je alokacija nije uspela ovaj put nije vršena.

Rekurentna jednačina koja opisuje složenost je $T(n) = 2 \cdot T(n/2) + O(n)$ te je složenost $O(n \log n)$.

4.2.6 *Quick sort*

Osnovna ideja kod *selection sort* algoritma je da se jedan element postavi na svoje mesto, a zatim da se isti metod rekurzivno primeni na niz koji je za jedan kraći od polaznog. S obzirom da je pripremna akcija zahtevala $O(n)$ operacija, dobijena je jednačina $T(n) = T(n-1) + O(n)$, čije rešenje je $O(n^2)$. Sa druge strane, kod *merge sort* algoritma sortiranje se svodi na sortiranje dva podniza polaznog niza dvostruko manje dimenzije. S obzirom da korak objedinjavanja dva sortirana niza zahteva $O(n)$ operacija, dobija se jednačina $T(n) = 2T(n/2) + O(n)$, čije je rešenje $O(n \log n)$. 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 tzv. *podeli i vladaj (divide-and-conquer)* algoritama u koje spada i *quick sort*.

Quick sort algoritam pokušava da postigne bolju efikasnost, modifikujući osnovnu ideju *selection sort* algoritma tako što umesto minimuma (ili maksimuma), u svakom koraku na svoje mesto dovede neki element (obično nazivan *pivot*) koji je relativno blizu sredine niza. Međutim, da bi nakon toga, problem mogao biti sveden na sortiranje dva dvostruko manja podniza, potrebno je prilikom dovođenja pivota na svoje mesto grupisati sve elemente manje od njega levo od njega, a sve elemente veće od njega desno od njega. Dakle, ključni korak *quick sort* je tzv. korak *particionisanja* koji nakon izbora nekog pivotrajućeg elementa podrazumeva da se niz organizuje da prvo sadrži elemente manje od pivota, zatim pivotirajući element, i na kraju elemente veće od pivota.

Qsort algoritam algoritam se može implementirati na sledeći način. Poziv `qsort(a, l, d)` sortira deo niza `a[l, d]`.

```
void qsort_(int a[], int l, int d) {
  if (l < d) {
    razmeni(a, l, izbor_pivota(a, l, d));
    int p = particionisanje(a, l, d);
    qsort_(a, l, p - 1);
    qsort_(a, p + 1, d);
  }
}
```

Funkcija `qsort` se onda jednostavno implementira

```
void qsort(int a[], int n) {
  qsort_(a, 0, n-1);
}
```

Funkcija `izbor_pivota` odabire 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 ili jednako `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 ili jednaki elementu `a[p]`, dok su svi elementi niza `a[p+1, d]` veći ili jednaki od elementa `a[p]`.

Kako bi se dobila jednačina $T(n) = 2T(n/2) + O(n)$ i efikasnost $O(n \log n)$, potrebno je da korak particionisanja (tj. funkcija `particionisanje`) bude izvršen 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 dužina dva podniza na koje se problem svodi bilo približno jednaka $n/2$). Međutim, određivanje srednjeg člana u nizu brojeva (što predstavlja idealnu strategiju za funkciju `izbor_pivota`) je problem koji nije značajno jednostavniji od samog sortiranja. S obzirom 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. Napomenimo da se za svaku strategiju izbora pivota (koja ne koristi slučajno izabrane brojeve) može konstruisati niz tako 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 složenosti ($O(n^2)$). Međutim, većina strategija je takva da se u prosečnom slučaju može očekivati relativno ravnomerna raspodela što dovodi do optimalne složenosti ($O(n \log n)$).

Napomenimo da je *quick sort* alogritam koji u praksi daje najbolje rezultate kod sortiranja dugačkih nizova. Međutim, važno je napomenuti da kod sortiranje kraćih nizova naivni algoritmi (npr. *insertion sort*) mogu da se pokažu praktičnijim. 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. Jedna od mogućih implementacija koraka particionisanja je sledeća:

```
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]` su veći ili jednaki od pivota. Nakon završetka petlje, `j` ima vrednost `d+1`, te su elementi `a[p+1, d]` veći ili jednaki od pivota. 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`).

Drugi način implementacije particionisanja je zasnovan na Dijkstrinom 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.

Treći mogući način implementacije koraka particionisanja je da se obilazi paralelno sa dva kraja i kada se na levom kraju nađe element koji je veći od pivota, a na desnoj neki koji je manji od pivota, da se izvrši njihova razmena.

```
int particionisanje(int a[], int l, int d) {
  int l0 = l;

  while (l < d) {
    while (a[l] <= a[l0] && l < d)
       l++;
    while (a[d] >= a[l0] && l < d)
       d--;
    if (l < d)
       razmeni(a, l, d);
  }

  if (a[l] >= a[l0])
    l--;
  razmeni(a, l0, l);
  return l;
}
```

Invarijanta spoljašnje petlje je da je `l` manje ili jednako `d`, da je (multi)skup elemenata niza `a` nepromenjen (što je očigledno jer algoritam zasniva isključivo

na razmenama), da se na poziciji `l0` nalazi pivot i da su elementi `a[l0, l-1]` manji od pivota (gde je `l0` početna vrednost promenljive `l`), a elementi `a[d+1, d0]` (gde je `d0` početna vrednost promenljive `d`) veći ili jednaki 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 su elementi `a[l0, l]` manji ili jednak od pivota, a elementi niza `a[l+1, d0]` veći ili jednaki od pivota. Odavde, nakon zamene pozicija `l0` i `l` postiže se postuslov particionisanja.

S obzirom da se u svakom koraku petlji smanjuje pozitivna celobrojna vrednost `d - l`, a sve petlje se zaustavljaju u slučaju kada je `d - l` nula zaustavljanje sledi.

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 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[l0, l-1]` manji od pivota (gde je `l0` početna vrednost promenljive `l`), a elementi `a[d+1, d0]` (gde je `d0` početna vrednost promenljive `d`) veći ili jednaki od pivota. 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.

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);
}
```

Bolje performanse se mogu postići ukoliko se npr. za pivot uzme srednji od tri slučajno izabrana elementa niza.

4.2.7 Korišćenje sistemske implementacije quick sort-a

Funkcija `qsort` (deklarisana u zagavlju `<stdlib.h>`) izvršava *quick sort* algoritam. Način korišćenja ove funkcije veoma je blizak načinu koiršćenja funkcije `bsearch`.

```
void qsort(void *base, size_t num, size_t width,
           int (*compare)(const void *elem1, const void *elem2));
```

Argumenti funkcije imaju sledeću ulogu:

`base` je pokazivač na početak niza koji se sortira; on je tipa `void*` kako bi mogao da prihti pokazivač na bilo koji konkretan tip;

`num` je broj elemenata niza;

`width` je veličina jednog elementa u bajtovima; ona je potrebna kako bi funkcija mogla da prelazi sa jednog na sledeći element niza;

`compare` je pokazivač na funkciju (često definisanu od strane korisnika) koja vrši poređenje dve vrednosti zadatog tipa; 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 vrednost na koju ukazuje prvi argument manja od vrednosti na koju ukazuje drugi argument;
- 0 ako su vrednosti na koju ukazuju prvi i drugi argument jednake;
- > 0 ako je vrednost na koju ukazuje prvi argument veća od vrednosti na koju ukazuje drugi argument;

Niz se sortira u rastućem poretku određenom funkcijom sortiranja.

Navedimo nekoliko primera. Naredni program gradi slučajan niz celih brojeva i uređuje ga rastuće.

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

int 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;
}
```

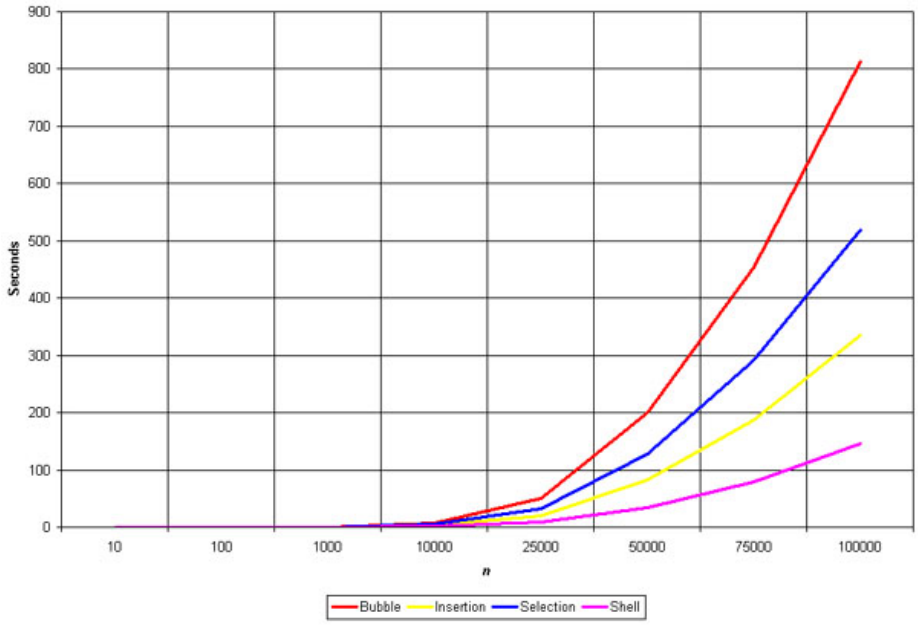
Naredni program leksikografski sortira argumente komandne linije.

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

int compare(const void *pa, const void *pb) {
   return strcmp(*(char**)pa, *(char**)pb);
}

void main( int argc, char **argv ) {
   int i;
   argv++; argc--;   /* argv[0] se ne sortira */
   qsort((void*)argv, (size_t)argc, sizeof(char*), compare);
   for(i = 0; i < argc; i++)
      printf("%s ", argv[i]);
   printf("\n");
}
```

Primer korišćenja:



Slika 4.1: Rezultati za algoritme sortiranja složenosti $O(n^2)$

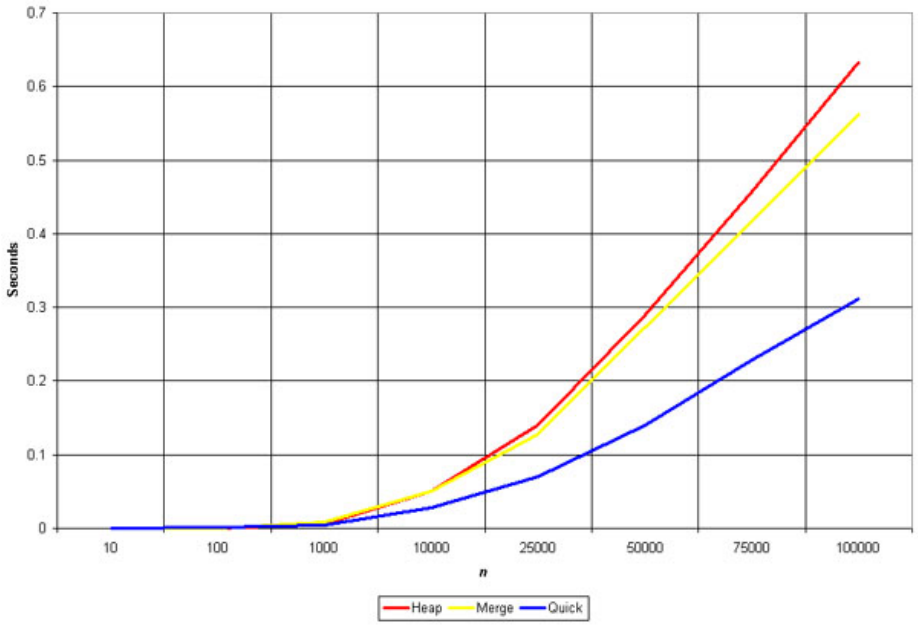
```
# ./qsort every good boy deserves favor
# boy deserves every favor good
```

4.2.8 Empirijsko poređenje različitih algoritama sortiranja

In addition to algorithmic complexity, the speed of the various sorts can be compared with empirical data. Since the speed of a sort can vary greatly depending on what data set it sorts, accurate empirical results require several runs of the sort be made and the results averaged together. The empirical data given is the average of a hundred runs against random data sets on a single-user 250MHz UltraSPARC II. The run times on your system will almost certainly vary from these results, but the relative speeds should be the same - the selection sort runs in roughly half the time of the bubble sort on the UltraSPARC II, and it should run in roughly half the time on whatever system you use as well.

These empirical efficiency graphs (4.1 and 4.1) are kind of like golf - the lowest line is the "best". Keep in mind that "bestđepends on your situation - the quick sort may look like the fastest sort, but using it to sort a list of 20 items is kind of like going after a fly with a sledgehammer.

As the graph 4.1 pretty plainly shows, the bubble sort is grossly inefficient, and the shell sort blows it out of the water. Notice that the first horizontal line in the plot area is 100 seconds - these aren't sorts that you want to use for huge



Slika 4.2: Rezultati za algoritme sortiranja složenosti $O(n \log n)$

amounts of data in an interactive application. Even using the shell sort, users are going to be twiddling their thumbs if you try to sort much more than 10,000 data items.

On the bright side, all of these algorithms are incredibly simple (with the possible exception of the shell sort). For quick test programs, rapid prototypes, or internal-use software they're not bad choices unless you really think you need split-second efficiency.

Speaking of split-second efficiency, the $O(n \log n)$ sorts (slika 4.2) are where it's at. Notice that the time on this graph is measured in tenths of seconds, instead hundreds of seconds like the $O(n^2)$ graph.

But as with everything else in the real world, there are trade-offs. These algorithms are blazingly fast, but that speed comes at the cost of complexity. Recursion, advanced data structures, multiple arrays - these algorithms make extensive use of those nasty things.

In the end, the important thing is to pick the sorting algorithm that you think is appropriate for the task at hand. You should be able to use the source code on this site as a "black box"if you need to - you can just use it, without understanding how it works. Obviously taking the time to understand how the algorithm you choose works is preferable, but time constraints are a fact of life.

The selection sort is in the group of n^2 sorts. It yields a 60% performance improvement over the bubble sort, but the insertion sort is over twice as fast as the bubble sort and is just as easy to implement as the selection sort. In

short, there really isn't any reason to use the selection sort - use the insertion sort instead.

If you really want to use the selection sort for some reason, try to avoid sorting lists of more than a 1000 items with it or repetitively sorting lists of more than a couple hundred items.

The insertion sort is a good middle-of-the-road choice for sorting lists of a few thousand items or less. The algorithm is significantly simpler than the shell sort, with only a small trade-off in efficiency. At the same time, the insertion sort is over twice as fast as the bubble sort and almost 40% faster than the selection sort. The insertion sort shouldn't be used for sorting lists larger than a couple thousand items or repetitive sorting of lists larger than a couple hundred items.

Realistically, there isn't a noticeable performance difference between the various sorts for 100 items or less, and the simplicity of the bubble sort makes it attractive. The bubble sort shouldn't be used for repetitive sorts or sorts of more than a couple hundred items.

