

1 Sortiranje

1. Šta radi sledeći algoritam?

U zavisnosti od veličine ulaznog niza A , koliko je vreme izvršavanja datog algoritma?

Da li se može ubrzati?

```
1: function PROGRAM1( $A$ )
2:    $n \leftarrow \text{length}(A)$ 
3:    $\text{flag} \leftarrow \text{true}$ 
4:   for  $j \leftarrow 1$  to  $n - 1$  do
5:     if  $A[j] > A[j + 1]$  then
6:        $\text{flag} \leftarrow \text{false}$ 
7:     end if
8:   end for
9:   return  $\text{flag}$ 
10: end function
```

Dati algoritam proverava da li je ulazni niz sortiran.

Naime, u liniji 2 se bulovska promenljiva flag ¹ postavlja na podrazumevanu vrednost true , koja pretpostavlja da je niz sortiran.

U for petlji se za sve članove niza proverava da li je član sa manjim indeksom veći od člana sa većim indeksom. Ako je to barem jednom ispunjeno, tj. ako su posmatrani elementi u inverziji, promenljiva flag će biti postavljena na false i po završetku algoritma će ostati i biti vraćena kao false , što znači da niz nije sortiran.

Linijama koda 2, 3, 4, 5, 6, 9 dodelimo redom vreme izvršavanja $c_2, c_3, c_4, c_5, c_6, c_9$. U tabeli imamo koliko se puta koja od ovih linija izvršava

vreme izvršavanja	c_2	c_3	c_4	c_5	c_6	c_9
broj izvršavanja	1	1	n	$n - 1$	m	1

Sa obzirom da ne znamo m , broj koliko puta će se naći par susednih brojeva tako da je član sa manjim indeksom veći od člana sa većim indeksom (jedino znamo da je $1 \leq m \leq n - 1$), razlikujemo tri slučaja:

Best case	$T(n) = c_2 + c_3 + nc_4 + (n - 1)c_5 + 0c_6 + c_9 = \Theta(n)$
Worst case	$T(n) = c_2 + c_3 + nc_4 + (n - 1)c_5 + (n - 1)c_6 + c_9 = \Theta(n)$
Average case	$T(n) = c_2 + c_3 + nc_4 + (n - 1)c_5 + \lfloor \frac{n}{2} \rfloor c_6 + c_9 = \Theta(n)$

U Average² case smo pretpostavili da je u pola slučajeva upoređivani par u inverziji (da je veći broj ispred manjeg).

Dati algoritam se može ubrzati tako što će se iza linije 6 ubaciti komanda break , koja izlazi iz for petlje, jer je dovoljno samo jednom postaviti promenljivu flag na false . Ukoliko u implementaciji ne postoji komanda break , umesto for petlje treba koristiti while petlju.

¹ flag = zastavica, EN; true = tačno, EN; false = netačno, EN; length = dužina, EN

² best = najbolji, EN; worst = najgori, EN; average = srednji, EN; case = slučaj, EN

2. Doraditi algoritam iz zadatka 1 tako da se izvrši zamena ako je upoređivani par u inverziji i da se to ponavlja u repeat-until petlji od početka i da se prestane kad se dobije sortiran niz.

Da li će i zašto dobijeni algoritam ikad izaći iz repeat-until petlje?

Koliko upoređivanja i koliko zamena će dati program imati za ulaz [5,2,4,6,1,3]?

Da li se dobijeni kod može ubrzati izmenom granice u for petlji?

Da li se dobijeni kod može ubrzati ako se pamti koji deo niza je sortiran?

```

1: procedure PROGRAM2(A)
2:    $n \leftarrow \text{length}(A)$ 
3:   repeat
4:      $flag \leftarrow \text{true}$ 
5:     for  $j \leftarrow 1$  to  $n - 1$  do
6:       if  $A[j] > A[j + 1]$  then
7:          $\text{swap}(A[j], A[j + 1])$ 
8:          $flag \leftarrow \text{false}$ 
9:       end if
10:    end for
11:    until  $flag$ 
12: end procedure

```

Program u for petlji "gura" veće elemente na koje nailazi usput prema kraju. Prvi put će for petlja "odgurati" najveći element do kraja, sledeći put će odgurati drugi po veličini element, i tako dalje.

Dakle: sigurno će se repeat-until petlja završiti posle najviše n ciklusa, gde je n dužina ulaznog niza.

Posle k izvođenja repeat-until petlje, k elementa sa kraja će biti sortirani, pa se može smanjiti gornja granica for petlje: Između linije 10 i 11 (vidi PROGRAM2A) ubacimo liniju

$$n \leftarrow n - 1.$$

Program se može i dalje ubrzati ako se uvede promenljiva koja će "pamtiti" poslednji element koji je u for petlji zamenjen, jer su svi ostali iza njega sortirani (vidi PROGRAM2B).

Dobijeni algoritam PROGRAM2 se naziva BUBBLE³ SORT i posmatrana ubrzanja PROGRAM2A i PROGRAM2B su uobičajena, mada ne smanjuju red kompleksnosti algoritma.

```

procedure PROGRAM2A(A)
   $n \leftarrow \text{length}(A)$ 
  repeat
     $flag \leftarrow \text{true}$ 
    for  $j \leftarrow 1$  to  $n - 1$  do
      if  $A[j] > A[j + 1]$  then
         $\text{swap}(A[j], A[j + 1])$ 
         $flag \leftarrow \text{false}$ 
      end if
    end for
     $n \leftarrow n - 1$ 
  until  $flag$ 
end procedure

```

```

procedure PROGRAM2B(A)
   $n \leftarrow \text{length}(A)$ 
  repeat
     $newn \leftarrow 0$ 
    for  $j \leftarrow 1$  to  $n - 1$  do
      if  $A[j] > A[j + 1]$  then
         $\text{swap}(A[j], A[j + 1])$ 
         $newn \leftarrow j$ 
      end if
    end for
     $n \leftarrow newn$ 
  until  $n = 0$ 
end procedure

```

³bubble = mehurić, EN; swap = zamena, EN

U algoritmu PROGRAM2, u liniji 6 se vrši upoređivanje, u liniji 7 zamena. U algoritmima PROGRAM2A i PROGRAM2B se te linije nalaze na drugom rednom broju.

Ako se dati ulaz [5,2,4,6,1,3] propusti kroz dobijene programe, pažljivim prebrojavanjem, dobićemo rezultate:

algoritam	br. upoređivanja	br. zamena
PROGRAM2	25	9
PROGRAM2A	15	9
PROGRAM2B	14	9

3. Napisati algoritam za sortiranje umetanjem, takozvani INSERTION SORT.

Propustiti ulaz [5,2,4,6,1,3] kroz dobijeni algoritam i prebrojati broj poređenja i upisivanja elemenata u niz, odnosno u privremenu promenljivu.

Uporediti dobijeni algoritam sa algoritmom iz zadatka 2.

Algoritam polazi od drugog elementa i ide do kraja: pretpostavlja da ja niz od početka do posmatranog elementa sortiran.

Upoređuje se posmatrani element sa elementima prema početku, pomeraju se elementi sortiranog dela koji su veći od posmatranog i kad se nađe mesto, upiše se posmatrani element.

U kodu sa desne strane upoređivanje se vrši u liniji 5, a upisivanja u linijama 3, 6 i 9.

Ako se propusti ulaz [5,2,4,6,1,3] kroz algoritam, pažljivim prebrojavanjem dobijamo da se upoređivanje ključeva vršilo 12 puta, a upisivanja (linije 3, 6, i 9 zajedno) 19 puta.

Znajući da je za jednu zamenu iz BUBBLE SORT (PROGRAM2) potrebno izvršiti 3 pisanja, ovaj algoritam je bolji.

```

1: procedure INSERTION SORT(A)
2:   for j ← 2 to length(A) do
3:     key ← A[j]
4:     i ← j - 1
5:     while i > 0 & A[i] > key do
6:       A[i + 1] ← A[i]
7:       i ← i - 1
8:     end while
9:     A[i + 1] ← key
10:  end for
11: end procedure

```

4. Napisati algoritam za sortiranje biranjem, takozvani SELECTION SORT.

Propustiti ulaz [5,2,4,6,1,3] kroz dobijeni algoritam i prebrojati broj upoređivanja, broj zamena elemenata niza, i broj upisivanja indeksa.

Od prvog do poslednjeg elementa algoritam bira najmanji desno od njega i menja mu mesto sa trenutnim elementom.

Razlika funkcije exchange(*A*[*i*], *A*[*i*_{min}]) i swap(*A*[*i*], *A*[*i*_{min}]) je u tome što se u exchange⁴ prvo proveruje da li je u pitanju isti element, da se ne bi vršilo nepotrebno pisanje.

⁴exchange = zamena, EN

```

1: procedure SELECTION SORT( $A$ )
2:    $n \leftarrow \text{length}(A)$ 
3:   for  $i \leftarrow 1$  to  $n$  do
4:      $i_{\min} \leftarrow i$ 
5:     for  $j \leftarrow i + 1$  to  $n$  do
6:       if  $A[j] < A[i_{\min}]$  then
7:          $i_{\min} \leftarrow j$ 
8:       end if
9:     end for
10:     $\text{exchange}(A[i], A[i_{\min}])$ 
11:  end for
12: end procedure

```

Za dati ulaz SELECTION SORT algoritam izvršava 15 poređenja (linija 6), 3 zamene (unuta exchange iz linije 10, kad je u pitanju različit indeks), i 11 zapisivanja indeksa (linije 4 i 7).

Ovaj algoritam je po broju upisivanja (zame-na) elemenata niza bolji od oba algoritma iz zadatke 2. Kad se sortiranje vrši na mediju na kome je upisivanje "skupo" (spore memorije), preporučuje se ovaj algoritam.