4.3 Jednostavni algebarsko-numerički algoritmi

4.3.1 Izračunavanje vrednosti polinoma

Any polynomial in the form

$$P(x) = ax^n + bx^{n-1} + \ldots dx + e$$

can be expressed in a form requiring fewer operations. For example:

$$P(x) = ax^3 + bx^2 + cx + d = x(x(ax + b) + c) + d$$

4.3.2 Karacubin algoritam

4.4 Generisanje kombinatornih objekata

4.4.1 Varijacije

4.4.2 Permutacije

```
#include<stdio.h>

#define N 3 static int used[N+1]; static int p[N+1];

void permutation(int n) ;

main() {
    int i;
    for (i=1;i<=N;i++)
        used[i]=0;
    permutation(1);
}

void permutation(int n) {
    int i;
    if(n>N)
    {
        for(i=1;i<=N;i++)
            printf("%d ",p[i]);
        printf("\n");
    }
    else
        for (i=1;i<=N;i++)
            if (!used[i])
            {
                used[i]=1;
                p[n]=i;
                permutation(n+1);
                used[i]=0;
            }
}
```

Rezultat rada za N=3:

```
1 2 3 1 3 2 2 1 3 2 3 1 3 1 2 3 2 1
```

Druga varijanta:

```
#include<stdio.h>

#define N 3 static int p[N+1];

void permutation(int n) ;

main() {
    for (i=1;i<=N;i++)
        p[i]=i;

    permutation(1);
}

void permutation(int n) {
    int i,tmp;
    if(n==N)
    {
        for(i=1;i<=N;i++)
            printf("%d ",p[i]);
        printf("\n");
    }
    else
        for (i=n;i<=N;i++)
        {
            tmp=p[n];
            p[n]=p[i];
            p[i]=tmp;

            permutation(n+1);

            tmp=p[n];
            p[n]=p[i];
            p[i]=tmp;
        }
}
```

4.4.3 Kombinacije

4.4.4 Particionisanje

A partition of a positive integer n is a sequence of positive integers that sum to n. Write a program to print all partitions of n, e.g:

```
n=4 4 3 1 2 2 2 1 1 1 3 1 2 1 1 1 2 1 1 1 1
```

Rešenje:

```
#include<stdio.h>

#define N 10 static int p[N];

void partition(unsigned int n);

main() {
    partition(4);
}

void partition(unsigned int n) {
    unsigned int i;
    static unsigned int k;

    if(n==0)
    {
        for(i=0;i<k;i++)
            printf("%d ",p[i]);
        printf("\n");
        return;
    }

    for(i=n;i>0;i--)
    {
        p[k]=i;
        k++;
        partition(n-i);
        k--;
    }
}
```

4.5 Algoritmi zasnovani na bitovskim operatorima

Glava 5

Fundamentalne strukture podataka

An array is a list of elements with a fixed size, accessed by index. A more flexible data structure is the linked list. Linked lists, stacks, queues, hash tables, trees are all different types of data structures that can help accommodate almost any type of data.

5.1 Liste

Linked lists are the most basic self-referential structures. Linked lists allow you to have a chain of structs with related data.

A linked list is composed of nodes, each of which contains a piece of data and a reference to the next node.

- A linked list can grow and shrink as much as needed.
- Adding an element to a linked list is $O(1)$.
- Finding an element in a linked list is $O(n)$

So how would you go about declaring a linked list? It would involve a struct and a pointer:

```
struct llnode
{
  <type> data;
  struct llnode *next;
};
```

The `<type>` signifies data of any type. This is typically a pointer to something, usually another struct. The next line is the next pointer to another `llnode` struct. Another more convenient way using `typedef`:

```
typedef struct list_node
{
  <type> data;
  struct list_node *next;
} llnode;

llnode *head = NULL;
```

Note that even the `typedef` is specified, the next pointer within the struct must still have the struct tag! There are two ways to create the root node of the linked list. One method is to create a head pointer and the other way is to create a dummy node. It's usually easier to create a head pointer. Now that we have a node declaration down, how do we add or remove from our linked list? Simple! Create functions to do additions, removals, and traversals (*oblizak*).

Additions: A sample Linked list addition function:

```
void add(llnode **head, <type> data_in) {
  llnode *tmp;

  if ((tmp = malloc(sizeof(*tmp))) == NULL) {
    ERR_MSG(malloc);
    (void)exit(EXIT_FAILURE);
  }
  tmp->data = data_in;
  tmp->next = *head;
  *head = tmp;
}

/* ... inside some function ... */
llnode *head = NULL;
<type> *some_data;
/* ... initialize some_data ... */

add(&head, some_data);
```

(*Napomena:* umesto `(*tmp).data` može da se piše `tmp->data` (isto važi za bilo koji pokazivač na strukturu, dakle umesto `(*a).member` može da se piše `a->member`).)

What's happening here? We created a head pointer, and then sent the address-of the head pointer into the add function which is expecting a pointer to a pointer. We send in the address-of head. Inside add, a `tmp` pointer is allocated on the heap. The data pointer on `tmp` is moved to point to the `data_in`. The next pointer is moved to point to the head pointer `(*head)`. Then the head pointer is moved to point to `tmp`. Thus we have added to the beginning of the list.

Removals: You traverse the list, querying the next struct in the list for the target. If you get a match, set the current target next's pointer to the pointer of the next pointer of the target. Don't forget to free the node you are removing

(or you'll get a memory leak)! You need to take into consideration if the target is the first node in the list. There are many ways to do this (i.e. recursively). Think about it!

Traversals: Traversing list is simple, just query the data part of the node for pertinent information as you move from next to next. What about freeing the whole list? You can't just free the head pointer! You have to free the list. A sample function to free a complete list:

```
void freelist(llnode *head)
{
  llnode *tmp;
  while (head != NULL)
  {
    free(head->data);  /* Don't forget to free memory within the list! */
    tmp = head->next;
    free(head);
    head = tmp;
  }
}
```

Now we can rest easy at night because we won't have memory leaks in our lists!

5.1.1 Stek (LIFO lista)

A stack is a basic data structure that is used all through out programming. Stacks are a specific kind of linked list. They are referred to as LIFO or Last In First Out. A stack is called a LIFO (Last In First Out) to demonstrate the way it accesses data.

A stack has two basic operations:

push puts an element on top of the stack.

pop takes an element from the top of the stack.

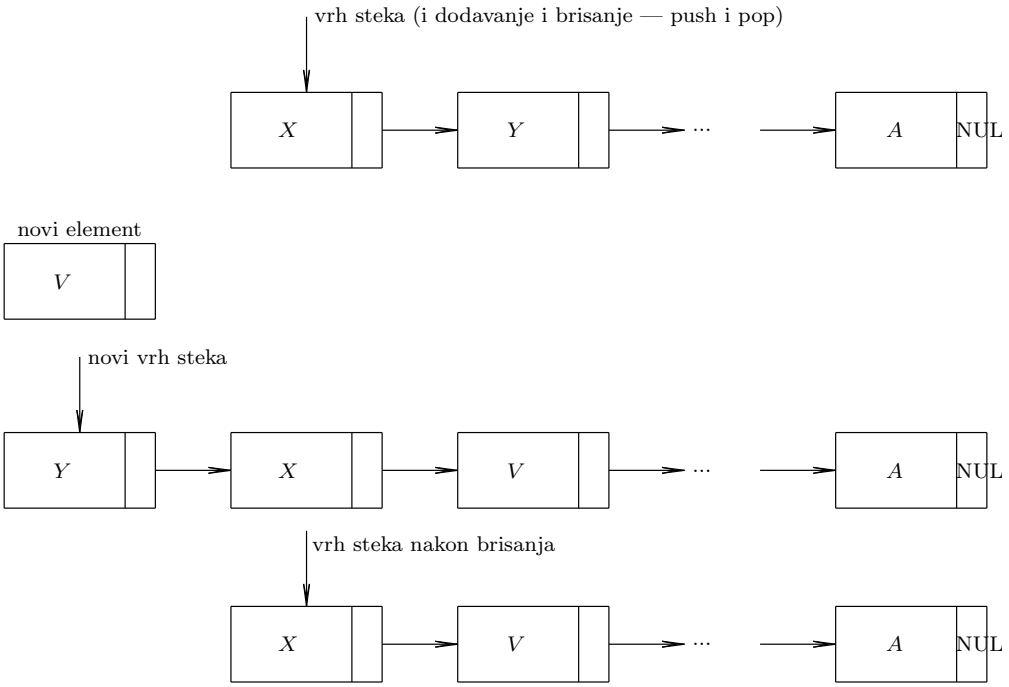
Stacks have specific adds and removes called push and pop. Pushing nodes onto stacks is easily done by adding to the front of the list. Popping is simply removing from the front of the list. It would be wise to give return values when pushing and popping from stacks. For example, pop can return the struct that was popped.

The idea is to think of your data as a stack of plates or books where you can only take the top item off the stack in order to remove things from it.

We can implement a stack with a linked list.

5.1.2 Red (FIFO lista)

A queue is a basic data structure that is used throughout programming. Queues are FIFO or First In First Out. Think of a typical (non-priority) printer



Slika 5.1: Stek

queue: The first jobs submitted are printed before jobs that are submitted after them. You can think of it as a line in a grocery store. The first one in the line is the first one to be served. A queue is called a FIFO (First In First Out) to demonstrate the way it accesses data.

Red se može ilustrovati i na primereu čekaonice u kojoj svako zna ko je na redu posle njega.

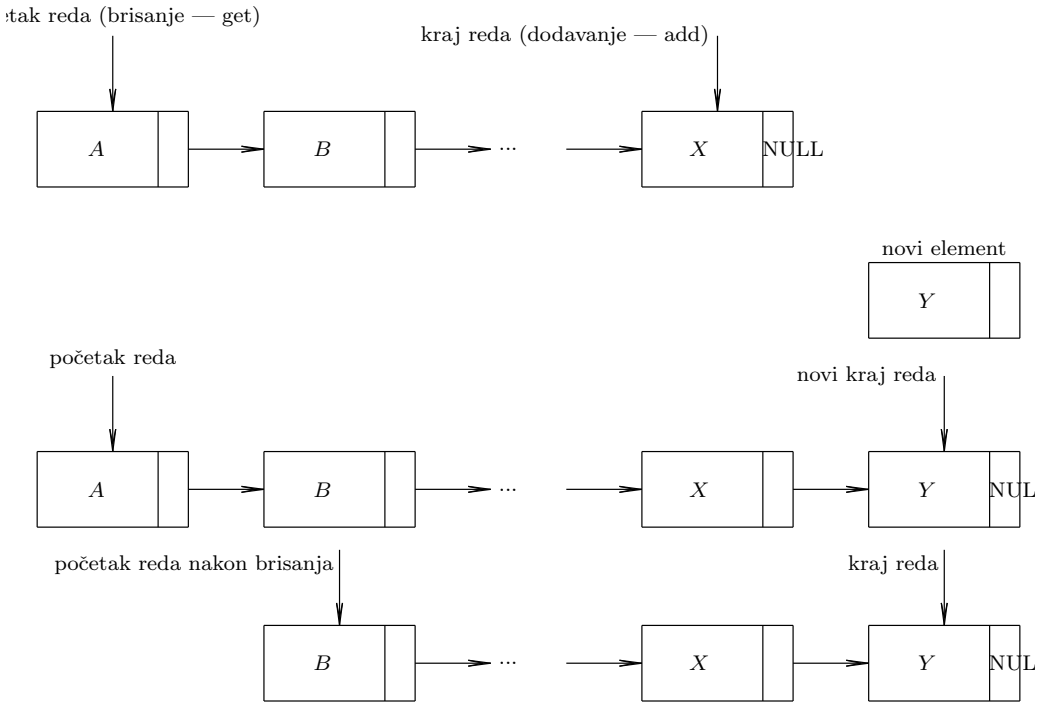
A queue has two basic operations:

add puts an element at the end of the queue.

get takes an element from the front of the queue.

Queues aren't more difficult to implement than stacks. By creating a tail pointer you can keep track of both the front and the tail ends of the list. So you can enqueue onto the tail of the list, and dequeue from the front of the list!

We can implement a queue with a linked list.



Slika 5.2: Red

5.1.3 Dvostruko povezane (dvostruko ulančane) liste

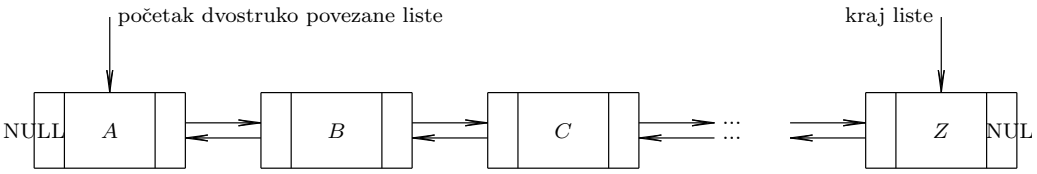
Each element in a singly-linked list contains a single reference – a reference to the successor (next) element of the list. As a result, deleting the head of the linked list is easy: The new head is the successor of the old head.

However, deleting the tail of a linked list is not so easy: The new tail is the predecessor of the original tail. Since there is no reference from the original tail to its predecessor, the predecessor must be found by traversing the linked list from the head. This traversal gives rise to the $O(n)$ running time.

In a doubly-linked list , each list element contains two references–one to its successor and one to its predecessor. There are many different variations of doubly-linked lists.

A doubly linked list allows us to traverse the list backwards and forwards. Each node has a reference to both the next and the previous node.

Double linked lists allow us to very quickly ($O(1)$) remove an element from either the front or the back of the list. Removing the last element of a singly linked list is $O(n)$.

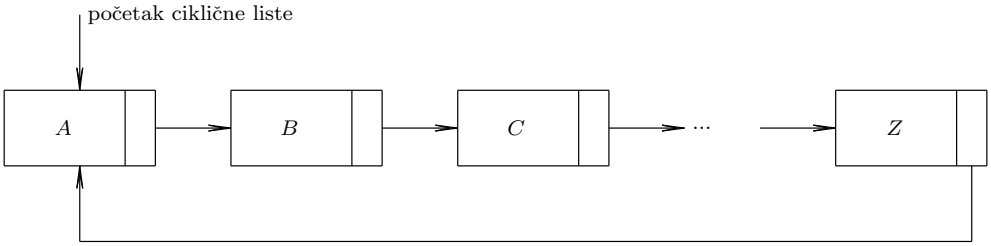


Slika 5.3: Dvostruko povezana lista

5.1.4 Kružne (ciklične, cirkularne) liste

A list can šwallow its tail"by having the pointer in the tail item contain the location of the first item, instead of NULL. Now, the position of the head is rather arbitrary, for it is possible to get from any item to any other one. If items were frequently added near each other, it might make sense to change the head pointer so that it always pointed to the most recent insertion.

A list in which the last item points to the first item is said to be *circular*.