Ovo je, isto kao BUBBLE SORT i INSERTION SORT inkrementalno sortiranje. Ideja inkrementalnog sortiranja je da je deo niza sortiran i da se taj deo uvećava.

5. Napraviti analizu brzine rada SELECTION SORT algoritma za niz dužine n .

Napisali smo ponovo algoritam sa brojem ciklusa izvršavanja desno u komentaru.

Način implementacije pseudo koda SELECTION SORT algoritma, brzina i sposobnost procesora i memorije može da utiče na brzinu rada algoritma. Uz male razlike izvršna verzija koja se dobije kompajliranjem programa napisanog na osnovu ovog pseudo koda treba da bude otprilike kao u ovoj analizi.

Liniji algoritma i dodeljujemo vreme izvršavanja c_i . "End" linijama ne dodeljujemo vreme.

Vreme za ulazni niz $A(n)$ je

$$T(n) = c_2 + c_3(n + 1) + c_4n + c_5 \sum_{i=1}^n t_i + c_6 \sum_{i=1}^n (t_i - 1) + c_7 \sum_{i=1}^n (s_i - 1) + c_{10}n + c_{11}r$$

Kad uvrstimo da je $t_i = n - i + 1$, $s_i \leq t_i$, $r \leq n$ dobijamo

$$T(n) = c_2 + c_3(n + 1) + c_4n + c_5 \left(\frac{n^2}{2} + \frac{n}{2} \right) + c_6 \left(\frac{n^2}{2} - \frac{n}{2} \right) + c_7 \sum_{i=1}^n (s_i - 1) + c_{10}n + c_{11}r$$

Worst case

$$s_i = t_i, \quad r = n$$

```

1: procedure SELECTION SORT( $A$ )
2:    $n \leftarrow \text{length}(A)$                                 ▷ 1
3:   for  $i \leftarrow 1$  to  $n$  do                            ▷  $n + 1$ 
4:      $i_{\min} \leftarrow i$                                   ▷  $n$ 
5:     for  $j \leftarrow i + 1$  to  $n$  do                    ▷  $\sum_{i=1}^n t_i$ 
6:       if  $A[j] < A[i_{\min}]$  then                          ▷  $\sum_{i=1}^n (t_i - 1)$ 
7:          $i_{\min} \leftarrow j$                               ▷  $\sum_{i=1}^n (s_i - 1)$ 
8:       end if
9:     end for
10:    if  $i \neq i_{\min}$  then                                  ▷  $n$ 
11:       $\text{swap}(A[i], A[i_{\min}])$                               ▷  $r$ 
12:    end if
13:  end for
14: end procedure

```

$$\begin{aligned}
T(n) &= c_2 + c_3(n+1) + c_4n + c_5\left(\frac{n^2}{2} + \frac{n}{2}\right) + (c_6 + c_7)\left(\frac{n^2}{2} - \frac{n}{2}\right) + c_{10}n + c_{11}n \\
&= n^2 \frac{c_5 + c_6 + c_7}{2} + n\left(c_3 + c_4 + \frac{c_5 - c_6 - c_7}{2} + c_{10} + c_{11}\right) + c_2 + c_3 \\
&= \Theta(n^2)
\end{aligned}$$

Best case

$$s_i = 0, \quad r = 0$$

$$\begin{aligned}
T(n) &= c_2 + c_3(n+1) + c_4n + c_5\left(\frac{n^2}{2} + \frac{n}{2}\right) + c_6\left(\frac{n^2}{2} - \frac{n}{2}\right) + c_{10}n \\
&= n^2 \frac{c_5 + c_6}{2} + n\left(c_3 + c_4 + \frac{c_5 - c_6}{2} + c_{10}\right) + c_2 + c_3 \\
&= \Theta(n^2)
\end{aligned}$$

Naravno da je **Average case** $T(n) = \Theta(n^2)$.

6. Napraviti analizu brzine rada INSERTION SORT algoritma za niz dužine n .

Dodelimo linijama 2,3,4,5,6,7,9 redom vremena izvršavanja $c_2, c_3, c_4, c_5, c_6, c_7, c_9$. Broj izvršavanja određene linije ćemo napisati desno kao komentar.

```

1: procedure INSERTION SORT( $A$ )
2:   for  $i \leftarrow 2$  to length( $A$ ) do           ▷  $n$ 
3:      $key \leftarrow A[i]$                            ▷  $n - 1$ 
4:      $j \leftarrow i - 1$                              ▷  $n - 1$ 
5:     while  $j > 0$  &  $A[j] > key$  do             ▷  $\sum_{i=2}^n t_i$ 
6:        $A[j+1] \leftarrow A[j]$                      ▷  $\sum_{i=2}^n (t_i - 1)$ 
7:        $j \leftarrow j - 1$                          ▷  $\sum_{i=2}^n (t_i - 1)$ 
8:     end while
9:      $A[j+1] \leftarrow key$                          ▷  $n - 1$ 
10:  end for
11: end procedure

```

Dobijamo vreme izvršavanja algoritma:

$$T(n) = c_2n + (c_3 + c_4)(n - 1) + c_5 \sum_2^n t_i + (c_6 + c_7) \sum_2^n (t_i - 1) + c_9(n - 1),$$

gde je t_i broj koliko puta će i -ti element trebati da se pomeri prema početku plus jedan. Tako je t_i najmanje 1, a najviše i puta.

Best case $t_i = 1$

$$T_B(n) = c_2n + (c_3 + c_4)(n - 1) + c_5(n - 1) + c_9(n - 1) = \Theta(n)$$

Worst case $t_i = i$

$$T_W(n) = c_2n + (c_3 + c_4)(n - 1) + c_5(n(n+1)/2 - 1) + (c_6 + c_7)(n - 1)n/2 + c_9(n - 1) = \Theta(n^2)$$

7. Dati definiciju i primer Θ (veliko theta) i O (veliko O) ponašanja.

$$\Theta(g) = \{f \mid (\exists c_1 > 0)(\exists c_2 > 0)(\exists n_0 \in \mathbb{N})(\forall n)(n \geq n_0) \Rightarrow (0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n))\}$$

Umesto da pišemo $f \in \Theta(g)$, pišemo $f = \Theta(g)$ i čitamo: funkcija f se ponaša kao $\Theta(g)$ (kao veliko theta od g).

Na primer, $3n^2 = \Theta(n^2)$. ($c_1 = 1, c_2 = 4, n_0 = 1$)

$$O(g) = \{f \mid (\exists c_1 > 0)(\exists n_0 \in \mathbb{N})(\forall n)(n \geq n_0) \Rightarrow (0 \leq f(n) \leq c_1 g(n))\}$$

Umesto da pišemo $f \in O(g)$, pišemo $f = O(g)$ i čitamo: funkcija f se ponaša kao $O(g)$ (kao veliko O od g).

Na primer, $\ln n = O(n)$. ($c_1 = 1, n_0 = 1$) Pri tome nije $\ln n$ veliko theta od n .

8. Pokazati da je $\frac{2}{3}n^2 - 2n = \Theta(n^2)$

Za desnu nejednakost je dovoljno uzeti $c_2 = \frac{2}{3}$. Da bismo našli c_1 , podelimo levu nejednakost sa n^2 . Dobijamo

$$0 \leq c_1 \leq \frac{2}{3} - \frac{2}{n}, \text{ odakle } n \geq 4 =: n_0.$$

Sad možemo uzeti za c_1 bilo koji broj koji zadovoljava $0 \leq c_1 \leq \frac{2}{3} - \frac{2}{4} = \frac{1}{6}$, recimo $c_1 := \frac{1}{6}$.

9. Pokazati da je $100n - 1000 = O(n^2)$.

Očigledno za $c_1 = 100$ i $n_0 = 10$.

10. Pokazati da je $100n + 1000 = O(n^2)$.

Da, za, recimo, $c_1 = 20$ i $n_0 = 10$, jer je

$$0 \leq 100n + 1000 \leq c_1 n^2 \Leftrightarrow 0 \leq \frac{100}{n} + \frac{1000}{n^2} \leq c_1 \Leftrightarrow n \geq 10.$$

11. Da li je $100n + 1000 = \Theta(n^2)$?

Ne, jer $0 \leq c_1 n^2 \leq 100n + 1000 \Leftrightarrow \overbrace{c_1 n^2 - 100n - 1000}^* \leq 0$, počev od nekog n_0 , a to je nemoguće, jer je parabola \star zakrivljena nagore (zato što je $c_1 > 0$), i u beskonačnosti sigurno jeste iznad x -ose, a ne ≤ 0 .

12. Dati definiciju i primer Ω (veliko omega) ponašanja.

$$\Omega(g) = \{f \mid (\exists c > 0)(\exists n_0 \in \mathbb{N})(\forall n) (n \geq n_0) \Rightarrow (0 \leq cg(n) \leq f(n))\}$$

Umesto da pišemo $f \in \Omega(g)$, pišemo $f = \Omega(g)$ i čitamo: funkcija f se ponaša kao $\Omega(g)$ (kao veliko omega od g).

Na primer, $3n^2 = \Omega(n^2)$. ($c_1 = 1, c_2 = 4, n_0 = 1$)

13. Dati vezu Ω , Θ i O ponašanja.

$$\text{Očigledno važi: } f = \Theta(g) \Leftrightarrow (f = \Omega(g) \wedge f = O(g))$$

14. Pokazati da je $\frac{2}{3}n^2 - 2n = \Omega(n^2)$

Da, jer je u jednom od prethodnih zadataka bilo $\frac{2}{3}n^2 - 2n = \Theta(n^2)$.

15. Da li je $100n + 1000 = O(n)$?

Da, očigledno, za, recimo, $c_1 = 101$ i $n_0 = 1001$. Vidimo da je tvrđenje zadatka 10 pregrubo. Važi i $100n + 1000 = o(n^2)$ (malo o).

16. Da li je $100n + 1000 = \Theta(n)$?

Da, očigledno, za, recimo, $c_2 = 101, c_1 = 100$ i $n_0 = 1001$.

17. Pokazati da je $n \ln n + n = O(n^2)$?

Da, jer je niz $\frac{n \ln n + n}{n^2}$ konvergentan (konvergira ka nuli), zato je ograničen, to jest postoji c_1 takvo da počev od nekog n_0 važi $\frac{n \ln n + n}{n^2} \leq c_1$, što je ekvivalentno sa $n \ln n + n \leq c_1 n^2$.

18. Pokazati da je $n \ln n + n = \Omega(n)$?

Zato što je $\lim_{n \rightarrow \infty} \frac{n \ln n + n}{n} = \infty$, postoji konstanta $c > 0$ i $n_0 \in \mathbb{N}$ tako da je

$$\forall n \in \mathbb{N}, n \geq n_0 \Rightarrow c < \frac{n \ln n + n}{n} \Leftrightarrow cn < n \ln n + n.$$

19. Znajući da je worst-case vreme izvršavanja INSERTION SORT algoritma $O(n^2)$, n je veličina ulaza, da li to znači da će za svaki ulazni niz veličine n , vreme izvršavanja biti $O(n^2)$?

Da. Zato što $O(n^2)$ ponašanje daje gornju granicu za vreme izvršavanja.

20. Znajući da je worst-case vreme izvršavanja INSERTION SORT algoritma $\Theta(n^2)$, n je veličina ulaza, da li to znači da će za svaki ulazni niz veličine n , vreme izvršavanja biti $\Theta(n^2)$?

Ne. Zato što $\Theta(n^2)$ ponašanje daje i gornju i donju granicu. Donja granica neće biti ispunjena jer za već sortiran niz veličine n (best-case za INSERTION SORT) vreme izvršavanja je $\Theta(n)$.

21. Naći asimptotsku ocenu za $T(n)$, vreme izvršavanja INSERTION SORT algoritma za ulaz dužine n .

U zadatku 6 smo videli da je za $T(n)$ Best case $\Theta(n)$ i Worst case $\Theta(n^2)$.

Možemo reći da je $T(n) = \Omega(n)$ i $T(n) = O(n^2)$.

Asimptotske oznake imaju svoju analogiju sa brojevima:

$$f = \Omega(g) \Leftrightarrow f \geq g$$

$$f = O(g) \Leftrightarrow f \leq g$$

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

22. Dati definiciju i primer malog o ponašanja.

$$o(g) = \{f \mid (\forall c > 0)(\exists n_0 \in \mathbb{N})(\forall n) (n \geq n_0) \Rightarrow (0 \leq f(n) < cg(n))\}$$

Kao i do sad, $f \in o(g)$ pišemo $f = o(g)$.

U matematičkoj analizi je česta upotreba oznake malo o . Jedina razlika je što mi zahtevamo da su funkcije koje posmatramo nenegativne.

Za (počev od nekog n_0) nenegativne funkcije f i g važi

$$f = o(g) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

Analogija sa brojevima bi bila: $f = o(g) \Leftrightarrow f < g$.

Oznaka malo o se koristi da bi jasnije pokazala asimptotski poredak: Na primer

$$n \ln n + n = o(n^2).$$

Iz definicije je očigledno da $f = o(g) \Rightarrow f = O(g)$.

Napomenimo još da za asimptotske oznake važi tranzitivnost:

$$f = O(g) \wedge g = O(h) \Rightarrow f = O(h)$$

$$f = \Theta(g) \wedge g = \Theta(h) \Rightarrow f = \Theta(h)$$

$$f = \Omega(g) \wedge g = \Omega(h) \Rightarrow f = \Omega(h)$$

$$f = o(g) \wedge g = o(h) \Rightarrow f = o(h)$$

23. Napisati algoritam koji za dati niz nalazi najmanji i najveći element. Odrediti red broja poređenja u datom algoritmu (prema veličini niza n).

Rešenje je program MINIMAXI.

24. Napisati algoritam koji za dati niz nalazi najmanji i najveći element a da pri tome ima najviše $1.5n$ poređenja, gde je n veličina niza.

Rešenje je program MINIMAXI1.

```

1: function MINIMAXI(A)
2:    $n \leftarrow \text{length}(A)$ 
3:    $min \leftarrow A[1]$ 
4:    $max \leftarrow A[1]$ 
5:   for  $i \leftarrow 2$  to  $n$  do
6:     if  $A[i] < min$  then
7:        $min \leftarrow A[i]$ 
8:     end if
9:     if  $A[i] > max$  then
10:       $max \leftarrow A[i]$ 
11:    end if
12:  end for
13:  return  $[min, max]$ 
14: end function

```

Očigledno je broj poređenja (linije 6 i 9): $n - 1 + n - 1 = 2n - 2 = O(n)$ (sa $c_1 = 2, n_0 = 1$).

Odnosno: za svaki element (osim prvog, a to nije bitno) se vrši po dva poređenja. Objašnjenje zadatka 24:

Za svaka dva elementa (osim prvog za neparno n) se vrši 3 poređenja, što daje ukupni broj poređenja $\frac{3}{2}n = 1.5n$ za n parno, a ukupni broj poređenja je i manji za n neparno.

```

1: function MINIMAXI1(A)
2:    $n \leftarrow \text{length}(A)$ 
3:   if  $\text{odd}(n)$  then
4:      $max \leftarrow A[1]$ 
5:      $min \leftarrow A[1]$ 
6:      $j \leftarrow 2$ 
7:   else
8:     if  $A[1] > A[2]$  then
9:        $max \leftarrow A[1]$ 
10:       $min \leftarrow A[2]$ 
11:    else
12:       $min \leftarrow A[1]$ 
13:       $max \leftarrow A[2]$ 
14:    end if
15:     $j \leftarrow 3$ 
16:  end if
17:  while  $j < n$  do
18:    if  $A[j] > A[j + 1]$  then
19:      if  $A[j] > max$  then
20:         $max \leftarrow A[j]$ 
21:      end if
22:      if  $A[j + 1] < min$  then
23:         $min \leftarrow A[j + 1]$ 
24:      end if
25:    else
26:      if  $A[j] < min$  then
27:         $min \leftarrow A[j]$ 
28:      end if
29:      if  $A[j + 1] > max$  then
30:         $max \leftarrow A[j + 1]$ 
31:      end if
32:    end if
33:     $j \leftarrow j + 2$ 
34:  end while
35:  return  $[min, max]$ 
36: end function

```

Odd⁵ funkcija vraća true ako je n neparan broj.

⁵odd = neparno, EN; even = parno, EN

25. Napisati rekurzivni i iterativni algoritam za računanje faktoriijala $n! = n \cdot (n - 1) \cdot \dots \cdot 2 \cdot 1$.

```

function FACTORIAL( $n$ )
  if  $n \leq 1$  then
    return 1
  else
    return  $n \cdot$  FACTORIAL( $n - 1$ )
  end if
end function

```

```

function FACTORIAL1( $n$ )
   $f \leftarrow 1$ 
  for  $k \leftarrow 2$  to  $n$  do
     $f \leftarrow f \cdot k$ 
  end for
  return  $f$ 
end function

```

26. Fibonačijev niz čine redom brojevi 0, 1, 1, 2, 3, 5, 8, 13, ... Prva dva su redom 0 i 1, svaki sledeći je zbir prethodna dva. Napisati rekurzivni i iterativni algoritam za računanje n -tog Fibonačijevog broja.

```

function FIBONACCI( $n$ )
  if  $n \leq 1$  then
    return  $n$ 
  else
    return FIBONACCI( $n - 1$ ) +
      FIBONACCI( $n - 2$ )
  end if
end function

```

```

function FIBONACCI1( $n$ )
  if  $n \leq 1$  then
    return  $n$ 
  else
     $f_0 \leftarrow 0$ 
     $f_1 \leftarrow 1$ 
    for  $k \leftarrow 2$  to  $n$  do
       $f \leftarrow f_0 + f_1$ 
       $f_0 \leftarrow f_1$ 
       $f_1 \leftarrow f$ 
    end for
  end if
  return  $f$ 
end function

```

Rekurzivna procedura FIBONACCI ne sme da se pozove sa velikim n jer zaglavi kompjuter velikim drvetom rekurzije.

Vreme izvršavanja je manje za iterativne verzije programa iz dva poslednja zadatka.

27. Napisati proceduru za spajanje dva susedna sortirana podniza niza A u sortirani podniz. Iskoristiti dobijenu proceduru za pisanje MERGE SORT⁶ algoritma za sortiranje.

U realizaciji ćemo koristiti takozvani džoker simbol ∞ koji je veći od svih elemenata korišćenog tipa. Ova tehnika omogućava pojednostavljenje koda uz neznatni utrošak memorije. Ti simboli ne postoje u svakom tipu podataka, za neke tipove kao što je int, može se koristiti maxint.