Slika 5.4: Dvostruko povezana lista

In a circular list, there is in a sense no such thing as a last item, but it is still useful to keep a tail pointer to find the last logical item–after all, it can no longer be found by searching for a NULL pointer. It might also be useful to maintain a variable that stores the number of active items in the list, so that a traverse with a high index number does not waste time by going around the circle and examining items more than once. In such a list, inserting before the head changes the head pointer to point to the new item, and inserting after the tail changes the tail pointer to point to the new item, but both place a new item after the old tail and before the old head in the circle.

5.1.5 Liste: primer

Naredni primer ilustruje rad sa listama. Lista koja se koristi je jednostruko povezana i za nju je implementirano više funkcija nego što je neophodno za stek i red (npr. implentirane su funkcije i za ubacivanje na početak i za ubacivanje na kraj liste).

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

typedef struct cvor
{
    int br;
    struct cvor* sl;
} CVOR;

/* Pomocna funkcija koja kreira cvor liste
   sa datim sadrzajem.
   Funkcija kreira cvor i postavlja mu sadrzaj
   na dati broj.
   Polje sl ostaje nedefinisano.
   Funkcija vraca pokazivac na kreirani cvor. */

CVOR* napravi_cvor(int br)
{
    CVOR* novi = (CVOR*)malloc(sizeof(CVOR));
    if (novi == NULL)
    {
        fprintf(stderr, "Greska prilikom
                    alokacije memorije\n");
        exit(1);   /* OVO NIJE LEPO - NIJE OSLOBODJEN ALOCIRANI PROSTOR */
    }
    novi->br = br;
    return novi;
}

/* ------------------------------------------- */

/* Ispisivanje liste : iterativna verzija */
void ispisi_listu_i(CVOR* l)
{
    CVOR* t;
    for (t = l; t != NULL; t=t->sl)
        printf("%d ", t->br);
}

/* Ispisivanje liste : rekurzivna verzija */
void ispisi_listu_r(CVOR* l)
{
    if (l != NULL)
    {
        printf("%d ", l->br);
        ispisi_listu_r(l->sl);
    }
}

/* Ispisivanje liste unatrag : rekurzivna verzija */
/* Prethodna funkcija se lako modifikuje tako da
   ispisuje listu unazad */

void ispisi_listu_unazad(CVOR* l)
{
    if (l != NULL)
```

5.2 Stabla

Trees are natural structures for representing certain kinds of hierarchical data. A (rooted) tree consists of a set of nodes (or vertices) and a set of arcs (or edges). Each arc links a parent node to one of the parent's children. A special root node has no parent. Every other node has exactly one parent. It is possible to reach any node by following a unique path of arcs from the root. If arcs are considered bidirectional, there is a unique path between any two nodes. The simplest kind of tree is a binary tree where each parent has at most two children. A simple binary tree involves having two types of "next"pointers, a left and a right pointer. You can halve your access times by splitting your data into two different paths, while keeping a uniform data structure. But trees can degrade into linked list efficiency. There are different types of trees, some popular ones are self-balancing. For instance, AVL trees are a typical type of tree that can move nodes around so that the tree is balanced without a > 1 height difference between levels. [1]

5.2.1 Binarna stabla

Until now, we've only looked at lists that have only one đimension": forward/backward or next/previous. Consider a structure that acts as a "parent"and has at most two čhildren"(a binary tree)

```
struct Node
{
    int Value;
    struct Node *Left;
    struct Node *Right;
};
```

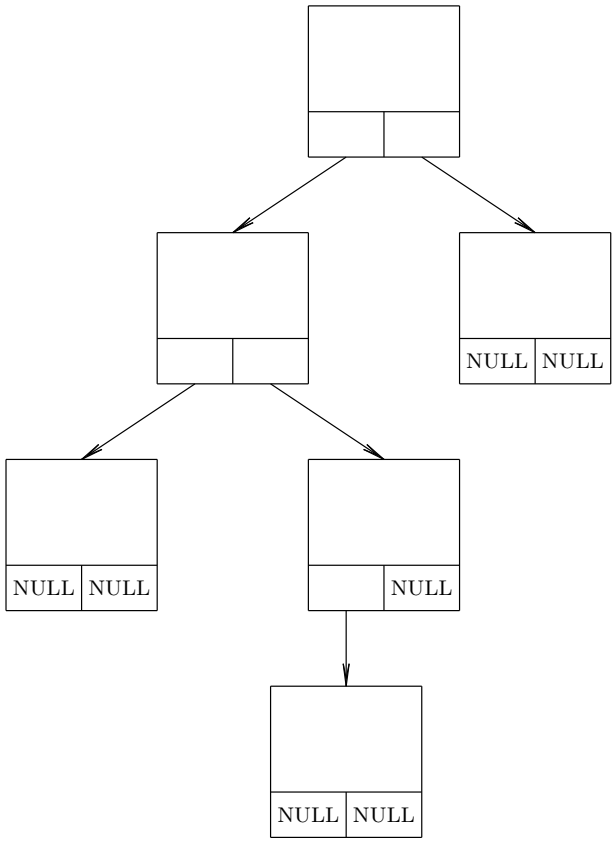
Trees are fun to use because you can easily add more children to the existing children.

5.2.2 Uređena binarna stabla

- With the trees we're working with, the left child always has a Value less than or equal to the parent's Value. The right child always has a Value greater than the parent's Value.
- You can always add a new child in the proper position (to the left or right of the parent).
- The tree is always fully sorted (how)?
- The tree is easily searchable.

Creating a binary ordered tree from a random data set (also called binary search tree):

[1] Više o stablima pročitati na adresi: http://www.csse.monash.edu.au/ lloyd/tildeAlgDS/Tree/

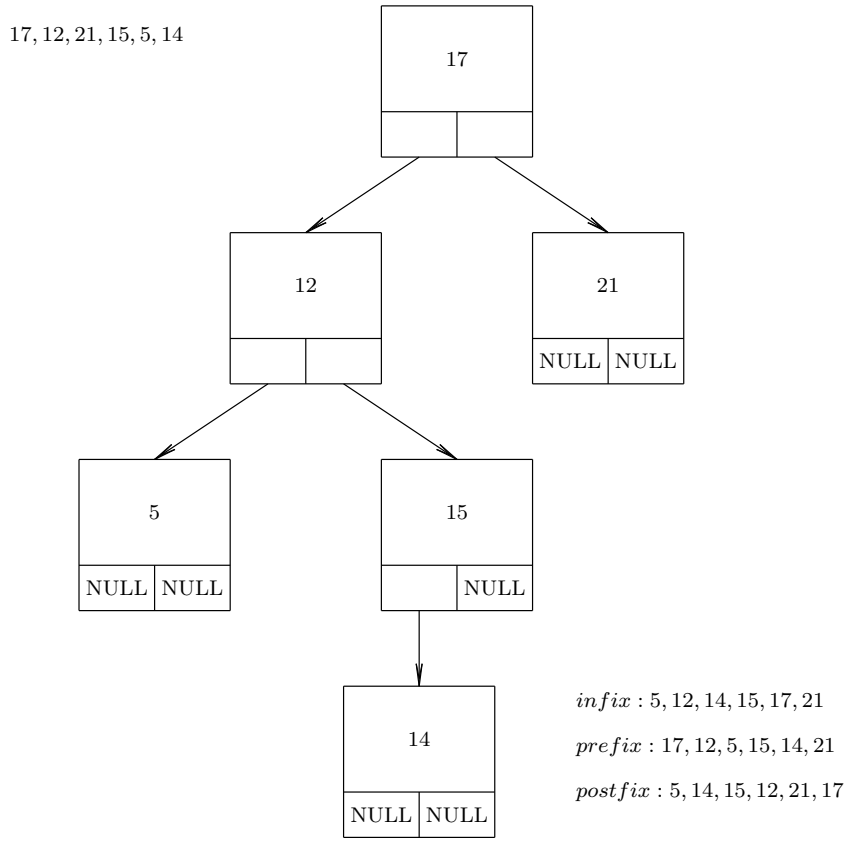


Slika 5.5: Stablo

- First element stored in the root node
- Compare the second element with the root node, if the new value is less than the parent's value, then move left (e.g. `Pointer = Pointer->Left`). Otherwise go right.
- Keep on comparing new value and keep on moving (left or right) until you reach a NULL pointer. Append the new node at that location.

Searching an ordered binary tree is just as easy as inserting something in a tree:

1. Set a `Pointer` to point at the root structure.
2. If the value we're looking for == the Pointer value, return the `Pointer`.



Slika 5.6: Uređeno stablo

3. If the value we're looking for is < the Pointer value, go left. (e.g. `Pointer = Pointer->Left`) And goto (2)

4. Otherwise go right and goto (2)

5. If the `Pointer` is ever NULL, return NULL to indicate that the value was not found in the tree.

Most of the tree functions can be implemented using recursion. The code is easily readable and understandable.

Kreiranje elementa

```
struct Node *Create_Node(int Value)
{
    struct Node *Ptr = NULL;
    Ptr = malloc(sizeof(struct Node));
    assert(Ptr != NULL);
    Ptr->Left = NULL;
    Ptr->Right = NULL;
    Ptr->Value = Value;
}
```

Dodavanje elementa

Iterativna verzija dodavanja elementa:

```
void Insert_Node(struct Node *Root, struct Node *New)
{
    while(1)
    {
        if (New->Value <= Root->Value)
            if (Root->Left == NULL)
            {
                Root->Left = New;
                return;
            }
            else
                Root = Root->Left;
        else
            if (Root->Right == NULL)
            {
                Root->Right = New;
                return;
            }
            else
                Root = Root->Right;
    }
}
```

Rekurzivna verzija dodavanja elementa:

```
void Insert_Node(struct Node *Root, struct Node *New)
{
    if (New->Value <= Root->Value)
        if (Root->Left == NULL)
        {
            Root->Left = New;
            return;
        }
        else
            Insert_Node(Root->Left, New);
    else
        if (Root->Right == NULL)
        {
            Root->Right = New;
            return;
        }
        else

    Insert_Node(Root->Right, New);
}
```

Napomena: umesto `(*New).Value` može da se piše `New->Value` (isto važi za bilo koji pokazivač na strukturu, dakle umesto `(*a).member` može da se piše `a->member`).

Pronalaženje elementa

Searching an ordered binary tree is just as easy as inserting something in a tree:

1. Set a `Pointer` to point at the root structure.
2. If the value we're looking for == the `Pointer` value, return the `Pointer`.
3. If the value we're looking for is < the `Pointer` value, go left. (e.g. `Pointer = Pointer->Left`) And goto (2)
4. Otherwise go right and goto (2)
5. If the `Pointer` is ever NULL, return NULL to indicate that the value was not found in the tree.

```
struct Node *Tree_Find(struct Node *Root,int Value)
{
    if (Root == NULL)
        return NULL; /* Not found */
    if (Value == Root->Value)
        return Root; /* Found it */
    if (Value < Root->Value) /* Go left */
        return Tree_Find(Root->Left, Value);
    return Tree_Find(Root->Right, Value);
}
```

Obilazak stabla u dubinu

How do we get at the sorted content of a tree?

- We know that an ordered binary tree is fully sorted. We'd like to take advantage of that.
- The "least"element in the tree is at the far left.
- The "greatest"element is at the far right.
- Our tree nodes do not point back to their parents.
- How can we start at the far left and go through each node in order?

Accessing each of the nodes of a tree in order is often called Tree Traversal or Iterating over a Tree. We can do this in several ways:

- infix, Least to greatest: For each node, access the left node recursively, then the node itself, then the right node recursively. (Abbreviated L-N-R)
- infix, Greatest to least: Same way except R-N-L.
- Prefix: N-L-R
- Postfix: L-R-N

Example of ordered printing (infix, Least to greatest):