⁶merge = spojiti, stopiti, EN

```

procedure MERGE( $A, p, q, r$ )
  ▷ spaja podniz niza  $A$  od  $p$  do  $q$  sa podnizom od  $q + 1$  do  $r$ 
  for  $k \leftarrow p$  to  $q$  do           ▷ Prebacujemo prvi podniz u  $L$ 
     $L[k - p + 1] \leftarrow A[k]$ 
  end for
   $L[q - p + 2] \leftarrow \infty$ 
  for  $k \leftarrow q + 1$  to  $r$  do       ▷ Prebacujemo drugi podniz u  $R$ 
     $R[k - q] \leftarrow A[k]$ 
  end for
   $R[r - q + 1] \leftarrow \infty$ 
   $i \leftarrow 1$ 
   $j \leftarrow 1$ 
  for  $k \leftarrow p$  to  $r$  do           ▷ Spajanje
    if  $L[i] \leq R[j]$  then
       $A[k] \leftarrow L[i]$ 
       $i \leftarrow i + 1$ 
    else
       $A[k] \leftarrow R[j]$ 
       $j \leftarrow j + 1$ 
    end if
  end for
end procedure
procedure SORT( $A, p, r$ )
  ▷ Procedura koja se rekurzivno poziva
  if  $p < r$  then
     $q \leftarrow \lfloor (p + r) / 2 \rfloor$ 
    SORT( $A, p, q$ )
    SORT( $A, q + 1, r$ )
    MERGE( $A, p, q, r$ )
  end if
end procedure
procedure MERGE SORT( $A$ )
  ▷ Glavna procedura koju poziva korisnik
   $n \leftarrow \text{length}(A)$ 
  SORT( $A, 1, n$ )
end procedure

```

Ovaj program deli dobijeni podniz na pola, na svakoj polovini primenjuje rekurzivno istu tehniku, potom spaja (merge) polovine u jedan, sortiran, podniz.

Povratak iz rekurzije je prazan niz za koji se smatra da je sortiran.

Za rekurzivno pozivanje ovog programa je pogodno statički alocirati memoriju za podnizove L i R koju bi koristile sve instance u svim rekurzivnim pozivima procedure MERGE.

28. Napisati algoritam za QUICK SORT sortiranje.

Ovo je vrlo često korišćen algoritam sortiranja koji se pokazao brzim i jednostavnim.

Koristi divide & conquer⁷ tehniku isto kao MERGE SORT.

Koristi se procedura PARTITION koja grupiše premeštanjem elemenata odabrani podniz (od p do r) tako da odabrani element (recimo poslednji) bude na svom mestu po redosledu, da ispred njega budu manji ili jednaki od njega, iza njega veći od njega. (★)

Ako je q redni broj mesta odabranog broja, onda se rekurzivno poziva ista procedura na podnizove od p do $q - 1$ i od $q + 1$ do r , gde su p i r granice posmatranog podniza. Prvi put se uzima $p = 1$ i $r = n$.

```
function PARTITION( $A, p, r$ )
  ▷ zamenama srediti niz u skladu sa (★)
   $x \leftarrow A[r]$ 
   $i \leftarrow p - 1$ 
  for  $j \leftarrow p$  to  $r - 1$  do
    if  $A[j] \leq x$  then
       $i \leftarrow i + 1$ 
      exchange( $A[i], A[j]$ )
    end if
  end for
  exchange( $A[i + 1], A[r]$ )
  return  $i + 1$ 
end function
procedure SORT( $A, p, r$ )
  ▷ Procedura koja se rekurzivno poziva
  if  $p < r$  then
     $q \leftarrow$  PARTITION( $A, p, r$ )
    SORT( $A, p, q - 1$ )
    SORT( $A, q + 1, r$ )
  end if
end procedure
procedure QUICK SORT( $A$ )
  ▷ Glavna procedura koju poziva korisnik
   $n \leftarrow$  length ( $A$ )
  SORT( $A, 1, n$ )
end procedure
```

⁷ divide & conquer = podeli pa osvoji, EN

29. Analizirati red vremena izvršavanja, red potrošnje memorije i stabilnost algoritama za sortiranje datih u prethodnim zadacima.

U sledećoj tabeli je data tražena asimptotska vrednost za sledeće algoritme: BUBBLE SORT (B), INSERTION SORT (I), SELECTION SORT (S), MERGE SORT (M), QUICK SORT (Q), za Best case (B), Average case (A) i Worst case (W).

Takođe je dat podatak o redu veličine dodatnog memorijskog prostora potrebnog za sortiranje (M) i podatak o stabilnosti posmatranog algoritma (S).

	B	A	W	M	S
B	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	DA
I	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	DA
S	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	NE
M	$\Theta(n \ln n)$	$\Theta(n \ln n)$	$\Theta(n \ln n)$	$\Theta(n)$	DA
Q	$\Theta(n \ln n)$	$\Theta(n \ln n)$	$\Theta(n^2)$	$\Theta(\ln n)$	NE

Stabilnost je osobina algoritma da ključevima sa istom vrednošću ne menja redosled. Ovo je korisna osobina algoritama za sortiranje, jer za podatke sortirane po jednom ključu stabilni algoritmi čuvaju redosled kada se primeni sortiranje po drugom ključu.

Naša implementacija QUICK SORT algoritma nije stabilna. Ako se ovaj algoritam modifikuje sa ciljem da dobije stabilnost, gube se performanse.

Inače je QUICK SORT u praksi (primenjen na uobičajene podatke) najbrži od posmatranih algoritama.

30. Izvršiti testiranje posmatranih algoritama na nizu *random*⁸ brojeva obima $10^1, 10^2, 10^3, 10^4, 10^5$.

Algoritmi iz prethodnih zadataka su kodirani i testirani na random uzorku i dobijeni su rezultati:

	B	I	S	M	Q
10	7.30E-05	6.08E-05	6.10E-05	0.00034	0.0003
100	0.00021	6.10E-05	0.0001	0.00235	0.0008
1000	0.01873	0.00417	0.00625	0.0233	0.0067
10000	1.92922	0.35455	0.60198	0.23819	0.0669
100000	192.796	35.4994	60.2424	2.38437	0.6754

Vidimo da je za veliko n QUICK SORT najbrži iako je na nizu do veličine 1000 brži INSERTION SORT. U praksi se za nizove obima manjeg od neke granice u rekurziji sa QUICK SORTA prelazi na INSERTION SORT. Time se šteti memorija i vreme, jer je INSERTION SORT brži za male nizove i troši manje memorije.

⁸*random* = slučajni, EN

2 Apstraktni tipovi podataka

31. Napisati algoritam DET za računanje determinante matrice formata $n \times n$ dovođenjem na gornje-trougao nu determinantu.

```

1: function DET(A,n)
2:   znak ← 1
3:   for i ← 1 to n - 1 do
4:     pm ← PIVOT(A,n,i)
5:     if ¬pm then
6:       return 0.0
7:     end if
8:     znak ← znak * pm
9:     for k ← i + 1 to n do
10:      α ← A[k,i] / A[i,i]
11:      A[k,i] ← 0.0
12:      for j ← i + 1 to n do
13:        A[k,j] ← A[k,j] - αA[i,j]
14:      end for
15:    end for
16:  end for
17:  if A[n,n] = 0 then
18:    return 0.0
19:  end if
20:  d ← znak * A[1,1]
21:  for i ← 2 to n do
22:    d ← d * A[i,i]
23:  end for
24:  return d
25: end function

function PIVOT(A,n,m)
  i1 ← m; j1 ← m; pm ← 1
  for i ← m to n do
    for j ← m to n do
      if |A[i,j]| > |A[i1,j1]| then
        i1 ← i; j1 ← j
      end if
    end for
  end for
  if A[i1,j1] = 0 then
    return 0
  end if
  if i1 ≠ m then
    pm ← pm * (-1)
    for j ← m to n do
      swap(A[i,j], A[i1,j])
    end for
  end if
  if j1 ≠ m then
    pm ← pm * (-1)
    for i ← 1 to n do
      swap(A[i,j], A[i,j1])
    end for
  end if
  return pm
end function

```

32. Napraviti biblioteku funkcija u programskom jeziku C koje omogućavaju računске operacije sa matricama.

Elementi matrice $A_{m \times n}$ se smestaju po vrstama u niz dužine $m * n$. Indeksiranje elemata u C-u počinje od 0, tako da element matrice $A[i,j]$ se u nizu nalazi na mestu $A[i * n + j]$.

matrice.h

```

void addmat(double *, double *, double *, int, int); // sabiranje (I,I,O,I,I)
void multmat(double *, double *, double *, int, int, int); // mnozenje (I,I,O,I,I,I)
void multscal(double *, double *, double *, int, int); // mnozenje skalarom (I,I,O,I,I)
void transpose(double*, double *, int, int); // transponovanje (I,O,I,I)
int inverse(double*, double *, int); // inverzna (I/O,O,I)
double det(double*, int); // determinanta (I/O,I)
int printmatrix(double*, int, int); // stampanje (I,I,I)

```

matrice.c

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define epsilon 1e-12
void addmat(double *A, double *B, double *C, int m, int n)
{ // sabiranje matrica A mXn + B mXn = C mXn
  int i;

  for(i=0;i<m*n;i++){
    C[i] = A[i]+B[i];
  }
}
void multmat(double *A, double *B, double *C,
             int m, int p, int n)
{ // mnozenje matrica A mXp * B pXn = C mXn
  int i,j,k;

  for(i=0;i<m;i++){
    for(j=0;j<n;j++){
      C[i*n+j] = 0;
      for(k=0;k<p;k++){
        C[i*n+j] = C[i*n+j]+A[i*p+k]*B[k*n+j];
      }
    }
  }
}
void multscal(double x, double *A, double *B, int m, int n)
{ // mnozenje matrica x * A mXn = B mXn
  int i;

  for(i=0;i<m*n;i++){
    B[i] = x*A[i];
  }
}
void transpose(double *A, double *B, int m, int n)
{ // transpose( A mXn ) = B nXm
  int i,j;

  for(i=0;i<m;i++){
    for(j=0;j<n;j++){
      B[j*m+i] = A[i*n+j];
    }
  }
}
int rowpivot(double *A, int n, int m)
{ // pivotizuje vrstu m sa il, vraca -1 ako rang<n
  int i,j,il=m;
  double temp;
  for(i=m+1;i<n;i++){
    if(fabs(A[i*n+m])>fabs(A[il*n+m])){
      il = i;
    }
  }
  if(fabs(A[il*n+m])<epsilon)
    return -1;
  if(il!=m){
    for(j=m;j<n;j++){
      temp=A[il*n+j]; A[il*n+j]=A[m*n+j]; A[m*n+j]=temp;
    }
  }
  return il;
}
int inverse(double *A, double *B, int n)
{ // A^(-1) = B (menja A) daje n-rang(A). rang(A)<n => nema inv
  int i,j,k,il;
  double alpha;
  for(i=0;i<n;i++){
    for(j=0;j<n;j++){
      B[i*n+j] = (double)(i==j);
    }
  }
  // Gausove eliminacije
  for(i=0;i<n-1;i++){
    if((il=rowpivot(A,n,i))==-1){
      return (n-i); // rang je i, A nema inverznu
    }
    for(j=0;j<n;j++){
      alpha = B[i*n+j]; B[i*n+j] = B[il*n+j];
      B[il*n+j] = alpha;
    }
    for(k=i+1;k<n;k++){
      alpha = A[k*n+i]/A[i*n+i];
      A[k*n+i] = 0;
      for(j=i+1;j<n;j++){
        A[k*n+j] = A[k*n+j] - alpha*A[i*n+j];
      }
      for(j=0;j<n;j++){
        B[k*n+j] = B[k*n+j] - alpha*B[i*n+j];
      }
    }
  }
}
  if(fabs(A[n*n-1])<epsilon)
    return 1; // rang je n-1, A nema inverznu
  // Zordanove eliminacije
  for(i=n-1;i>0;i--){
    for(k=0;k<i;k++){
      alpha = A[k*n+i]/A[i*n+i];
      for(j=0;j<n;j++){
        B[k*n+j]=B[k*n+j]-alpha*B[i*n+j];
      }
    }
    for(j=0;j<n;j++){
      B[i*n+j]=B[i*n+j]/A[i*n+i];
    }
  }
  for(j=0;j<n;j++){
    B[j]=B[j]/A[i];
  }
  return 0; // A ima inverznu, nalazi se u B
}
int pivot(double *A, int n, int m)
{ // pivotizuje vrstu il / kolonu j1 sa m / m, vraca +/-1
  int i,j,il=m,j1=m,znak=1;
  double temp;
  for(i=m;i<n;i++){
    for(j=m;j<n;j++){
      if(fabs(A[i*n+j])>fabs(A[il*n+j1])){
        il = i;
        j1 = j;
      }
    }
  }
  if(fabs(A[il*n+j1])<epsilon)
    return 0;
  if(il!=m){
    znak = znak*(-1);
    for(j=m;j<n;j++){
      temp=A[il*n+j]; A[il*n+j]=A[m*n+j]; A[m*n+j]=temp;
    }
  }
  if(j1!=m){
    znak = znak*(-1);
    for(i=0;i<n;i++){
      temp=A[i*n+m]; A[i*n+m]=A[i*n+j1]; A[i*n+j1]=temp;
    }
  }
  return znak;
}
double det(double *A, int n)
{ // racuna determinantu matrice A nXn, (menja elemente A)
  int i,j,k,pm,znak=1;
  double alpha;
  for(i=0;i<n-1;i++){
    if(!(pm=pivot(A,n,i)))
      return 0.0;
    znak = znak*pm;
    for(k=i+1;k<n;k++){
      alpha = A[k*n+i]/A[i*n+i];
      A[k*n+i] = 0;
      for(j=i+1;j<n;j++){
        A[k*n+j] = A[k*n+j] - alpha*A[i*n+j];
      }
    }
  }
  if(fabs(A[n*n-1])<epsilon)
    return 0.0;
  alpha = znak*A[0];
  for(i=1;i<n;i++){
    alpha = alpha * A[i*n+i];
  }
  return alpha;
}
void printmatrix(double *A, int m, int n)
{ // stampa matricu A mXn, m,n < 10
  int i,j;

  printf("\n+");
  for(j=0;j<m;j++) printf("_____");
  printf("-+\n");
  for(i=0;i<m;i++){
    printf("|_");
    for(j=0;j<n;j++){
      printf("_%6.2f_",A[i*n+j]);
    }
    printf("|_\n");
  }
  printf("-+\n");
  for(j=0;j<m;j++) printf("_____");
  printf("-+\n");
}

```

33. Koristeći biblioteku iz zadatka 32 napisati program u C-u koji rešava matričnu jednačinu $A + BX = C$, gde su

$$A = \begin{bmatrix} 1 & 2 & -3 \\ 4 & -7 & -16 \\ -7 & -18 & -16 \end{bmatrix}, B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 7 & 6 \\ 7 & 8 & -16 \end{bmatrix}, C = \begin{bmatrix} -1 & 3 & 4 \\ 2 & 2 & 10 \\ 16 & 20 & 21 \end{bmatrix}.$$

Primenjujući matričnu algebru dobijamo $X = B^{-1}(C - A)$. U C-u to računamo pomoću sledećeg programa

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "matrice.h"
#define epsilon 1e-12

int main()
{
    int nr, n;

    n = 3;
    double A[]={1,2,-3,4,-7,-16,-7,-18,-16};
    double B[]={1,2,3,4,7,6,7,8,-16};
    double C[]={-1,3,4,2,2,10,16,20,21};
    double *minusA=malloc(n*n*sizeof(double));
    double *CminusA=malloc(n*n*sizeof(double));
    double *Binv=malloc(n*n*sizeof(double));
    double *X=malloc(n*n*sizeof(double));

    if ((nr=inverse(B, Binv, n))){
        printf("\nNe postoji B^(-1), rang(B) je %d.", n-nr);
    }
    else{
        multscal(-1,A, minusA, 3, 3);
        addmat(C, minusA, CminusA, 3, 3);
        multmat(Binv, CminusA, X, 3, 3, 3);
        printmatrix(X, 3, 3);
    }
    return 0;
}
```

Kompajliranje i pokretanje gornjeg programa daje rešenje $X =$

```
+-
|  1.00    2.00    3.00  |
|  0.00    1.00    2.00  |
| -1.00   -1.00   -0.00  |
+-
```

```
Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```


34. Izračunati broj sabiranja i množenja elemenata matrice A prilikom računanja determinante reda n upotrebom algoritma iz zadatka 31.

U pivotizaciji nema množenja i sabiranja elemenata matrice A . Dovođenje na gornju trougaonu determinantu se vrši Gausovim eliminacijama:

Idući brojačem i po dijagonali od prvog do pretposljednog elementa, množenjem i -te vrste brojem $\alpha = A[k,i]/A[i,i]$ i oduzimanje od k -te vrste dobija se gornje trougaona determinanta. Njena vrednost je proizvod elemenata sa glavne dijagonale.

Sabiranja elemenata matrice se vrše u liniji 13 algoritma (oduzimanje brojimo kao sabiranje). Množenja se vrše u liniji 10 (deljenje brojimo kao množenje) i liniji 13, kao i na samom kraju, u liniji 22, kad se množe elementi sa glavne dijagonale.

i	$SAB(i)$	$MNO(i)$
1	$(n-1)(n-1)$	$(n-1)(n-1) + n-1$
2	$(n-2)(n-2)$	$(n-2)(n-2) + n-2$
\vdots	\ddots	\dots
$n-2$	$2 \cdot 2$	$2 \cdot 2 + 2$
$n-1$	1	$1 + 1$
linija 22	0	$n-1$
Σ	$\Sigma SAB(i)$	$\Sigma MNO(i)$

Koristeći formulu $\sum_{k=1}^n k^2 = \frac{1}{6}n(n+1)(2n+1)$ i $\sum_{k=1}^n k = \frac{1}{2}n(n+1)$, dobijamo broj sabiranja i množenja:

$$\sum SAB(i) = \frac{1}{6}(n-1)n(2n+1) = \Theta(n^3)$$

$$\sum MNO(i) = \sum SAB(i) + \frac{1}{2}(n-1)n + n-1 = \Theta(n^3)$$

35. Izračunati broj sabiranja i množenja elemenata matrice A prilikom računanja determinante reda n koristeći definiciju:

$$\begin{vmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{vmatrix} = \sum_{\substack{\text{po svim perm.} \\ (i_1, i_2, \dots, i_n)}} (-1)^{\sigma(i_1, i_2, \dots, i_n)} a_{1, i_1} a_{2, i_2} \cdots a_{n, i_n}$$

gde je $\sigma(i_1, i_2, \dots, i_n)$ broj inverzija permutacije (i_1, i_2, \dots, i_n) , čija parnost odlučuje znak sabirka.

Očigledno ima $n!$ sabiraka, tako da je za ovaj algoritam

$$\sum SAB(i) = n! - 1$$

$$\sum MNO(i) = n!(n-1)$$

36. Ako jedno sabiranje traje $3.13E-9$, a množenje $3.75E-9$, koliko vremena treba da se saberu i pomnože elementi matrice 100×100 pomoću algoritma iz zadatka 31, čija analiza je u zadatku 34, odnosno, preko definicije, analiza u zadatku 35?

Vreme dobijamo po formuli

$$\sum SAB(i) \times 3.13 \times 10^{-9} + \sum MNO(i) \times 3.75 \times 10^{-9}.$$

Za $n = 100$, formule iz zadatka 34 i 35 daju:

	vreme
zad 34	0.0023s
zad 35	$3.49 \times 10^{151}s =$ 1.11×10^{144} godina

Očigledno je računanje determinante dovođenjem na gornju trougaonu neuporedivo brže za velike formate matrice.

Štaviše, zbog manjeg broja računskih operacija nagomilavanje greške zaokruživanja je manje i dobija se precizniji rezultat.

Preciznosti rezultata doprinosi i pivotizacija. Dovođenje najvećeg elementa po apsolutnoj vrednosti na dijagonalu utiče na znatno smanjivanje uticaja greške zaokruživanja.

Pivotizacija je istovremeno i test da li je data matrica singularna: kad se procedura rowpivot vrati sa vrednošću nr različitom od 0, onda je matrica singularna, njena determinanta je 0, a rang je $n - nr$.

37. Analizirati algoritam za računanje inverzne matrice inverse iz zadatka 32.

Posmatrani algoritam prvo vrši Gausove eliminacije na matrici A i istovremeno na matrici B koja je u početku bila jedinična matrica.

Pivotizaciju vrši samo po vrstama. Akko je matrica singularna, onda će pri izvršavanju Gausovih eliminacija rowpivot vratiti -1, što će biti signal proceduri inverse da zaključi da je matrica singularna, a da je njen rang trenutna vrednost brojača i .

Naravno, isto kao u implementaciji algoritma za računanje determinante u *floating point*⁹ aritmetici ne vrši se upoređivanje dobijene vrednosti sa nulom po jednakosti. Razlog je što mala greška zaokruživanja, koja je neminovna, dovodi do pogrešnog zaključka. Stoga nulom smatramo brojeve čija je apsolutna vrednost manja od $\epsilon = 10^{-12}$.

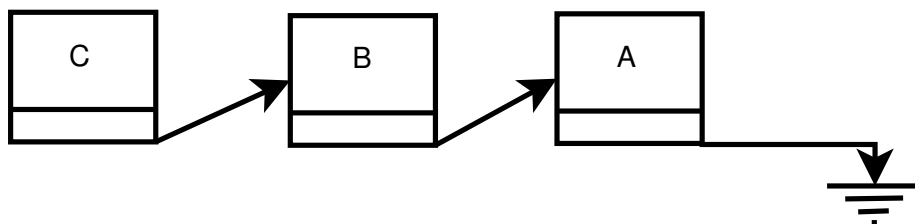
Algoritam potom vrši Žordanove eliminacije na matrici A i B, dovodeći elementarnim transformacijama po vrstama matricu A na jediničnu i matricu B na inverznu od A.

Žordanove transformacije se ne moraju vršiti na elementima matrice A.

⁹*floating point* = aritmetika pokretnog zareza, EN

38. Napisati program u programskom jeziku C koji pravi povezanu listu kao sa slike, ispisuje njen sadržaj, i oslobađa dinamički alociranu memoriju. Koristiti tip podataka cvor:

```
typedef struct _cvor cvor; struct _cvor {
    char podatak;
    cvor *sledeci;
};
```



gomila.c

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

typedef struct _cvor cvor;

struct _cvor
{
    char podatak;
    cvor *sledeci;
};

int main()
{
    cvor *gomila = NULL;
    cvor *S;

    S = malloc(sizeof(cvor));
    (*S).podatak = 'A';
    (*S).sledeci = NULL;

    gomila = S;
```

```
    S = malloc(sizeof(cvor));
    S->podatak = 'B';
    S->sledeci = gomila;
    gomila = S;

    S = malloc(sizeof(cvor));
    S->podatak = 'C';
    S->sledeci = gomila;
    gomila = S;

    while(S)
    {
        printf("%c\n", S->podatak);
        S = S->sledeci;
    }

    while(gomila){
        S = gomila;
        gomila = gomila->sledeci;
        free(S);
    }

    return 0;
}
```

39. Napisati algoritme za implementaciju apstraktnog tipa podataka (ADT¹⁰) Stack pomoću povezanih listi.

Treba realizovati MAKENULL, ISEMPY, PUSH, POP, TOP, ISMEMBER, CLEAR.

Napravićemo tip podataka koji sadrži data polje i pokazivač na sledećeg člana. Prazan stek je null pointer. Ovo je uobičajena implementacija steka pomoću povezanih listi.

Posmatrani tip podataka ćemo zvati *node* i to će biti struktura od dva polja čiji su tipovi: *S.data : listdata* i *S.next : *node* i

U programskom jeziku C struktura koja će sadržati čvorove naše povezane liste bila bi opisana sledećim kodom.

```
typedef char listdata;
typedef struct _node node;

struct _node {
    listdata data;
    node *next;
};
```

U pseudokodu kada se procedure pozivaju sa parametrom koji predstavlja adresu promenljive *A*, zapisivaćemo to sa **A*.

```
procedure MAKENULL(*S)
    S ← NULL
end procedure
```

```
function ISEMPY(*S)
    return S = NULL
end function
```

```
procedure PUSH(*S,d)
    new ← malloc(sizeof(node))
    new.data ← d
    new.next ← S
    S ← new
end procedure
```

```
function POP(*S)
    temp ← S
    d ← temp.data
    S ← temp.next
    free(temp)
    return d
end function
```

```
function TOP(*S)
    return S.data
end function
```

```
function ISMEMBER(*S,d)
    temp ← S
    while ¬ISEMPY(temp) do
        if temp.data = d then
            return 1
        end if
        temp ← temp.next
    end while
    return 0
end function
```

```
procedure CLEAR(*S)
    while ¬ISEMPY(S) do
        POP(S)
    end while
end procedure
```

¹⁰ADT = Abstract Data Type = apstraktni tip podataka, EN; stack = kamara, stog, gomila, EN

40. Dati implementaciju procedura za ADT STACK iz zadatka 39 u programskom jeziku C. Napraviti fajlove *stack.h* i *stack.c* koji će sadržati interfejs i implementaciju procedura.

Stek je pokazivač (*pointer, EN*) na čvor. Prazan stek je NULL pointer. U procedurama i funkcijama prenosimo pokazivač na stek kao parametar *S*. Ako sadržaj steka treba da se menja (u procedurama *MAKENULL*, *PUSH*, *POP*, *CLEAR*), šaljemo adresu pokazivača na stek kao parametar (**S*).

Data sadržaj elemenata steka je u programu stavljen da je *char* zbog upotrebe u sledećem zadatku. U tom slučaju poređenje na jednakost u funkciji *ISMEMBER* je moglo da se uradi komandom operatorom *==*. Ipak, zbog opštosti, u programu je stavljena funkcija *memcmp* iz string biblioteke, tako da je sad lako moguće kao data sadržaj elemenata steka staviti i druge tipove.

stack.h

```
typedef char listdata;
typedef struct _node node;
typedef node *stack;

void makenull(stack *);
int isempty(stack);
int push(stack *, listdata);
listdata pop(stack *);
listdata top(stack);
void clear(stack *);
int ismember(stack, listdata *);
void printstack(stack);
```

stack.c

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

struct _node
{
    listdata data;
    node *next;
};

int isequal(listdata * lhs, listdata * rhs)
{
    return !memcmp(lhs, rhs, sizeof(listdata));
}

void makenull(stack *S)
{
    *S = NULL;
```

```
}
int isempty(stack S)
{
    return (int) (S==NULL);
}
int push(stack *S, listdata d)
{
    node *S_new = malloc(sizeof(node));

    if(!S_new)
        return 1;

    S_new->data = d;
    S_new->next = *S;
    *S = S_new;
    return 0;
}
listdata pop(stack *S)
{
    node *S_temp = *S;
    listdata d = S_temp->data;
    *S = S_temp->next;
    free(S_temp);
    return d;
}
listdata top(stack S)
{
    listdata d = S->data;
    return d;
}
void clear(stack *S)
{
    while(!isempty(*S)) pop(S);
}
int ismember(stack S, listdata *chp)
{
    while(S)
    {
        if(isequal(&(S->data), chp))
            return 1;
        S = S->next;
    }
    return 0;
}
void printstack(stack S)
{
    while(S)
    {
        printf("%c\n", S->data);
        S = S->next;
    }
}
```

41. Napisati program u programskom jeziku C koji koristeći ADT stack iz zadatka 40: pravi stek, u njega ubacuje redom elemente 'A', 'B', 'C', zatim skida i ispisuje elemente steka.

mali_stek.c

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

int main(void)
{
    stack S;

    makenull(&S);

    push(&S, 'A');
    push(&S, 'B');
    push(&S, 'C');

    while (!isempty(S))
        printf("%c\n", pop(&S));

    return 0;
}
```

42. Napisati program u programskom jeziku C koji koristeći ADT stack učitava tekst iz fajla *ulaz.txt* i ispisuje u fajl *izlaz.txt* reč po reč unazad. Reči su nizovi karaktera odvojeni simbolima: space, tab, newline.

unazad.c

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

int main(void)
{
    FILE *ulaz, *izlaz;
    stack S;
    listdata ch;

    makenull(&S);
    ulaz = fopen("ulaz.txt", "r");
    izlaz = fopen("izlaz.txt", "w");

    while ((ch=fgetc(ulaz)) != EOF){
        if ((ch==' ') || (ch=='\t') || (ch=='\n')){
            while (!isempty(S))
                fprintf(izlaz, "%c", pop(&S));
            fprintf(izlaz, "%c", ch);
        }
        else
            push(&S, ch);
    }
    while (!isempty(S))
        fprintf(izlaz, "%c", pop(&S));
    fclose(izlaz);
    fclose(ulaz);

    return 0;
}
```

43. Napisati algoritam za proveru ispravnosti postavljenih zagrada u fajlu. Koristiti stek kao strukturu podataka za čuvanje otvorenih zagrada.