```
void Print_Tree(struct Node *Ptr)
{
    if (Ptr == NULL)
        return;
    Print_Tree(Ptr->Left);      /* Go left */
    printf("%d\n", Ptr->Value); /* Node */
    Print_Tree(Ptr->Right);    /* Go right */
}
```

Primer 5.1. *Primer za vežbu: napraviti ispis elemenata stabla kao u stablu koje prikazuje direktorijume (npr. u Windows Exploreru)*

Obilazak stabla u širinu

U obilasku stavla u širinu, obilazi se (i obrađuje) nivo po nivo stavla.

A breadth-first traversal of a tree starts at the root of the tree. It next visits the children, then the grand-children and so on.

The numbers indicate the order in which the nodes are visited, not the contents of the nodes. Because children are only accessible from a parent, they must be stored while the parent's siblings and cousins are visited. Usually, a queue is used to do this.

5.2.3 Izrazi u formi stabla

Matematički izrazi prirodno se mogu reprezentovati u vidu stabla. Na primer, izraz `3*(4+5)` može se reprezentovati kao stablo u čijem je korenu `*`, sa levim podstablom `3`, i sa desnim potomkom čvor `+`, koji ima 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 reprezentovanog stablom može se izračunati jednostavnom funkcijom.

5.3 Grafovi

5.4 Heš tabele

Deo II

Principi razvoja programa

Glava 6

Rešavanje problema uz pomoć računara

Rešavanja problema uz pomoć računara podrazumeva konstruisanje (pisanje) programa na nekom programskom jeziku, čijim izvršavanjem će se rešavati svaki pojedinačni zadatak tog problema. Da bi se program proizveo, potrebno je odabrati metodu za rešavanje problema, poznavati (ili razviti) algoritam za rešavanje problema, poznavati programski jezik, uveriti se u korektnost programa njegovim izvršavanjem za razne kombinacije ulaznih podataka. Jedini jezik na kome računar može da izvršava program jeste mašinski jezik tog računara. To je jezik najnižeg nivoa, u kome su sve "naredbe"(tzv. instrukcije) i podaci predstavljeni binarnom azbukom, i koji je zbog toga neodgovarajući za čoveka. Da bi se program izvršavao, potrebno je imati programske alate koji će omogućiti rad sa našim programom, od unošenja u računar, preko prevođenja na mašinski jezik, do testiranja programa i otkrivanja grešaka. Ako su u programu otkrivene greške, program se "ispravlja"(ponavlja se faza pisanja programa), a ako je potrebno, primenjuje se (ili razvija) drugi algoritam, odnosno menja metoda rešavanja problema. Pošto se izradi korektan i efikasan program za rešavanje zadatog problema, on se primenjuje kraće ili duže vreme, uz eventualne povremene modifikacije koje odgovaraju promenama u postavci problema. Dakle, u rešavanju problema uz pomoć računara javlja se veći broj poslova koje treba obaviti. Poslovi koji se javljaju u procesu rešavanja problema uz pomoć računara mogu se, koncepcijski, podeliti u nekoliko faza, koje čine životni ciklus programa. Zivotni ciklus programa je vreme od početka razvoja programa (od početka rešavanja problema uz pomoć računara), do kraja korišćenja programa, a sam termin asocira na ponovljeno (ciklično) izvršavanje tih faza, što i odgovara prirodi ovog procesa. Te faze su sledeće:

1. specifikovanje problema
2. projektovanje (eventualno sa analizom složenosti rešenja)
3. realizacija

4. testiranje (i, eventualno, dokazivanje korektnosti)
5. izrada dokumentacije
6. eksploatacija i održavanje

Specifikovanje problema je postupak opisivanja problema za koji se traži programsko rešenje. Specifikacijom je potrebno što preciznije opisati problem, prirodu ulaznih podataka i oblik u kome se žele rešenja — izlazni rezultati. Specifikacija programa bavi se pitanjem šta program treba da uradi, kojom brzinom, koja mu je maksimalna dozvoljena veličina, itd. Specifikacija se može izraziti raznim jezicima sa različitim sintaksama. Mada postoje i posebni jezici za pisanje formalne specifikacije, ona se može (u jednostavnijim slučajevima) izraziti i prirodnim jezikom, npr. šmestiti u z proizvod brojeva a i b, pod pretpostavkom da su a i b — celi brojevi i $a \geq 0$, $b \geq 0$.

Projektovanje predstavlja razumevanje problema, izradu matematičkog modela i izbor ili razvoj/konstrukciju odgovarajućeg algoritma.

Realizacija predstavlja implementaciju, ili ostvarenje rešenja. Ona uključuje detaljnu razradu algoritma, izbor struktura podataka i medija na kojima će se podaci držati (sa kojih će se unositi ili na koje će se izdavati), izradu sintaksno ispravnog programa na izabranom programskom jeziku (programa koji je napisan prema pravilima kojima je definisan taj jezik). Da bi se utvrdila sintaksna ispravnost programa, potrebno je uneti program u računar (npr. preko terminala, korišćenjem programa — editora), prevesti program na mašinski jezik — pomoću programa — prevodioca (pri čemu se ispituje sintaksna ispravnost) i izvršiti ga (npr. uz pomoć programa za otkrivanje grešaka — debagera), ili ga interpretirati (analizirati sintaksnu ispravnost programa i prevoditi ga deo po deo uz istovremeno izvršavanje analiziranih delova). Ako se pri prevođenju programa ili njegovom interpretiranju i izvršavanju ustanovi neka greška, potrebno je ispraviti program, ponovo ga prevesti i izvršiti.

Testiranje predstavlja, pre svega, proces utvrđivanja semantičke ispravnosti programa, tj. uveravanja, u što je moguće većem stepenu, da program izvršava željenu funkciju, tj. da odgovara specifikaciji, tj. da rešava postavljeni problem. Ukoliko je formalno dokazana korektnost programa, ili je program razvijen formalnim metodama iz formalne specifikacije, za njega se može garantovati da je semantički korektan, do na tipografske greške. Tada je testiranje maksimalno pojednostavljeno ili nepotrebno. U svim drugim slučajevima, testiranje je važna faza razvoja programa. Testiranje se obavlja višestrukim izvršavanjem programa za raznovrsne pripremljene ulazne podatke, za koje su nam poznati očekivani rezultati. Značajno je da pripremljeni ulazni podaci budu dobro odabrani tako da se za njihove razne kombinacije, pri raznim izvršavanjima programa, izvršavaju (testiraju) sve grane programa. Takvi ulazni podaci nazivaju se test primeri, i oni omogućuju proveru ispravnosti u što je moguće većoj meri, tako da program

ima veliku verovatnoću ispravnosti u primeni. Ovde je bitno naglasiti da se test primerima nikako ne može dokazati korektnost programa, ali se možemo, u visokom stepenu, uveriti da je program ispravan. Pri testiranju programa mogu se otkriti greške pri izvršavanju (npr. deljenje nulom), koje otkriva i prijavljuje sam računar tj. njegov operativni sistem, ili semantičke greške koje može otkriti samo čovek programer, poređenjem dobijenih rezultata izvršavanja programa sa očekivanim rezultatima. To mogu biti greške nastale pri pisanju programa (npr. umesto uvećanja promenljive i za 1 omaškom smo naveli uvećanje promenljive i za 10 u sintaksi C-a, i+ = 10), ali to mogu biti i greške u algoritmu (npr. sa brojačem ciklusa, i, pošli smo od vrednosti 0 umesto od vrednosti 1), ili greške u samom matematičkom modelu ili primenjenoj metodi (npr. za rešavanje sistema nehomogenih algebarskih jednačina primenjena je metoda kojom se rešava samo sistem homogenih algebarskih jednačina). Ako je greška ustanovljena na nivou programa, potrebno je vratiti se na korak realizacije. Ako je greška ustanovljena na nivou algoritma ili metode (modela), potrebno je vratiti se na fazu projektovanja, pri čemu se ponavljaju sve naredne faze. Pri testiranju programa pomaže izdavanje tekućih vrednosti promenljivih na karakterističnim mestima u programu, ili testiranje ulaznih podataka (tzv. logička kontrola podataka) da bismo se uverili da će program, kao ispravne ulazne podatke, prihvatiti samo podatke određenog (predviđenog) tipa, oblika i opsega.

Debager je alat za praćenje izvršavanja programa radi otkrivanja grešaga (bagova, eng. bugs). 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 sadržaj memorije, vrednosti promenljivih i druge elemente programa.

Izrada dokumentacije Kada je program dobro i uspešno istestiran, pristupa se izradi dobre, pregledne i detaljne dokumentacije. Dokumentacija je neophodna za uspešno korišćenje programa, i posebno za fazu održavanja programa koja prati njegovo korišćenje. Dokumentacija treba da sadrži specifikaciju problema, algoritam (globalni i detaljni), čitljivo napisan program (uz dobru meru komentara), način zadavanja ulaznih podataka i način izdavanja rezultata, značenje korišćenih imena (promenljivih, datoteka, programa, potprograma, itd.), rezultate testiranja, test primere, uputstva za korišćenje i održavanje programa. Eksploatacija i održavanje je faza korišćenja programa koja može da traje i duži niz godina. Održavanje programa podrazumeva izvesne modifikacije programa u toku njegove eksploatacije, koje su posledica promena samog problema u toku vremena, ili potrebe da se kvalitet programa (prostorna i vremenska efikasnost, primenljivost) poveća. Održavanje programa je lakše ako je program preglednije napisan i ako je dokumentacija kompletnija, posebno zbog toga što održavanje programa po pravilu ne vrši lice koje je program pisalo. Svaku izmenu u programu nastalu u fazi održavanja programa potrebno je takođe dokumentovati, tj. pored održavanja programa potrebno je održavati i njegovu dokumentaciju. Postoje alati koji olakšavaju kreiranje dokumentacije i delom je generišu automatski (npr. Doxygen).

Glava 7

Strukturna dekompozicija i druga načela pisanja programa

Ultimately,[1] the only way a piece of software can impact the world is by doing something useful. People learn that fact in colleges and try to make working software. For them, the important aspects of code become, in the order of priority, getting the task done, and getting it done correctly. As they get more experience, they realize that there are other important aspects of software:

- It costs more to maintain and modify the software than develop the software.
- A long living software is read more often than it is written.
- Usually, correctness of the code is correlated to the readability of the code.

All in all, we feel that the primary purpose of the code is to communicate to the other developers within the confines of a compiler specified medium. Even though readability is an auxiliary goal, aiming for it lets us get to the primary goal: working code that is correct.

Normally, as soon as people start a project, they agree on a set of coding guidelines. These coding guidelines spell out, in elaborate detail, what the syntactic conventions are: how to use underscores, camelHump style, and how to indent. While these guidelines are of good intentions, it ends up getting ignored for several reasons: they do not explain why these rules make sense; they do not say which rules are more important; they do not equip the coder with the rationale behind the rules.

[1] Zasnovano na tekstu Ramarao Kanneganti: Best Practices in Writing Readable Code

7.1 Pisanje čitljivih programa: vizualni elementi programa

Most of the code is read on the computers these days. People use their editor or their IDE (Intergrated Development Environment) to read and modify the code. Sometimes, they use the browser to read the code as well. People who write good code understand that. They make it easy to read code on the computer.

In fact, most coding guidelines are aimed at providing good readability. They capture the principles of readability in the form of specific rules of formatting, indenting, and naming rules. Naturally, such excessive specifications are violated. Or, even if people confirm to the rules, the intent behind the rules is violated. That is, the underlying principles of consistency and clarity that helps readability is violated.

Thankfully, these problems are easy to solve. These days we have tools that makes it easy to follow formatting guidelines. We are going to provide a configuration in the appendix. We also will explain the intent and the logic behind the rules so that the writers can make sure the final goals are met.

One of the aspects of formatting of the code is visual appeal of the code. It applies to spacing, grouping, and indenting of information. This issue is important enough that some languages attach meaning to indentation (example: python). The principle behind the visual appeal of a code is this: can you looking at the code on the screen find what you are looking for? Can you open a file and figure out if this is the file you need to look at? All our guidelines are around that principle. We are going to enumerate them here.

7.1.1 80 karaktera u liniji

These days people rarely print out code. They read the code in their editor or IDE. Even if the screen resolution is very good, people may have several windows open such that the code may be less than 80 columns.

Aha, you say. What about using the wordwrap feature?

Two problems: One is that the reader may want to print it out. Some printers will cut the characters beyond the specified width. The other is, some editors may not wrap the lines. For example, a browser does not wrap lines. Also, the wordwrap feature will not be able to do the right alignment. A structured program becomes a flattened prosaic statement. The readability of the code is reduced.

Then, what about the lines exceeding 80 characters?

More often than not, such code is a suspect. Either the depth of the code is too much, in which case, you are better off making it more modular, or the indirection is too much, in which case, you are better off introducing temporary, meaningful variables. This is a heuristic, but a darn good one.

In the rare cases where it is needed, you can go the next line and align the argument, or operator appropriately. When in doubt, run the Unix program

indent, and it will produce the indented program. It has gazillion options for you to tinker to produce the kind of output you want. For Java, there is jalopy that does all this and much more.

7.1.2 Broj naredbi po liniji

Leaving blank statements helps to focus the reader. For example, you can club all the related statements in a visual block. On the flip side, leaving blank lines where they are not needed is a bad idea. Excessive blank lines do not make code more readable. In fact, if any, the linguistic affinity between the elements get lost. Compare the following:

With blank lines:

```
for (i = n; i > 0; i --)
    {

        sq_n += 2*i - 1;
        if (sq_n > MAX_VAL)
            {
                printf("Exceeded size");
                exit(1);
            }
        }
```

Without Excessive blank lines:

```
for (i = n; i > 0; i --) {
    sq_n += 2*i - 1;
    if (sq_n > MAX_VAL) {
        printf("Exceeded size");
        exit(1);
    }
}
```

Blank lines create the effect of paragraphs in the code; thus they should be used to group all the statements that belong in a logical unit. If you can write a comment on what the next group doing and why, then perhaps that group of statements deserves to be together.

Excessive blank lines rob the legitimacy of the needed breaks. In addition, they take up valuable real estate on the screen.

Along with the žeroštatement lines, we need to be concerned with more than one statement for a line also. Conventional wisdom states that number of statements to a line should be restricted to one. The exceptions are simple initializations and simple statements. For example, `int i=10; int j=20` can be on one line. In fact, leaving those statements on one line focuses the reader better. However, if the initializations need explanations, place them on a separate line.

7.1.3 Nazubljivanje/uvlačenje teksta

There are many religious wars about indentation. Ultimately, there are two biggest properties you look for in any rules about indentation style: Consistency and readability. Within these parameters, the following suggestions can be made:

- Do not use tab characters in your code. That is, do not use tabs to indent your files. Fortunately, these days most editors offer an option to convert tabs into spaces as you type. Check your editor on how to turn that option on.

  Why is this important? Tabs expand to different number of spaces on different editors. For example, you can set tab length as 4 chars in one editor. When you open the file in another editor, it will show 8 spaces causing misalignment.

- Align the new line with the beginning of the expression at the same level on the previous line. i.e. Do block related statements together. Braces are useful for compilers, but humans still rely on indentation and blank lines to group statements together. Use as appropriate number of spaces for indentation: I personally use 2 spaces. It means, that I can keep my lines to 80 columns without resorting to continuation lines. Anything less than two spaces do not provide the visual cues needed for grouping statements.

These formatting issues can best be explained by an indenting program. If you follow these guidelines, you get these benefits.

- The reader can read most code on the modern terminals and on the paper.
- The reader will see the code as you intended (tabs to spaces).
- The reader can see the logic through the concrete structure of the code.

7.2 Imenovanje promenljivih i funkcija

For a compiler, as long as we are sticking to the language rules, it does not matter what kind of names we use for the entities in our programs: files, classes, objects, and variables. However, for readers, it matters how the variables are named. They are used to communicate the concepts at a domain level. Here are some of the do's and don'ts.

- It is OK to name loop variables as i and j. It is a long standing convention that treat i,j,k as loop variables.
- For common suffixes and prefixes, agree on a standard. For example, for number, people use "no"or "num"or use it as prefix or suffix. For example, number of books can be referred to as "booknum", "numbooks"and some other combinations. Establishing a common notation and stems would help come with names that are readily understandable.

- Avoid complicated Hungarian notation. You are not a type checker. It is the job of a compiler. Use the language that is closest to what people speak. For reference, Hungarian notation encodes type information of a variable in the name. For example, they use "p"as a prefix for variable name, if the variable is a pointer. Or, š"for String. I do not believe "ppiName"is a good variable name, as the type information is more important for a compiler than the reader of a program.

- Avoid using generic names such as User, customer. Use specific names just as in writing prose. If a system has several kind of users, use appropriate name as in administrator or clerk and so on. If a function computes averages, name it as `"computeAverages"` instead of `"doTask"`. If a boolean variable tests if the cart is empty, call it "isCartEmpty"instead of "flag".

- As far as camelHump notation (popular in Java world, where people use Capital letters to denote word break in the name), or `word_underscore` notation (popular in C world), use whichever the standard is in that language. I personally find underscores make the word break real and visible, but I am not going to fight the establishment on that one.

The real test is always this: If you can read the code over the phone so that listener can comprehend it, then the naming has achieved its objective. For example, naming the procedure as a verb like šetupChannel"reads better. Naming a function such as `a = average(1,2,3)` reads better than `a = computeAverage(1,2,3)`.

7.3 Komentari

Several good books have been written about this subject out of which I would recommend Čode Complete". The fundamental principle is that in general code should convey all the meaning to the readers. However, syntactic constraints may not make it possible. More over, unlike a book where we can recast the matter for different audiences, we cannot write the code for different audience. In several such cases, you can add comments to the code as an extension to the code.

Writing comments should not provide an excuse to write poorly structured code. Comments can only add information at a different level.

- Comments should explain the domain portion, not the code.

- Comments should tell the reader why and what you are doing it, rather than how.

- In case the code is tricky, explain the how, and tell them explicitly why it is tricky.

- Use commenting style that is easy to maintain.

7.3.1 Komentari ne treba da objačnjavaju kôd

The worst kind of comments are the ones that explain the code. E.g.:

`a = a +1 // Increment a by 1.`

Well written code is self explanatory. However, code cannot explain why something is being done. Comments can explain that part. For example, the earlier statement may become:

`a = a + 1 // Add a bonus point to the customer if he makes it here.`

Thus, comments can explain the intent, the domain portion.

In some cases, we may have to explain the code. For example, if you are implementing a complex search algorithm, you may have to explain in the comments what you are doing. If your code contains unexpected or unseen effects, you should use comments to explain that to the reader. E.g.:

```
isCartEmpty= true; if (! isCartEmpty) { ....} // Since other threads
may be accessing
// isCartEmpty, we need to do this test
```

Btw, the same principle can be stated as: Comments should not repeat information. If it does, there is every chance of the code and comments being out of sync, confusing the reader.

7.3.2 Komentari treba da su takvi da ih je moguće održavati

We often see comments that are formatted beautifully, often by hand. For example, see this:

```
/**************************
* Author: Rama            *
**************************/
```

If the name of the author changes, then the alignment can get messed up. So follow a simple style of comments with using blank lines and spaces to communicate the ideas. Do not depend on complex formatting. You can correct the previous code with:

```
// Author: Rama
```

7.3.3 Komentari treba da budu koncizni

Since reading any text takes time, we should make comments as brief as possible. You can refer to other portions of the code to simplify the comments. As much as possible, use the comments to explain the domain relating to the requirements and use cases.

7.3.4 Korišćenje specifičnih komentara

Several experienced coders use standard markers in the code to denote certain standard concepts in the code. These markers can easily be picked out by

programs such as "grepšo that they can be acted upon. Most popular markers are:

- TODO marker: It is used to identify the tasks yet to be done. In fact, some IDEs like Eclipse can even show a list of TODO tasks. It is an excellent way to keep track of things to be done in the code. Make sure that all of the TODO is one line, so that it can be "grepped". It is also a good practice not to write to many TODO's clubbed at one place. For example, the following is not preferred.