```

function ZAGRADE(ime_ulaza)
  ▷ Proverava ispravnost postavljenih zagrada. Vraća retcode :
  ▷ = 0 - zagrada ispravno postavljene
  ▷ = 1 - otvorena ch1 do kraja nije zatvorena
  ▷ = 2 - zatvorena ch1 prethodno nije otvorena
  ▷ = 3,4,5 - zatvorena ch neuparena sa otvorenom ch1 = (, [, {
  ulaz = open(ime_ulaza); MAKENULL(S); retcode ← 0
  while ¬eof(ulaz) do
    ch ← fgetc(ulaz)
    if (ch = ' ')|(ch = ' '|(ch = '{') then
      PUSH(S,ch)
    else if (ch = ')')|(ch = ']')|(ch = '}') then
      if ISEMPY(S) then
        retcode ← 2; ch1 ← ch
        break
      else
        ch1 ← POP(S)
        if (ch = ')')&(ch1 ≠ ' (') then
          retcode ← 3
          break
        else if (ch = ']')&(ch1 ≠ '[') then
          retcode ← 4
          break
        else if (ch = '}')&(ch1 ≠ '{') then
          retcode ← 5
          break
        end if
      end if
    end if
  end while
  if ¬retcode then
    if ISEMPY(S) then
      retcode ← 0; ch1 ← \0
    else
      retcode ← 1; ch1 ← TOP(S); CLEAR(S)
    end if
  end if
  fclose(ulaz)
  return retcode,ch1
end function

```

44. Dati implementaciju programa za proveru postavljenih zagrada u programskom jeziku C, koristeći ADT Stack.

zgrade.c

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

int main(void)
{
    FILE *ulaz;
    stack S = NULL;
    int retcode = 0;
    listdata ch, ch1;

    makenull(&S);
    ulaz = fopen("ulaz.txt", "r");
    if (ulaz == NULL){
        retcode = 6;
    }
    while( !retcode && !feof(ulaz) ){
        ch = fgetc(ulaz);

        if((ch=='(')||(ch=='[')||(ch=='{')){
            if(push(&S,ch))
                retcode = 7;
        }
        else if((ch==')')||(ch==']')||(ch=='}')){
            if(isempty(S)){
                retcode = 2;
            }
            else{
                ch1 = pop(&S);
                if(((ch==')')&&(ch1!='('))){
                    retcode = 3;
                }
                else if((ch==']')&&(ch1!='['))){
                    retcode = 4;
                }
                else if((ch=='}')&&(ch1!='{')){
                    retcode = 5;
                }
            }
        }
    }
    fclose(ulaz);
    if(!retcode){
        if(isempty(S)){
            retcode = 0;
        }
        else{
            retcode = 1;
            ch1 = top(S);
        }
    }
    clear(&S);

    switch (retcode){
        case 0:
            printf("\nZagrade_su_dobro_postavljene!\n");
            break;
        case 1:
            printf("\nOtvorena_je_zagrada_%c_koja_do_kraja_fajla_nije_zatvorena!\n", ch1);
            break;
        case 2:
            printf("\nZatvorena_je_zagrada_%c_koja_pre_toga_nije_otvorena!\n", ch);
            break;
        case 6:
            printf("\nNe_moze_da_se_otvori_fajl_ulaz.txt!\n");
            break;
        case 7:
            printf("\nNema_dovoljno_memorije!\n");
            break;
        default:
            printf("\nZagrade_nisu_dobro_postavljene:\n");
            printf("otvorena_zagrada_%c_je_uparena_sa_zatvorenom_zagradom_%c!\n", ch1, ch);
            break;
    }

    return retcode;
}
```


45. Dati implementaciju ADT Stack u programskom jeziku C pomoću niza.

Zbog kompatibilnosti interfejsa sa rešenjem ADT Stek preko povezanih lista zadržali smo isti heder fajl, tako da u istim situacijama prenosimo pokazivač na poslednji ubačen čvor kao parametar *S*, odnosno adresu tog pokazivača kao parametar (**S*).

Tako je stack pokazivač na strukturu koja se iz razloga kompatibilnosti zove node u kojoj se pamte elementi steka i dužina steka. Dužina je u polju count i ona je faktički za jedan manja od stvarne dužine steka. U polju node pamtimo niz data maksimalne dužine MAXS u koji smeštamo elemente steka.

Inicijalizacija steka (makenull) se vrši alokacijom memorije za node i potom alokacijom memorije za data sadržaj steka. Na ovaj način je teoretski moguće povećati kapacitet steka realokacijom memorije.

stack.h

```
typedef char listdata;
typedef struct _node node;
typedef node *stack;

void makenull(stack *);
int isempty(stack);
int push(stack *, listdata);
listdata pop(stack *);
listdata top(stack);
void clear(stack *);
int ismember(stack, listdata *);
void printstack(stack);
```

stack.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "stack.h"
#define MAXS 1000

struct _node
{
    listdata *data;
    int count;
};

int isequal(listdata * lhs, listdata * rhs)
{
    return !memcmp(lhs, rhs, sizeof(listdata));
```

```
}
void makenull(stack *S)
{
    (*S)=malloc(sizeof(node));
    (*S)->data=malloc(sizeof(listdata)*MAXS);
    (*S)->count = -1;
}

int isempty(stack S)
{
    return (int) (S->count== -1);
}

int push(stack *S, listdata d)
{
    if ((*S)->count+1==MAXS)
        return 1;
    (*S)->data [(*S)->count+1] = d;
    (*S)->count = (*S)->count + 1;
    return 0;
}

listdata pop(stack *S)
{
    listdata d = (*S)->data [(*S)->count];
    (*S)->count = (*S)->count - 1;
    return d;
}

listdata top(stack S)
{
    listdata d = S->data[S->count];
    return d;
}

void clear(stack *S)
{
    free ((*S)->data);
    free (*S);
}

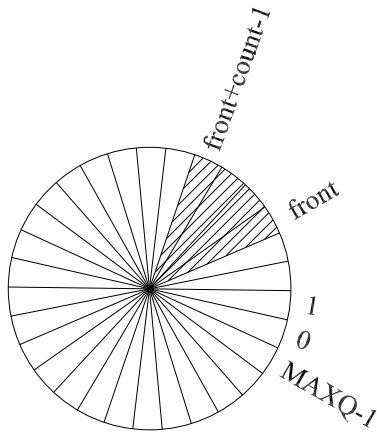
int ismember(stack S, listdata *ch)
{
    int i;
    for (i=S->count; i >=0; i--)
    {
        if (isequal(&(S->data[i]), ch))
            return 1;
    }
    return 0;
}

void printstack(stack S)
{
    int i;
    for (i=S->count; i >=0; i--)
    {
        printf ("%c\n", S->data[i]);
    }
    printf ("\n");
}
```

46. Napisati algoritme za implementaciju ADT Queue¹¹, pomoću niza (array).

Elemente reda ćemo smestiti u niz susednih memorijskih lokacija, tzv. array¹². Ovaj ADT treba da omogući dodavanje elemenata na početak reda i skidanje sa kraja. Pošto se to može naizmenično dešavati, početak i kraj reda se mogu pomeriti prema kraju niza.

Da izbegnemo pomeranje elemenata u memoriji, omogućavamo "premotavanje" elemenata u kružnoj strukturi, kao na slici:



Treba paziti da kad kraj reda pređe preko $MAXQ =$ maksimalnog rezervisanog broja elemenata niza, da se za kraj reda uzme ostatak pri deljenju sa $MAXQ$.

Red je pun kad je kraj za jedan manji od početka. Pošto je to takođe stanje praznog reda, odlučujemo da umesto reprezentacije početni - krajnji ($front - rear$) element, koristimo reprezentaciju početni - broj ($front - count$), gde se krajnji element nalazi na rednom broju $(front + count) \% MAXQ$.

```
struct _queue {
    listdata data[MAXQ];
    int front;
    int count;
};
```

Pakujemo u strukturu queue:

- data, niz od $MAXQ$ elemenata,
- front, redni broj početnog elementa
- count, broj elemenata.

U pseudo kodu numeracija elemenata niza kreće od 1, ovde ćemo, radi lakše implementacije u C-u, numerisati od 0.

```
procedure MAKENULLQ(*Q)
```

```
    Q.front ← 0
```

```
    Q.count ← 0
```

```
end procedure
```

```
function ISEMPYQ(Q)
```

```
    return ¬Q.count
```

```
end function
```

```
procedure ENQUEUE(Q,d)
```

```
    rear ← (Q.front + Q.count)%MAXQ
```

```
    Q.data[rear] ← d
```

```
    Q.count ← Q.count + 1
```

```
end procedure
```

```
function DEQUEUE(Q)
```

```
    d ← Q.data[Q.front]
```

```
    Q.front ← (Q.front + 1)%MAXQ
```

```
    Q.count ← Q.count - 1
```

```
    return d
```

```
end function
```

```
function FRONT(*Q)
```

```
    return Q.data[Q.front]
```

```
end function
```

```
function ISMEMBERQ(*Q,d)
```

```
    for i ← front to front + count - 1 do
```

```
        if Q.data[i%MAXQ] = d then
```

```
            return 1
```

```
        end if
```

```
    end for
```

```
    return 0
```

```
end function
```

```
procedure CLEARQ(*Q)
```

```
    free(Q)
```

```
end procedure
```

¹¹queue = red, EN

¹²array = poredak, red, EN

47. Dati implementaciju ADT Queue u programskom jeziku C pomoću niza.

queue.h