  ```
  //TODO: Add Street name
  //TODO: Add second phone number
  //TODO: Add mobile phone number
  ```

  Instead, the following works well. It is one line and it clubs all of them into one TODO.

  ```
  //TODO: Add second address field to the customer class.
  ```

- FIXME marker: In the code, there are several places, you may take short cuts knowing fully well that it needs to be fixed later. In such cases, you can use FIXME marker to denote that activity. While TODO denotes unfinished activity, FIXME denotes code that needs fixing later on. For example, consider the following:

  ```
  //FIXME: Used only 50 chars for the name. Make it dynamic string.
  char name[50];
  ```

- XXX Marker: This marker is private marker. It is generally used only for the developer to be deleted by the time he is done. It is generally used as note to the developer.

7.4 Pisanje programa: modularnost i podela na datoteke

The preceding section described the ways in which you can use formatting to communicate the program better to the readers. Programming languages have mechanisms such as functions to create groups of statements. These functional, object decomposition has several purposes including reuse.

The physical structure of a large program is exposed through files. Each file again structured through functions (even in the OO world). So, to properly structure the program we need to structure the files. Here, I am not going to describe only the physical structure of the files. In the later section I will describe the logical structure of the files.

7.4.1 Deljenje programa u više datoteka

How do we break program into files? In most OO languages, each class gets a file. In non OO language, each cohesive set of functions get a file. In either case, we need to follow these guidelines.

- File length should not exceed a few hundred lines. Normal recommendation is 200 or so. However, if logic demands that the file should be larger, try using auxiliary classes or auxiliary courses.
- Make sure each file contains at least sizable information. Excessive fragmentation of a program through large number of files can make it difficult to read.
- If a group of developers are working on the project, make sure that a file is under control of one developer. That is, to understand and work with that file it should not take knowledge from disparate areas of the the domain or programming. For example, if a file contains knowledge about SQL, parsing, and numerical computing, there are not many people who know enough about these areas to own the file.

In addition to these reasons, a proper decomposition of the code into files help version control in the groups. Most version control systems treat files as a unit and provide versioning at only file level.

7.4.2 Modularnost

There are several ways the application can be made modular. Files, classes, and other structural elements help to organize the program.

One important purpose in organizing the program is to isolate the moving parts. That is, any program contains mature code and the code that is undergoing rapid transformation. It contains fixed entities and the entities that need customizing. A good coding style separates these so that customizing, modifying, and extending programs becomes easy.

Since customization and extensions of the code typically deals with a small subset of the code, it is a good idea to isolate that code. Use separate functions and files that encapsulates the changing code.

7.4.3 Kako koristiti konstante u programu

That is, unless they are cosmic constants (say π or e), people want to change the parameters in the code. So, do not use constants in there. For example, the following is bad.

```
char streetName [50];
```

We can correct it by:

```
#define FIFTY 50 char streetName[FIFTY];
```

Still bad. What if we use the length in multiple places and we want to change it to 75? It would read

```
#define FIFTY 75.
```

We can make it more readable by:

```
#define arrayLen 50 char streetName[arrayLen];
```

Even this can be made better by referring to the constant by its semantic significance. For example, it can be street name length. In which case, it be made:

```
#define STREETNAMELENGTH 50 streetName char[STREETNAMELENGTH];
```

As the next step, you collect all these constants into a separate file and through `#include` mechanism to use in other parts of the code. Of course, you should use language facilities to define constants instead of `#define`. So the next version will be:

```
const int STREETNAMELENGTH=50      /* To be placed in a central file.
*/ char streetName[STREETNAMELENGTH]; /* Declare it where used.
*/
```

7.5 Pisanje programa: dizajniranje programa

Have you read any good code lately? Code that makes you feel like exploring where you can progressively zoom into the topics you need to understand, and glance through the topics you already understand? Or, the code that lets you skip portions of unneeded code so that you can get to the portion you need to read?

Well, if you have not read such code, you should read code. Reading code is one of the best ways to write code, just like prose. Some samples of code I liked over the years, for example, include ghostscript (A software postscript interpreter), Scheme interpreter from Rice University, several code packages from GNU software.

One of the most important skills that are required to write good programs is structural decomposition. [2]

7.5.1 Dizajn u vidu tokovnika

Data-flow design is concerned with designing a sequence of functional transformations that convert system inputs into the required outputs. The design is represented as data-flow diagrams. These diagrams illustrate how data flows through a system and how the output is derived from the input through a sequence of functional transformations. Data-flow diagrams are a useful and intuitive way of describing a system. They are normally understandable without special training, especially if control information is excluded. They show end-to-end processing. That is, the flow of processing from when data enters the system to

[2] Delom zasnovano na pogljavlju "Function-oriented design"knjige Ian Sommerville: Software Engineering

where it leaves the system can be traced. Data-flow design is an integral part of a number of design methods and most CASE tools support data-flow diagram creation. Different methods may use different icons to represent data-flow diagram entities but their meanings are similar. The notation which I use is based on the following symbols:

- Rounded rectangles represent functions which transform inputs to outputs. The transformation name indicates its function.
- Rectangles represent data stores. Again, they should be given a descriptive name.
- Circles represent user interactions with the system that provide input or receive output.
- Arrows show the direction of data flow. Their name describes the data flowing along that path.
- The keywords and and or. These have their usual meanings as in boolean expressions. They are used to link data flows when more than one data flow may be input or output from a transformation.

Data-flow diagrams show functional transformations but do not suggest how these might be implemented. A system described in this way might be implemented as a single program using functions or procedures to implement each transformation. Alternatively, it could be implemented as a number of communicating tasks.

7.5.2 Funkcijski-orijentisan dizajn

A function-oriented design strategy relies on decomposing the system into a set of interacting functions with a centralised system state shared by these functions. Functions may also maintain local state information but only for the duration of their execution. Function-oriented design has been practised informally since programming began. Programs were decomposed into subroutines which were functional in nature. In the late 1960s and early 1970s several books were published which described top-down functional design. They specifically proposed this as a structured design strategy (Myers, 1975; Wirth, 1976; Constantine and Yourdon, 1979). These led to the development of many design methods based on functional decomposition. Function-oriented design conceals the details of an algorithm in a function but system state information is not hidden. This can cause problems because a function can change the state in a way which other functions do not expect. Changes to a function and the way in which it uses the system state may cause unanticipated changes in the behaviour of other functions. A functional approach to design is therefore most likely to be successful when the amount of system state information is minimised and information sharing is explicit. Systems whose responses depend on a

single stimulus or input and which are not affected by input histories are naturally functionally-oriented. Many transaction-processing systems and business data-processing systems fall into this class. In essence, they are concerned with record processing where the processing of one record is not dependent on any previous processing. An example of such a transaction processing system is the software which controls automatic teller machines (ATMs) which are now installed outside many banks. The service provided to a user is independent of previous services provided so can be thought of as a single transaction. In this design, the system is implemented as a continuous loop and actions are triggered when a card is input. Functions such as `Dispense_cash`, `Get_account_number`, `Order_statement`, `Order_checkbook`, etc. can be identified which implement system actions. The system state maintained by the program is minimal. The user services operate independently and do not interact with each other. An object-oriented design would be similar to this and would probably not be significantly more maintainable.

7.5.3 Strukturna dekompozicija

Traditional computer science and most if not all rely on the idea of modularfunctionalism: that structural and functional decomposition are one and the same. It is necessary first to understand the ideas of structural and functional decomposition. Structural decomposition refers to the natural decomposition into "pieces" or components of the system. For example, a cup might logically decompose into its handle and its bowl. Alternately, we can talk about the functional decomposition of the cup: it can be used to hold liquid, and it can be carried. In this case, there is a clear coupling between the structural and functional decompositions of the system: the handle admits the carrying function, while the bowl is responsible for the holdingliquid part of the job. If the bowl were to break, the handle would still work, and vice versa. Ponekad struktura i funkcionalna podela ipak nisu isto.

Once you start reading code, you realize on basic tenet: at most your brain will comprehend code spread over 25 or so lines. Anything beyond, your brain will try to break into chunks. When writing prose, will you have a paragraph stretching pages? Think of the code same way. Different programming languages offer different compositional units. The most common one is a functional or procedural decomposition. In this model, you take a large function and break it down into sub-functions. The reasons for decomposition is for two reasons:

- Reuse of the code: If properly decomposed, the parts of abstraction can be reused elsewhere. For example, suppose you refer to a patient by SSN (Social Security Number, a unique identifier in the US), then there must be code that pulls up the patient record given SSN. That code, if isolated, can be referred through proper function.

- Ease of understanding: Even if there is no reuse, sometimes, breaking the code into smaller parts makes it more readable. It lets the writer code

> at a uniform level, preferably that of the domain, hiding the details of exceptional logic and language objects into functions that do not clutter the mainline code.

Normally, people do not decompose the code as an afterthought. They write the code from top down, resulting in a code that is naturally readable, and naturally well decomposed. Here is an example.

Consider the case of patient record system. When a patient walks in for an appointment, here is what the system might do: First his SSN is entered. If there is a previous history, that is pulled up. If there no history, his information is collected and entered into the system. After that, the purpose of his visit is entered into the system. Next, the name of the doctor is entered. His insurance information may be verified and the name of the verifier may be noted. The patient is advised of his rights and that fact should be noted in the system.

Suppose, we write the entire process in one function, clubbing the database retrieval, and adding the information to the database, and validation of SSN – the system can get unreadable. We will lose the information at the domain level (as presented in the preceding paragraph). Here is a possible rendition of the code:

```
//This code is executed when the patient visits for the first time:
Get Patent Information If the patient is not there in the system, add
him to the system. Obtain the purpose of the visit Enter the name of
the provider Verify the insurance Information Obtain the proof of
advising of the rights.
```

Turning it into more čode like":

```
// This code gets executed when the patient visits for the first time
// Get the patient information
Patient = getPatient(getSSN());
// If the patient is not there in the system, add him to the system.
if (Patient == null) { createPatient(InputSystem.getSSN()); }
// Obtain the purpose of the visit
recordPurposeOfVisit(Patient);
//Enter the name of the providerenterProvidersName(Patient);
// Verify the insurance Information
verifyInsurance(Patient);
//Obtain the proof of advising of the rights.
adviseTheRights(Patient);
```

Of course, this code is not real. However, it helps you in two ways. You can retain the comments that help us show what we are trying to do. It also tells us more about the kind of objects we need. Here, we have two nouns: Patient and Visit. We also may have a linguistic object like session that deals with the current session with the patient. We also may have a glue object like `PatientRights`.

With that in mind, we will redo the code.

```
// This code gets executed when the patient visits for the first time
// Get the patient information
currentPatient = Patient.getPatient(currentSession.getSSN());
// If the patient is not there in the system, add him to the system.
if (currentPatient == null) {
currentPatient=createPatient(currentSession); }
// Obtain the purpose of the visit
currentVisit = new Visit(currentPatient);
currentVisit.recordPurpose(currentSession.getPurpose());
//Enter the name of the provider
currentVisit.recordProvider(currentSession.getProvider());
//Verify if the insurer is still current.
currentPatient.verifyInsurer(currentSession);
// Advise the rights, and record that information.
currentPatient.adviseRights(PatientRights.getRights(currentPatient),
                            currentSession.getOperator());
```

Depending on the objects you may have and the complexity of the code, this can result in a few hundreds of lines. For example, the simple function getSSN, may become something like:

```
// Get the Social Security Number from the user and validate
SSN = getInput("Enter SSN:"); if (isValid(SSN)) return SSN; for (int
i = 0; i < MAX_NUMBER_OF_TRIES, i++) {
    SSN = getInput("Invalid SSN. Enter SSN:");
    if (isValid(SSN)) return SSN;
} throw InvalidInput("Invalid SSN");
```

While this code is certainly needed, if we placed it in the main code, it would clutter up the rest of the code. By separating it out, we have achieved our two purposes: Reuse (in case we need to get the SSN from the user somewhere else), and Readability. Notice that validating SSN is pushed into some other routine to separate such non-domain logic from a domain specific constraints (max number of times you can ask for SSN) achieves the same goals. Here are the rules of thumb with this approach:

Have functions that are smaller than 25 lines. Best way to create new functions is to abstract some part of the problem (domain, technical, or linguistic) and provide a function for that. It always should be possible to get such an abstraction going. Use the English description you write as you refine the code as comments. It tells you the domain level activities that tell the reader what you are doing. For complex logic functions, include the algorithm before the function body and after the comment section. If the function is large, break down into blocks, where each block is doing some unique activity. Use one line comment describe that activity. Use appropriate formatting scheme to cut down on excessive lines. For examples, ornate commenting scheme is not good. Placing empty lines that does not indicate some semantic separation to the reader is not good. Also, use K&R scheme of indenting to maximize the information

to lines ratio.

7.5.4 Strukturalni model sistema

As well as a data-flow model of a system, it is also useful to develop a structural system model. This structural model shows how a function is realised by a number of other functions which it calls. Structure charts are a graphical way to represent this decomposition hierarchy. Like data-flow diagrams, they are dynamic rather than static system models. They show how one function calls others. They do not show the static block structure of a function or procedure. A function is represented on a structure chart as a rectangle. The hierarchy is displayed by linking rectangles with lines. Inputs and outputs (which may be implemented either as parameters or shared variables) are indicated with annotated arrows. An arrow entering a box implies input, leaving a box implies output. Data stores are shown as rounded rectangles and user inputs as circles. Converting a data-flow diagram to a structure chart is not a mechanical process. It requires designer insight and creativity. However, there are several rules of thumb which may be applied to help designers assess if their decomposition is likely to be a reasonable one:

- Many systems, particularly business systems for which functional design is most appropriate, can be considered as three-stage systems. These are input some data, perhaps with validation and checking, process the data then output the data, perhaps in the form of a report or perhaps to some other file. A master file may also be updated. The first-level structure chart may therefore have 3 or 4 functions corresponding to input, process, master-file update and output.

- If data validation is required, functions to implement these should be subordinate to an input function. Output formatting, printing and writing to disk or tape should be subordinate to an output function.

- The role of functions near the top of the structural hierarchy may be to control and coordinate a set of lower-level functions.

- The objective of the design process is to have loosely coupled, highly cohesive components. Functions should therefore do one thing and one thing only.

- Each node in the structure chart should have between two and seven subordinates. If there is only a single subordinate, this implies that the unit represented by that node may have a low degree of cohesion. The component may not be single function. A single subordinate means that another function has been factored out. If a node has too many subordinates, this may mean that the design has been developed to too low a level at that stage.

Three process steps, which follow these guidelines, can be identified for the transformation process from data-flow diagram to structure chart:

- Identify system processing transformations These are the transformations in the diagram which are responsible for central processing functions. They are not concerned with any input or output functions such as reading or writing data, data validation or filtering or output formatting. Group these transformations under a single function at the first-level in the structure chart.
- Identify input transformations These are concerned with reading data, checking it, removing duplicates, etc. These should also be grouped under a single function at the first-level in the structure chart.
- Identify output transformations These are transformations which prepare and format output or write it to the users screen or other device. However, there are several rules of thumb that can be applied to help designers in this process.

Deo III

Principi programskih jezika

Glava 8

Programski jezici i paradigme

8.1 Istorijat razvoja viših programskih jezika

Programiranje prvih elektronskih računara 1940-tih godina vršeno je isključivo korišćenjem mašinskog i asemblemblerskog jezika. Ipak, već u to vreme, postojala je ideja o razvoju apstraktnijih programskih jezika koji bi automatski bili prevođeni na mašinski jezik. Tako je 1948. Konrad Cuze (nem. Konrad Zuse) objavio rad o višem programskom jeziku *Plankalkül*, međutim prevodilac za ovaj jezik nikada nije implementiran.

Najznačajni jezici nastali 1950-tih godina su *Fortran*, *Lisp* i *Cobol*.

Prvi viši programski jezik koji je stekao širok krug korisnika je programski jezik *Fortran*, nastao u periodu 1953-1957 godine. Vođa tima u okviru kompanije IBM koji je projektovao ovaj jezik i implementirao prvi prevodilac za njega, bio je Džon Bekus (engl. John Backus). Ime Fortran je skraćenica za *IBM mathematical FORmula TRANslator system*. Jezik se i danas uspešno koristi (naravno, uz velike izmene u odnosu na prvobitne verzije) i namenjen je, pre svega, za numerička i naučna izračunavanja.

Programski jezik *Lisp* razvio je Džon Makarti (engl. John McCarthy) 1958. godine na univerzitetu MIT. Lisp se smatra jezikom veštačke inteligencije. Zasnovan je na Čerčovom formalizmu λ-računa i spada u grupu funkcionalnih programskih i od njega su se dalje razvili svi savremeni funkcionalni programski jezici. Ime Lisp je nastalo od *LISt Processing* jer jezik podržava liste i drveta kao osnovne strukture podataka. Neke varijante Lisp-a se i danas koriste.

Programski jezik *Cobol* napravljen je 1959. godine zajedničkom inicijativom nekoliko vodećih firmi i ljudi sa univerziteta da se napravi jezik pogodan za poslovnu upotrebu. Ime predstavlja skraćenicu od *COmmon Business Oriented Language* što ga i definiše kao jezik koji se primenjuje za izradu poslovnih, finansijskih i administrativnih aplikacija za firme i vlade. Aplikacije programirane u Cobol-u se i danas mogu sresti, pri čemu se sam Cobol danas veoma retko

koristi.

Jedan od najuticajnijih programskih jezika je svakako programski jezik *Algol*.
Pascal
Basic
PL/I
Prolog Simula, *Smalltalk ML*
C
SQL
Ada C++ Perl
Haskell Python Java JavaScript Php C#
Najznačajniji događaji u razvoju programiranja i programskih jezika:

- *Plankalkul* (1948) Konrad Zuse, Nemački inženjer, definisao je, za svoj računar Z3, prvi programski jezik, ali jezik nije implementiran.
- FORTRAN: John Backus, 1957 (blizak matematičkoj notaciji, efikasno prevodenje na mašinski jezik, novi korisnici, paketi, prenosivost, čitljivost)
- LISP (1958) McCarthy, rad sa listama;
- COBOL
- Operativni sistemi
- Prolog (1972) Kovalski (neproceduralan jezik, deskriptivno programiranje);
- Algol 60 (60-tih g.)
- Algol W (1966): poboljšano struktuiranje podataka
- Pascal (1971): dalja poboljšanja
- Modula 2 (1983): koncept modula (Wirth)
- Oberon (1988): Wirth
- CPL (Strachy 1966) - Combined Programming Language: nije u potpunosti implementiran; uvodi koncepte
- BCPL (Richards 1969): Basic CPL - za pisanje kompilatora
- C 1972 - Dennis Ritchie - implementacioni jezik za softver vezan za operativni sistem UNIX (1973. prepisan u C)
- 1977 usavršen sistem tipova C-a
- C++ (Stroustrup 1986) - definisanje novih tipova; klase pozajmljene od Simule 67; strogo tipiziran sistem
- Jezici IV generacije - specijalne namene - za rad sa bazama podataka, raširenim tabelama, obradu teksta
- Internet - teleizracunavanje

8.2 Programske paradigme

U ovom poglavlju biće opisani neki način da se klasifikuju međusobno slični programski jezici. Važno je naglasiti da je veoma često da programski jezik deli osobine nekoliko različitih klasa i da granice ovakvih klasifikacija nikada ne mogu biti oštre.

U odnosu na način rešavanja problema programski jezici mogu biti:

Proceduralni - Proceduralni (ili imperativni) programski jezici zahtevaju od programera da precizno specifikuje algoritam kojim se problem rešava navodeći naredbe oblika: ,,Uradi ovo, uradi ono". Značajni proceduralni programski jezici su npr. C, Pascal, Java, ...

Deklarativni - Deklarativni programski jezici 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 npr. 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 (npr. promenljive, funkcije, objekti, ograničenja) i koracima od kojih se sastoje izračunavanja (dodele, sračunavanja vrednosti izraza, tokovi podataka, itd.). Najznačajnije programske paradigme su:

Imperativni jezici - Jezici imperativne paradigme posmatraju 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.

Najznačajniji imperativni programski jezici su Fortran, Algol i njegovi naslednici, Pascal, C, Basic, ...

Imperativni jezici, uz objektno orijentisane jezike, se najčešće koriste u industrijskom sistemskom i aplikativnom programiranju.

Imeperativni jezici su obično izrazito proceduralni.

Funkcionalni jezici - Jezici funkcionalne paradigme posmatraju izračunavanje kao proces izračunavanja matematičkih funkcija i izbegava koncept stanja i promenljive. 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 se mogu smatrati nadogradnjama λ-računa.

U praksi, razlika između matematičkog pojma funkcije i pojma funkcija u imperativnim programskim jezicima je da imperativne funkcije mogu imati *bočne efekte* odnosno da uz izračunavanje rezultata menjaju 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 prilično drugačijim od matematičkih funkcija koje uvek predstavljaju isti rezultat za iste vrednosti ulaznih argumenata. Pošto funkcionalni programski jezici zabranjuju koncept promenljivih i stanja programa, u njima nisu mogući bočni efekti i one poseduju svojstvo *referencijalne transparentnosti* koje garantuje da funkcija uvek daje isti rezultat za iste ulazne argumente. Ovim je mnogo jednostavnije razumeti i predvideti ponašanje programa i eliminišu se mnoge greške koje nastaju tokom razvoja imperativnih programa, što je jedna od ključnih motivacija za razvoj funkcionalnih programskih jezika. Takođe, referencijalna transparentnost omogućava mnoge automatske optimizacije izračunavanja što današnje funkcionalne jezike čini veoma efikasnim. Jedna od značajnih optimizacija je mogućnost automatske paralelizacije i izvršavanja na višeprocesorskim sistemima.

Važnost funkcionalnih programskih jezika je obično bolje shvaćena u akademskom nego u industrijskom okruženju. Korektnost funkcionalnih programa se dokazuje mnogo lakše nego u imperativnom slučaju, tako da se funkcionalno programiranje koristi za izgradnju bezbednosno kritičnih sistema (npr. softver u nuklearnim elektranama, svemirskim programima, avionima...). Najpoznatiji funkcionalni programski jezici su Lisp, Scheme, ML, Haskell, ...

Logički jezici - Logički programski jezici (po najširoj definiciji ove paradigme) su oni jezici koji 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 - Objektno-orijentisani jezici 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 OO jezici su C++, Java, Eifell, Simula, SmallTalk, ...

Glava 9

Uvod u prevođenje programskih jezika

9.1 Implementacija programskih jezika

- Various strategies depend on how much preprocessing is done before a program can be run, and how CPU-specific the program is.
- Interpreters run a program "as is"with little or no pre-processing, but no changes need to be made to run on a different platform.
- Compilers take time to do extensive preprocessing, but will run a program generally 2- to 20- times faster.
- Some newer languages use a combination of compiler and interpreter to get many of the benefits of each.
- Examples are Java and Microsofts .NET, which compile into a virtual assembly language (while being optimized), which can then be interpreted on any computer.
- Other languages (such as Basic or Lisp) have both compilers and interpreters written for them.
- Recently, "Just-in-Timečompilers are becoming more common - compile code only when its used!

9.2 Kratka istorija razvoja kompilatora

- 1953 IBM develops the 701 EDPM (Electronic Data Processing Machine), the first general purpose computer, built as a đefense calculator"in the Korean War. No high-level languages were available, so all programming was done in assembly

- As expensive as these early computers were, most of the money companies spent was for software development, due to the complexities of assembly.
- In 1953, John Backus came up with the idea of špeed coding", and developed the first interpreter. Unfortunately, this was 10-20 times slower than programs written in assembly. He was sure he could do better.
- In 1954, Backus and his team released a research paper titled "Preliminary Report, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN."
- The initial release of FORTRAN I was in 1956, totaling 25,000 lines of assembly code. Compiled programs ran almost as fast as handwritten assembly! Projects that had taken two weeks to write now took only 2 hours. By 1958 more than half of all software was written in FORTRAN.

9.3 Moderni kompilatori

- Compilers have not changed a great deal since the days of Backus. They still consist of two main components:
- The front-end reads in the program in the source languages, makes sense of it, and stores it in an internal representation...
- ...and the back-end, which converts the internal representation into the target language, perhaps with optimizations. The target language used is typically an assembly language, but it is often easier to use a more established, higher-level language.

9.4 Struktura kompilatora

Source Language

⇓

Front End

⇓

Intermediate Code

⇓

Back End

⇓

Target Language

Front End:

- Lexical Analyzer
- Syntax Analyzer
- Semantic Analyzer
- Intermetiate Code Generator

Front End:

- Code Optimizer
- Target Code Generator

Overall structure:

Source Language

⇓

Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Intermetiate Code Generator

⇓

Intermediate Code

⇓

Code Optimizer
Target Code Generator

⇓

Target Language

9.5 Leksička analiza

Lexical analysis is the processing of an input sequence of characters (such as the source code of a computer program) to produce, as output, a sequence of symbols called "lexical tokens", or just "tokens". For instance, lexical analyzers (*lexers*) for many programming languages will convert the character sequence `123abc` into the two tokens `123` and `abc`. The purpose of producing these tokens is usually to forward them as input to another program, such as a parser.

Consider the code:

```
if (i==j);
    z=1;
else;
    z=0;
endif;
```

This is really nothing more than a string of characters:
`if_(i==j);\n\tz=1;\nelse;\n\tz=0;\nendif;`
During lexical analysis phase we must divide this string into meaningful sub-strings.

The output of our lexical analysis phase is a streams of *tokens*. A token is a syntactic category. In English this would be types of words or punctuation, such as a "noun", "verb", "adjective"or "end-mark". In a program, this could be an "identifier", a "floating-point number", a "math symbol", a "keyword", etc

A sub-string that represents an instance of a token is called a *lexeme*.

For the token `IDENTIFIER`, possible lexemes are `a`, `b`,...

The class of all possible lexemes in a token is described by the use of a pattern. The pattern to describe the token `IDENTIFIER` is a string of letters, numbers, or underscores, beginning with a non-number. Patterns are typically described using regular expressions.

For lexical analysis:

- Regular expressions describe tokens
- Finite automata are mechanisms to generate tokens from input stream.

`lex` is a program that generates lexers in the C programming language.

9.6 Sintaksna analiza

Syntax analysis is a process in compilers that recognizes the structure of programming languages. It is also known as parsing. After lexical analysis, it is much easier to write and use a parser, as the language is far simpler.

Context-free grammar is usually used for describing the structure of languages and BNF notation is typical to define that grammar. Grammatical tokens include numerical constants and literal strings and control structures such as assignments, conditions and loops.

Programs or code that do parsing are called *parsers*.

Goal: we must determine if the input token stream satisfies the syntax of the program

What do we need to do this?

- An expressive way to describe the syntax
- A mechanism that determines if the input token stream satisfies the syntax description

Regular expressions don't have enough power to express any non-trivial syntax of a programming language. Syntax of programming languages is often described by context-free grammars.

Parse Tree:

- Internal Nodes: Nonterminals
- Leaves: Terminals
- Edges:
  - From Nonterminal of LHS of production
  - To Nodes from RHS of production
- Captures derivation of string

```
   Expr
Expr Op Expr Int     Int 2    -   1
```

`Yacc` (yet another compiler compiler) is a program that generates parsers in the C programming language.

9.7 Primer

- Source Code:

  ```
  cur_time = start_time + cycles * 60
  ```

- Lexical Analysis:

  ```
  ID(1) ASSIGN ID(2) ADD ID(3) MULT INT(60)
  ```

- Syntax Analysis:

  ```
      ASSIGN
  ID(1)      ADD
        ID(2)   MULT
             ID(3) INT(60)
  ```

- Sematic Analysis:

  ```
      ASSIGN
  ID(1)      ADD
        ID(2)   MULT
             ID(3) int2real
                   INT(60)
  ```

- Intermediate Code:

```
temp1 = int2real(60) temp2 = id3 * temp1 temp3 = id2 + temp2 id1 =
temp3
```

- Optimized Code :

  Step 1:

  ```
  temp1 = 60.0 temp2 = id3 * temp1 temp3 = id2 + temp2 id1 = temp3
  ```

  Step 2:

  ```
  temp2 = id3 * 60.0 temp3 = id2 + temp2 id1 = temp3
  ```

  Step 3:

  ```
  temp2 = id3 * 60.0 id1 = id2 + temp2
  ```

  Optimized Code:

  ```
  temp1 = id3 * 60.0 id1 = id2 + temp1
  ```

- Target Code Generator

  Target Code:

  ```
  MOVF id3, R2 MULF #60.0, R2 MOVF id2, R1 ADDF R2, R1 MOVF R1, id1
  ```

Kako bi bilo moguće prevođenje programa u odgovarajuće programe na mašinskom jeziku nekog konkretnog računara, neophodno je precizno definisati sintaksu i semantiku programskih jezika. Podsetimo se, pitanjima ispravnosti programa bavi se *sintaksa programskih jezika* (i njena podoblast *leksika programskih jezika*). Leksika se bavi opisivanjem osnovnim gradivnim elementima jezika, a sintaksa načinima za kombinovanje tih osnovnih elemenata u ispravne jezičke konstrukcije. Pitanjem značenja programa bavi *semantika programskih jezika.*

9.8 Teorija formalnih jezika

Naglašeno je da izgradnja jezičkih procesora zahteva matematički precizne definicije ispravnih konstrukcija programskih jezika. Oblast matematike koja se bavi formalnim definisanjem pojma jezika i formalnim opisima leksike i sintakse programskih jezika naziva se *formalna teorija jezika (engl. formal language theory).* U nastavku će biti ukratko uvedeni osnovni pojmovi ove grane računarstva i biće date smernice na njihovu primenu u okviru opisa programskih jezika.

Slovo, azbuka, reč, jezik

Azbuke. Za zapis kako prirodnih tako i veštačkih jezika koriste se unapred fiksirani skupovi simbola (slova). Skup simbola koji se koriste za zapis nekog jezika naziva se *azbuka* tog jezika.

Definicija 9.1. Azbuka (alfabet) *je konačan, neprazan skup simbola (slova) za koje pretpostavljamo da nijedan od njih ne sadrži neki drugi. Azbuka se obično obeležava sa* Σ.

Reči. Reči nad nekom azbukom su konačni nizovi simbola te azbuke. Specijalno, *prazna reč* je je reč koja ne sadrži nijedan simbol i označavamo je sa ε. Definišimo sada i formalno skup Σ^* svih reči nad azbukom Σ.

Definicija 9.2. Σ^* *je najmanji skup za koji važi:*

1. $\varepsilon \in \Sigma^*$,
2. *za svako slovo* $a \in \Sigma$ *i svaku reč* $x \in \Sigma^*$ *važi da je* $ax \in \Sigma^*$.

Elementi skupa Σ^* *nazivaju se* reči nad azbukom Σ.
Skup $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ *je skup svih* nepraznih reči nad azbukom Σ.

Dakle, svaka reč je ili prazna reč ε ili je oblika ax za neko slovo a i reč x.

Primetimo da je skup Σ^* beskonačan, ali prebrojiv (ima istu kardinalnost kao skup prirodnih brojeva, tj. svi njegovi elementi mogu se poredjati u niz).

Osnovna operacija koja se može izvoditi nad rečima je operacija *dopisivanja (konkatenacije)*. Ova operacija se obično označava simbolom $\cdot$ ili se, još češće, ovaj simbol izostavlja.

Operaciju dopisivanja, moguće je uvesti narednom (rekurzivnom) definicijom:

Definicija 9.3. *Neka su* x *i* y *reči iz* Σ^*. Konkatenacija *reči* x *i* y *je reč* $x \cdot y$ *iz* Σ^* *takva da važi:*

1. *ako je* $x = \varepsilon$, *tada je* $x \cdot y = y$,
2. *ako je* $x = ax'$, *tada je* $x \cdot y = a(x' \cdot y)$.

Skup svih reči Σ^* u odnosu na operaciju dopisivanja čini tzv. *slobodni monoid*. U ovom monoidu važi i zakon skraćivanja (kancelacije).

Teorema 9.1. *Svojstva konkatenacije (dopisivanja):*

1. *Konkatenacija je asocijativna:*

$$(xy)z = x(yz)$$

2. *Prazna reč* ε *je jedinični element za konkatenaciju:*

$$\varepsilon x = x\varepsilon = x$$

3. Za konkatenaciju je dozvoljeno "skraćivanje":

$$xz = yz \;\Rightarrow\; x = y$$

$$zx = zy \;\Rightarrow\; x = y$$

Jezici. Jezici su proizvoljni skupovi reči.

Definicija 9.4. *Neka je* Σ *neka azbuka. Svaki podskup skupa svih reči* Σ^* *nad* Σ *zovemo* jezik *nad* Σ*.*

Nad jezicima moguće je primenjivati određene operacije.

Skupovne operacije - pošto su jezici skupovi reči, nad njima je moguće izvoditi uobičajene skupovne operacije: unija ($\cup$), presek ($\cap$), razlika ($\setminus$), komplement u odnosu na Σ^* ($'$), ...

Proizvod jezika - Operacija dopisivanja reči se prirodno uzdiže na nivo jezika.

Definicija 9.5. *Neka su* L_1 *i* L_2 *jezici nad azbukom* Σ*.* Proizvod jezika $L_1 \cdot L_2$ *je jezik nad azbukom* Σ *definisan na sledeći način:*

$$L_1 \cdot L_2 = \{x \cdot y \mid x \in L_1, y \in L_2\}$$

Proizvod jezika podrazumeva nadovezivanje proizvoljne reči prvog jezika sa proizvoljnom reči drugog jezika.

Primer 9.1. *Neka je*

$$L_1 = \{\mathrm{BG},\ \mathrm{NS},\ \mathrm{NI}\},$$

$$L_2 = \{000,\ 001,\ \ldots,\ 998,\ 999\}.$$

Tada je

$$L_1 \cdot L_2 = \{\mathrm{BG000},\ \ldots,\ \mathrm{BG999},\ \mathrm{NS000},\ \ldots,\ \mathrm{NS999},\ \mathrm{NI000},\ \ldots, \mathrm{NI999}\}.$$

S obzirom da je konkatenacija asocijativna, korišćenjem proizvoda, na uobičajeni način se može uvesti i stepen jezika i to narednom induktivnom definicijom.

Definicija 9.6. *Neka je* L *jezik nad azbukom* Σ*.* Stepeni jezika L*, definisani su sa:*

$$L^0 = \{\varepsilon\},$$

$$L^{i+1} = L \cdot L^i, \quad i \geq 0$$

Primetimo da n-ti stepen jezika podrazumeva nadovezivanje n proizvoljnih reči jezika L.

Klinijevo zatvorenje - Veoma često se razmatraju jezici koji se sastoje od reči nastalih dopisivanjem proizvoljnog broja reči nekog jezika L. Pošto i-ti stepen jezika L predstavlja nadovezivanje proizvoljnih i reči jezika L, od interesa je uniju i-tih stepena za sve brojeve i, tj. razmatrati *Klinijevo zatvorenje (iteraciju)* jezika L

$$L^* = \bigcup_{i \geq 0} L^i$$

Ipak, Klinijevo zatvorenje uvodimo narednom definicijom koja ne koristi pojmove proizvoda, stepena i unije.

Definicija 9.7. *Neka je L jezik nad azbukom Σ. Klinijevo zatvorenje L^* jezika L je najmanji jezik nad azbukom Σ koji sadrži ε i sve reči jezika L i zatvoren je u odnosu na konkatenaciju reči.*

Primer 9.2. *Neka je jezik $L = \{ab, ba\}$. Tada je*

$$L^* = \{\varepsilon, ab, ba, abab, abba, baab, baba, ababab, \ldots\}$$

Regularni izrazi

U slučaju da je jezik konačan, on se računaru može specifikovati nabrajanjem njegovih elemenata. Sa druge strane, beskonačni jezici zahtevaju druge načine specifikacije koje će biti računarski čitljive. Neki složeni jezici se mogu izgraditi od jednostavnijih, konačnih jezika, ili čak jednočlanih jezika korišćenjem navedenih operacija nad jezicima.

Definicija 9.8. Klasa regularnih jezika Reg_Σ *nad azbukom Σ je najmanji skup koji zadovoljava:*

1. $\{\varepsilon\} \in Reg_\Sigma$,
2. *za svako slovo $a \in \Sigma$, važi $\{a\} \in Reg_\Sigma$,*
3. *za svako L_1 i L_2 iz Reg_Σ, važi $L_1 \cup L_2 \in Reg_\Sigma$,*
4. *za svako L_1 i L_2 iz Reg_Σ, važi $L_1 \cdot L_2 \in Reg_\Sigma$,*
5. *za svako L iz Reg_Σ, važi $L^* \in Reg_\Sigma$.*

Elemente skupa Reg_Σ nazivamo regularnim jezicima.

Iako je definicija klase regularnih jezika zahtevala samo zatvorenost u odnosu na uniju, moguće je pokazati da je ova klasa zatvorena i u odnosu na ostale skupovne operacije nad jezicima.

Primer 9.3. *Identifikatori u programskom jezikom C mogu da se sastoje od slova (bilo malih, bilo velikih), cifara i podvlaka _, pri čemu ne smeju da počnu*

cifrom. Neka je $L_S = \{a, \ldots, z, A, \ldots, Z\}$, $L_C = \{0, \ldots, 9\}$ *i* $L_\_ = \{\_\}$. *Tada za jezik svih identifikatora* L_i *važi:*

$$L_I = (L_s \cup L_\_) \cdot (L_S \cup L_C \cup L_\_)^*.$$

Pošto su L_S *i* L_C *konačni jezici, oni se mogu dobiti operacijom unije krenuvši od jednočlanih skupova koji sadrže slova i cifre. Dakle, jezik* L_i *je regularan jezik.*

Kako bi se regularni jezici mogli specifikovati računaru, koriste se regularni izrazi koji su već ranije bili opisani. Iako se regularnim izrazima predstavljaju regularni jezici, obično se u zapis regularnih izraza uvode određene skraćenice kojim se olakšava rad, pri čemu se suštinski ne proširuje klasa dopuštenih jezika. Ovakvi regularni izrazi se nazivaju i *prošireni regularni izrazi.*

Formalne gramatike

Definicija 9.9. *Neka su* Σ *i* N *dve disjunktne azbuke i neka je* S *slovo azbuke* N *(*$S \in N$*). Formalna gramatika (gramatika Čomskog) je uredjena četvorka*

$$G = (N, \Sigma, P, S),$$

gde je P *(konačan) skup* pravila izvodjenja *(*pravila zamene *ili* produkcionih pravila*) oblika:*

$$\alpha \rightarrow_G \beta\ ,$$

pri čemu je $\alpha \in (N \cup \Sigma)^* N (N \cup \Sigma)^*$, $\beta \in (N \cup \Sigma)^*$.

Skup Σ *zovemo* skupom završnih (terminalnih) simbola*, skup* N *zovemo* skupom nezavršnih (neterminalnih) simbola*, a slovo* S *zovemo* početnim (ili polaznim) simbolom *ili* aksiomom gramatike*.*

Pravila izvodjenja $\alpha \rightarrow_G \beta_1$, $\alpha \rightarrow_G \beta_2$, ..., $\alpha \rightarrow_G \beta_n$, kraće zapisujemo na sledeći način:

$$\alpha \ \rightarrow_G \ \beta_1 \,|\, \beta_2 \,|\, \ldots \,|\, \beta_n \ .$$

Definicija 9.10. *Neka su reči* x *i* y *reči nad azbukom* $N \cup \Sigma$*. Kažemo da da je reč* y *neposredna posledica reči* x *i da je reč* y *neposredno izvodiva iz reči* x*, u gramatici* $G = (N, \Sigma, S, P)$ *i pišemo*

$$x \Rightarrow_G y,$$

ako i samo ako postoje reči $\phi, \psi, \alpha, \beta$ *takve da je* $x = \phi\alpha\psi$, $y = \phi\beta\psi$ *i* $\alpha \rightarrow_G \beta \in P$*.*

Definicija 9.11. *Relacija* $\Rightarrow_G^k$ *je* k*-ti stepen relacije* $\Rightarrow_G$*. Dakle, ako važi* $x \Rightarrow_G x_1$, $x_1 \Rightarrow_G x_2$, ..., $x_{k-1} \Rightarrow_G y$*, onda pišemo kraće* $x \Rightarrow_G^k y$*, i kažemo* reč y se izvodi u k koraka iz reči x*. Ovaj lanac izvođenja nekada kraće zapisujemo sa* $x \Rightarrow_G x_1 \Rightarrow_G x_2 \Rightarrow_G \ldots \Rightarrow_G x_{k-1} \Rightarrow_G y$*.*

Definicija 9.12. *Tranzitivno zatvorenje relacije* $\Rightarrow_G$ *označavamo sa* $\Rightarrow_G^+$*, a tranzitivno i refleksivno sa* $\Rightarrow_G^*$*. Vezu* $x \Rightarrow_G^* y$ *ili* $x \Rightarrow_G^+ y$ *čitamo:* iz y se posredno izvodi x *ili* reč y je posredna posledica reči x *ili* iz x posredno sledi y*.*

Definicija 9.13. *Jezik gramatike $G = (N, \Sigma, P, S)$, u oznaci $L(G)$ definišemo na sledeći način:*

$$L(G) = \{x \mid x \in \Sigma^* \ \wedge \ S \Rightarrow_G^+ x\} \ .$$

Primetimo da je formalna gramatika konačan opis potencijalno beskonačnog jezika. Ako je jezik generisan formalnom gramatikom beskonačan, onda je on prebrojiv.

Primer 9.4. *Neka je $\Sigma = \{a, b\}$, $N = \{S\}$ i $P = \{S \to \varepsilon \mid aSb\}$. Tada važi*

$$S \Rightarrow_G aSB \Rightarrow_G aaSbb \Rightarrow_G aaaSbbb \Rightarrow_G aaabbb.$$

Dakle, važi da $S \Rightarrow_G^+ aaabbb$, pa pošto je $aaabbb \in \Sigma^*$, reč $aaabbb$ pripada jeziku gramatike $G = (\Sigma, N, S, P)$. Slično, svaka reč oblika $a^n b^n$ pripada jeziku $L(G)$. Važi i obratno, da je svaka reč jezika $L(G)$ oblika $a^n b^n$ tako da je jezik gramatike G jezik $\{a^n b^n, n \geq 0\}$. Podsetimo se da smo za ovaj jezik napomenuli da nije regularan.

Klasifikacija Čomskog.

Definicija 9.14. *Gramatika $G = (N, \Sigma, P, S)$ je*

1° Desno linearna (DL) *ako su sva njena pravila oblika*

$$A \to w \quad ili \quad A \to wB \quad ,$$

gde je $A, B \in N$ i $w \in \Sigma^*$.

2° Levo linearna (LL) *ako su sva njena pravila oblika*

$$A \to w \quad ili \quad A \to Bw \quad ,$$

gde je $A, B \in N$ i $w \in \Sigma^*$.

3° Kontekst slobodna (KS) *ako su sva njena pravila oblika*

$$A \to \alpha \quad ,$$

gde je $A \in N$ i $\alpha \in (N \cup \Sigma)^*$.

4° Kontekst zavisna (KZ) *ako su sva njena pravila oblika*

$$\alpha \to \beta \quad ,$$

gde je $\alpha, \beta \in (N \cup \Sigma)^*$ i $|\alpha| \leq |\beta|$.

Primetimo da su sve desno (i levo) linearne gramatike kontekst slobodne. Kontekst slobodna gramatika je kontekst zavisna ukoliko ne sadrži nijedno $e-$pravilo (za pravilo $A \to e$ ne važi $1 = |A| \leq |e| = 0$).

Za jezik L se kaže da je desno linearan, odnosno kontekst slobodan, odnosno kontekst zavisan ako postoji gramatika odgovarajućeg tipa koja ga generiše, tj. $L = L(G)$.

Klase levo linearnih, desno linearnih i regularnih jezika se poklapaju.

9.9 Načini opisa leksike i sintakse programskih jezika

Kako bi se izvršila standardizacija i olakšala izgradnja jezičkih procesora potrebno je precizno definisati šta su ispravne leksičke, odnosno sintaksne konstrukcije programskog jezika. Opisi na govornom jeziku, iako mogući, obično nisu dovoljno precizni i potrebo je korišćenje preciznijih formalizama. Ovi formalizmi se nazivaju *metajezici* dok se jezik čija se sintaksa opisuje korišćenjem metajezika naziva *objektni jezik*. Meta jezici obično rezervišu neke simbole kao specijalne i imaju svoju posebnu sintaksu. Meta jezici se obično konstruišu tako da se za njihov zapis koriste ASCII karakteri i koji se može lako zadati računaru.

U praksi se kao metajezici obično koriste *regularni izrazi*, *BNF (Bekus-Naurova forma)*, *EBNF (proširena Bekus-Naurova forma)* i *sintaksni dijagrami*. BNF je pogodna notacija za zapis kontekstno slobodnih gramatika, EBNF proširuje BNF operacijama regularnih izraza čime se dobija pogodniji zapis, dok sintaksni dijagrami predstavljaju slikovni metajezik za predstavljanje sintakse. Dok se BNF može veoma jednostavno objasniti, precizna definicija EBNF zahteva malo više rada i ona je data kroz ISO 14977 standard.

Regularni izrazi Za opis leksičkih konstrukcija pogodno je koristiti 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. Npr. klasa `[0-9]` označava cifru.

alternacija - navodi se sa `|` i označava alternativne mogućnosti. Npr `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 9.5. *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]*`

Kontekstno slobodne gramatike Formalizam regularnih izraza je obično dovoljan da se opišu leksički elementi programskog jezika (npr. 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 aritmetičkih 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 kao govorni jezik kao meta jezik u okviru kojega se opisuju ispravne konstrukcije, dok se u slučaju programskih jezika, koriste znatno precizniji i formalniji opisi. Za opis sintaksnih 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 tzv. pomoćni simboli (neterminali), dok se sa desne strane nalaze niske u kojima mogu da se javljaju bilo pomoćni simboli bilo tzv. završni simboli (terminali). Svakom pomoćnom simbolu pridružena je neka sintaksna kategorija. Jedan od pomoćnih simbola se smatra istaknutim, naziva se početnim simbolom (ili aksiomom). Niska je opisana gramatikom ako je moguće dobiti krenuvši od početnog simbola, zamenjujući u svakom koraku pomoćne simbole desnim stranama pravila.

Primer 9.6. *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):*

$$
\begin{array}{rcl}
I & \to & XZ \\
X & \to & S \mid P \\
Z & \to & YZ \mid \varepsilon \\
Y & \to & S \mid P \mid C \\
S & \to & \texttt{a} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \ldots \mid \texttt{Z} \\
P & \to & \texttt{\_} \\
C & \to & \texttt{0} \mid \ldots \mid \texttt{9}
\end{array}
$$

Identifikator `x_1` *je moguće izvesti kao*

$I \Rightarrow XZ \Rightarrow SZ \Rightarrow \texttt{x}Z \Rightarrow \texttt{x}YZ \Rightarrow \texttt{x}PZ \Rightarrow \texttt{x\_}Z \Rightarrow \texttt{x\_}YZ \Rightarrow \texttt{x\_}CZ \Rightarrow \texttt{x\_1}Z \Rightarrow \texttt{x\_1}.$

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 9.7. *Neka je gramatika određena skupom pravila:*

$$\begin{array}{rcl} E & \to & E+E \\ & | & E*E \\ & | & (E) \\ & | & a. \end{array}$$

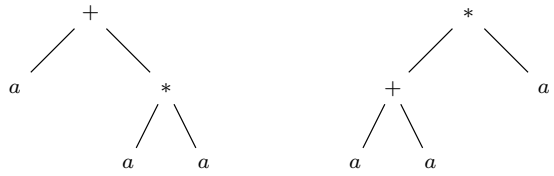
Ova gramatika opisuje ispravne aritmetičke izraze u kojima su dopuštene operacije sabiranja i množenja. Npr. izraz $a+a*a$ se može izvesti na sledeći način:

$$E \Rightarrow E+E \Rightarrow a+E \Rightarrow a+E*E \Rightarrow a+a*E \Rightarrow a+a*a.$$

Međutim, isti izraz je moguće izvesti i kao

$$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow a+E*E \Rightarrow a+a*E \Rightarrow a+a*a.$$

Prvo izvođenje odgovara levo, a drugo desno prikazanom sintaksnom stablu:



Primer 9.8. *Neka je gramatika zadata sledećim pravilima:*

$$\begin{array}{rcl} E & \to & E+T \\ & | & T \\ T & \to & T*F \\ & | & F \\ F & \to & (E) \\ & | & a \end{array}$$

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 je 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.

Konteksne gramatike čine samo jednu specijalnu vrstu formalnih gramatika. U konteksno slobodnim gramatikama sa leve strane pravila nalaze se uvek tačno jedan neterminalni simbol, a u opštem slučaju, sa leve strane pravila može se nalaziti proizvoljan niz terminalnih i neterminalnih simbola.

BNF. Metajezik pogodan za zapis pravila kontekstno slobodnih gramatika je *BNF*. 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, tzv. ,,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. sintaksne kategorije objekt-jezika. Ove promenljive odgovaraju pomoćnim (neterminalnim) simbolima formalnih gramatika. Na primer, u programskom jeziku, sintaksne kategorije su `<program>`, `<ceo broj>`, `<cifra>`, `<znak broja>`, `<identifikator>`, itd. U prirodnom jeziku, sintaskne kategorije su `<rec>`, `<recenica>`, `<interpunkcijski znak>`, itd. Metalingvističke promenljive ne pripadaju objekt jeziku. U nekim knjigama se umesto uglastih zagrada `<` i `>` metalingvističke promenljive označavaju korišćenjem masnih slova.

Metalingvističke konstante su simboli objekt jezika. To su, na primer, `0`, `1`, `2`, `+`, `-`, ali i rezervisane reči programskog jezika, npr. `if`, `then`, `begin`, `for`, itd.

Dakle, uglaste zagrade razlikuju neterminalne simbole tj. imena sintaksnih kategorija od terminalnih simbola objektnog jezika koji se navode tačno onako kakvi su u objekt 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 9.9. *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 9.10. *Gramatika aritmetičkih izraza može u BNF da se zapiše kao:*

```
<izraz> ::= <izraz> + <term> | <term> <term> ::= <term> * <faktor> |
<faktor> <faktor> := ( <izraz> ) | <ceo broj>
```

EBNF, ISO 149777 *EBNF*, je skraćenica od ,,Extend Backus-Naur Form" tj. ,,Proširena Bekus-Naurova forma". Ovaj formalizam ne omogućava da se definiše bilo šta što nije moguće definisati korišćenjem BNF, međutim uvodi skraćene zapise koji olakšavaju zapis gramatike i čine je čitljivijom. Nakon korišćenja BNF za definisanje sintakse Algola 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 korišć enje 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. Izražajnost i BNF metajezika i EBNF metajezika jednaka je izražajnosti kontekst 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 meta-simbola EBNF.

Primer 9.11. *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 9.12. *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 9.13. *Gramatika aritmetičkih izraza može u EBNF da se zapiše kao:*

```
<izraz> ::= <term> {"+" <term>} <term> ::= <faktor> {"*" <faktor>}
<faktor> := "(" <izraz> ")" | <ceo broj>
```

Primer 9.14. *Naredba grananja programskih jezika sa opcionim pojavljivanjem* `else` *grane se može opisati sa:*

```
<if_statement> ::=  if <boolean_expression> then
                         <statement_sequence>
                    [ else
                         <statement_sequence> ]
                    end if ";"
<statement_sequence> ::= <statement> { ";" <statement> }
```

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 meta simboli 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.

Sintaksni dijagrami Sintaksni dijagrami ili sintaksni 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 sintaksnih dijagrama:

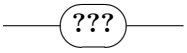
- Sintaksni dijagram za `BinarnaCifra`:

  ```
  <BinarnaCifra> ::= "0" | "1";
  ```

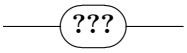
- Sintaksni dijagram za `NeoznacenCeoBroj`:

  ```
  <NeoznacenCeoBroj> ::= ( cifra ) +
  ```



- Sintaksni dijagram za `identifikator`:

```
<identifikator> ::= <slovo> { <slovo> | <cifra> }
```



9.10 Načini opisa semantike programskih jezika

Semantika pridružuje značenje sintaksno 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 **??**.

Deo IV

Socijalni aspekti informatike

Glava 10

Istorijski i društveni kontekst računarstva kao naučne discipline; društveni značaj računara i Interneta; profesionalizam, etički kodeks; autorska prava; intelektualna svojina, softverska piraterija

Technologies cannot be divorced from a social framework. Computer scientists need to be educated to understand some of the complex linkages between the social and the technical . . . computer science education should not drive a wedge between the social and the technical.

10.1 Istorijski i društveni kontekst računarstva

Historical and Societal Context of Computing
Understanding of the impact of computer technology on society.

- prve prvih računara
- prvi računari i prve primene

- epoha personalnih računara

Social Impact: The social impact of artificial intelligence is explored by examining public perceptions and the potential social implications of existing AI technology, not only from the point of view of the responsibilities of the developer, but also from the point of view of society and how it has been influenced by AI. To increase the student's awareness of the public perceptions and of the potential social implications of Artificial Intelligence.

Impact of computers–Is the technology neutral?

Društveni značaj Inteneta.

Računarstvo kao naučna disciplina.

10.2 Profesionalizam i etički kodeks

Develop a code of ethics for computing professionals by presenting codes of ethics of other professionals.

- Odgovornost profesionalca u računarstvu Responsibility of the Computer Professional: Personal and professional responsibility is the foundation for discussions of all topics in this subject area. The five areas to be covered under the responsibility of the computer professional are:
  - 1) why be ethical?
  - 2) major ethical models,
  - 3) definition of computing as a profession, and
  - 4) codes of ethics and professional responsibility for computer professionals.
- Ethical claims can and should be discussed rationally,
- Osnovi etičke analize: Basis Skills of Ethical Analysis: Five basic skills of ethical analysis that will help the computer science student to apply ethics in their technical work are:
  - 1) arguing from example, analogy, and counter-example,
  - 2) identifying stakeholders in concrete situations,
  - 3) identifying ethical issues in concrete situations,
  - 4) applying ethical codes to concrete situations, and
  - 5) identifying and evaluating alternative courses of action.
- Osnovi socijalne analize: Basic Elements of Social Analysis: Five basic elements of social analysis are:
  - 1) the social context influences the development and use of technology,

- 2) power relations are central in all social interaction,
- 3) technology embodies the values of the developers,
- 4) populations are always diverse, and
- 5) empirical data are crucial to the design and development processes.

Ethical issues for computer professionals

A wide variety of issues from privacy and security to intellectual property, democracy and freedom of expression, equal opportunities, globalisation, use and misuse and many other issues.

- Ethical and Legal Issues of Data Data Protection legislation, security and privacy of data issues.
- Ethical and Legal Issues of Information Systems Organisational and social issues of IS, their construction and their effects. Security and privacy issues also enter here, as do legal and issues of professional responsibility
- Professional Responsibility, Codes of Conduct

Mikro-situacije:

- A contract requires an experienced Web designer to develop an e-business system. You have just been awarded the contract but you failed to mention that your sole knowledge was second-hand and that you had not worked on an e-business system before. During this contract, you decide to accept a better-paid contract that requires you to start immediately.
- You are working as a systems designer for a company producing radiation monitoring equipment. You feel that there are faults in the design, but your manager rejects your concerns.
- You are an agent for a software tools producer, that has a product DoItAll. You are being paid to advise an inexperienced small business about developing their software, so you recommend that they must purchase a DoItAll tool.
- You are working as a contractor for a small company and have become aware that some of their software might be unlicensed. You are also concerned about the safeguard of their customers' data. You are currently hoping that your contract will be extended.
- You work for a software house that specialises in helping organisations to set up their E-commerce site. You are worried about the following ethical issues. You are aware that the client is inexperienced in E-commerce and that the client plans to launch its publicity for its new site in time for the Christmas market. You, as a consultant, have just been moved on to this project. You think that the project will be too late for your client's Christmas marketing campaign and that, although in line with the

client's specifications, the level of safety checks in the specification seems inadequate. You realise that the contract with the client is for a fixed price.

10.3 Rizici i pouzdanost računarskih sistema

Reliability and safety of computer systems.

Risks, Liabilities, and Bias Considerations. Risks inherent in any software application. Misuse and misunderstanding as well as developer bias. An empirical study of the evaluation of a piece of software is included to sensitize students to identifying and eliminating bias during the design phase. To sensitize the software developer to the kinds of biases present in each of us that could be passed on to the software we develop, and to point out innate risks in software applications

Liability and Privacy: The kinds of risks that are intrinsic in the development of any computer application and how these risks can be taken into account during development are discussed. Risks include software or hardware bugs, unforeseen user interactions, security risks, violations of privacy, unethical uses, and inappropriate applications of the system. Protecting data and privacy with respect to a database management system are emphasized. To provide a clear understanding of risks that a developer faces with respect to his or her software, and to emphasize the importance of protecting the data from misuse or unauthorized access

Reliability of computer systems: errors, major and minor failures.

10.4 Intelektualna svojina i autorska prava

- Intellectual property including copyright, patents, trademarks, and trade secrets.
- The Concept of Fair Use.
- Computer Regulatory Measures/Laws
- Open source, public-ware, share-ware
- Obeveze poslodavca prema zaposlenima, obaveze zaposlenog prema poslodavcu.
- Zakoni različitih zemalja, uvoz/izvoz softverskih proizvoda

10.5 Privatnost i građanske slobode

- Freedom of speech and press – censoring the Internet
- Govern and private use of personal data.

- Encryption Policy
- Computers in the workplace: effects on employment, telecommuting, employee monitoring, e-mail privacy.

10.6 Računarski kriminal

- Finansijski računarski kriminal,
- zloupotreba mejla i spam
- neovlašćeni upadi u sisteme
- distribuiranje nelegalnih sadržaja itd.

10.7 Ekonomski aspekti računarstva