```
typedef char listdata;
typedef struct _node node;
typedef struct _queue *queue;

void makenullQ(queue *);
int isemptyQ(queue);
int enqueue(queue, listdata);
listdata dequeue(queue);
listdata front(queue);
void clearQ(queue);
int ismemberQ(queue, listdata *);
void printqueue(queue);
```

queue.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "queue.h"
#define MAXQ 1000

struct _queue
{
    listdata *data;
    int front;
    int count;
};

typedef struct _queue queues;

int isequal(listdata * lhs, listdata * rhs)
{
    return !memcmp(lhs, rhs, sizeof(listdata));
}

void makenullQ(queue *Q)
{
    (*Q) = malloc(sizeof(queues));
    (*Q)->data = malloc(sizeof(listdata)*MAXQ);
    (*Q)->front = 0;
    (*Q)->count = 0;
}

int isemptyQ(queue Q)
{
    return !(Q->count);
}
```

```
int enqueue(queue Q, listdata d)
{
    if (Q->count+1==MAXQ){
        return 1;
    }
    Q->data [(Q->front+Q->count)%MAXQ] = d;
    Q->count++;
    return 0;
}

listdata dequeue(queue Q)
{
    listdata d = Q->data[Q->front];
    Q->front = (Q->front+1) % MAXQ;
    Q->count--;
    return d;
}

listdata front(queue Q)
{
    listdata d = Q->data[Q->front];
    return d;
}

void clearQ(queue Q)
{
    free(Q->data);
    free(Q);
}

int ismemberQ(queue Q, listdata *chp)
{
    int i;
    for (i=0; i<Q->count; i++)
        if (isequal(
            &(Q->data [(Q->front+i)%MAXQ]),
            chp))
            return 1;
    return 0;
}

void printqueue(queue Q)
{
    int i;
    for (i=0; i<Q->count; i++)
        printf("%c\n",
            Q->data [(Q->front+i)%MAXQ]);
}
```

48. Dati implementaciju ADT Queue u programskom jeziku C pomoću povezane liste.

Za operacije potrebne za rad sa kjuom (*queue*, *EN*) dovoljna je jednostruko povezana lista, kao za stek. Ipak, pošto se elementi dodaju na kraj liste, pored pokazivača na prvi element liste (*front*) čuvamo i pokazivač na pokazivač pretposlednjeg elementa u listi (*rear*). Kad je kju prazan, taj pokazivač pokazuje na *front* pokazivač, a *front* pokazivač je *NULL* pointer.

queue.h

```
typedef char listdata;
typedef struct _node node;
typedef struct _queue *queue;

void makenullQ(queue *);
int isemptyQ(queue);
int enqueue(queue, listdata);
listdata dequeue(queue);
listdata front(queue);
void clearQ(queue);
int ismemberQ(queue, listdata *);
void printqueue(queue);
```

queue.c

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

struct _node
{
    listdata data;
    node *next;
};

struct _queue
{
    node *front;
    node **rear;
};

typedef struct _queue queues;

int isequal(listdata * lhs, listdata * rhs)
{
    return !memcmp(lhs, rhs, sizeof(listdata));
}

void makenullQ(queue *Q)
{
    (*Q) = malloc(sizeof(queues));
    (*Q)->front = NULL;
    (*Q)->rear = &((*Q)->front);
}

int isemptyQ(queue Q)
{
    return (int) (Q->front==NULL);
}
```

```
int enqueue(queue Q, listdata d)
{
    node *N_new = malloc(sizeof(node));

    if (!N_new)
        return (1);

    N_new->data = d;
    N_new->next = NULL;
    *(Q->rear) = N_new;
    Q->rear = &(N_new->next);
    return 0;
}

listdata dequeue(queue Q)
{
    node *N_temp = Q->front;
    listdata d = N_temp->data;
    Q->front = N_temp->next;
    if (!(Q->front))
        Q->rear = &(Q->front);
    free(N_temp);
    return d;
}

listdata front(queue Q)
{
    listdata d = (Q->front)->data;
    return d;
}

void clearQ(queue Q)
{
    while (!isemptyQ(Q))
        dequeue(Q);
    free(Q);
}

int ismemberQ(queue Q, listdata *chp)
{
    node *N = Q->front;
    while (N){
        if (isequal(&(N->data), chp))
            return 1;
        N = N->next;
    }
    return 0;
}

void printqueue(queue Q)
{
    node *N = Q->front;
    while (N)
    {
        printf("%c\n", N->data);
        N = N->next;
    }
}
```

49. Napisati u programskom jeziku C program koji, koristeći ADT Queue, iz fajla ulaz.txt u fajl izlaz.txt prepisuje prvo pojavljivanje karaktera redom kojim se pojavljuju (ignoriše duplikate).

main.c

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

int main(void)
{
    FILE *ulaz;
    FILE *izlaz;
    queue Q;
    listdata ch;

    ulaz = fopen("ulaz.txt", "r");
    izlaz = fopen("izlaz.txt", "w");

    makenullQ(&Q);

    while((ch=fgetc(ulaz)) != EOF)
        if(!ismemberQ(Q,&ch))
            if(enqueue(Q,ch)){
                printf("\n_Nema_dovoljno_memorije!\n");
                clearQ(Q);
                fclose(izlaz);
                fclose(ulaz);
                return 1;
            }

    while(!isemptyQ(Q)){
        fprintf(izlaz, "%c", dequeue(Q));
    }

    clearQ(Q);

    fclose(izlaz);
    fclose(ulaz);

    return 0;
}
```

3 Grafovi

Graf G je uređena trojka $(V(G), E(G), \psi_G)$, gde je

- $V(G)$ neprazan skup **čvorova** (*vertices*, EN),
- $E(G)$ skup **grana** (*edges*, EN), $V(G) \cap E(G) = \emptyset$,
- ψ_G **funkcija incidencije** (*incidence*, EN), koja svakoj grani pridružuje neuređen par (neobavezno različitih) čvorova.

Ako funkcija ψ_G granama pridružuje uređene parove (neobavezno različitih) čvorova, kažemo da je graf **orijentisan**, odnosno **digraf**.

Kažemo da je grana uv **incidentna** čvorovima u i v , u i v su njeni **krajevi**, a ako je graf orijentisan, onda je u **početak** i v **kraj** orijentisane grane uv .

Ako ne naglasimo, posmatramo neusmerene grafove.

Grane su **paralelne** ako su incidentne istim čvorovima.

Ako graf ima paralelnih grana kažemo da je **multigraf**.

Ako je $u = v$, grana uv je **petlja**.

Graf je **prost** ako nema paralelnih grana ni petlji.

Ako drugačije ne naglasimo, kad kažemo graf, mislimo na prost graf.

Čvorovi su **susedni** ako postoji grana kojoj su incidentni.

Susedstvo (*adjacency*, EN) čvora u je skup $\text{Adj}(u)$ svih čvorova sa kojima je u susedan.

Stepen čvora $d(u)$ je broj njegovih suseda. Naravno: $\sum_{u \in V(G)} d_G(u) = 2|E(G)|$.

Kompletan graf K_n je graf sa n čvorova čija su svaka dva čvora susedna.

Kažemo da je graf **bipartitan** ako se skup čvorova može podeliti na dve neprazne disjunktne klase tako da čvorovi u jednoj klasi nisu međusobno susedni. Ako su pritom čvorovi iz jedne klase susedni sa svim čvorovima iz druge klase, kažemo da je to **kompletan bipartitan** graf $K_{m,n}$ gde je m broj čvorova jedne, a n broj čvorova druge klase.

Graf H je **podgraf** grafa G ako je $V(H) \subseteq V(G)$ i $E(H) \subseteq E(G)$. Tada je G **nadgraf** od H . Ako je, pritom, $V(H) = V(G)$, G je **pokrivajući nadgraf** (H je **pokrivajući podgraf**).

Podgraf je **indukovan** podskupom čvorova, odnosno grana, ako mu pripadaju sve odgovarajuće grane, odnosno čvorovi.

Šetnja kroz graf je konačan, neprazan niz $W = v_0 e_1 v_1 e_2 v_2 \dots e_k v_k$ u kojem se smenjuju čvorovi i grane grafa G , i čvorovi v_{i-1} i v_i su krajevi grane e_i . Kažemo da ova šetnja W spaja čvorove v_0 i v_k , pišemo $v_0 \rightsquigarrow v_k$.

Staza je šetnja u kojoj se grane ne ponavljaju.

Put je staza u kojoj se čvorovi ne ponavljaju.

Dva čvora su **povezana** ako postoji put koji ih povezuje.

Povezanost čvorova je relacija ekvivalencije, klase ekvivalencije su **komponente povezanosti**. Graf je **povezan** ako ima tačno jednu komponentu povezanosti.

Ako se početni i krajnji čvor šetnje, staze, puta, poklapaju, kažemo da je **zatvorena šetnja, staza, put**. Zatvorena staza je **kontura**.

Kontura koja sadrži sve grane grafa je **Ojlerova kontura**. Graf koji ima **Ojlerovu konturu** je **Ojlerov graf**. Graf u kome postoji staza koja sadrži sve grane grafa zove se **Polu Ojlerov graf**.

Netrivijalni graf (ili multigraf) bez izolovanih čvorova je Ojlerov ako i samo ako je povezan i svaki čvor je parnog stepena.

Put koji sadrži sve čvorove grafa je **Hamiltonov put**, graf koji ima Hamiltonov put je **Polu Hamiltonov graf**. Ako graf ima zatvoreni Hamiltonov put, onda je **Hamiltonov**.

Graf sa n čvorova ($n \geq 3$) u kome za svaka dva susedna čvora u i v važi $d_G(u) + d_G(v) \geq n$ je Hamiltonov graf.

Graf koji nema konturu je **acikličan**.

Za graf sa skupom čvorova $V(G) = \{v_1, \dots, v_m\}$ **matrica susedstva** (*adjacency matrix*, EN) je $M_G = [m_{i,j}]$, gde je

$$m_{i,j} = \begin{cases} 1, & \text{ako su } v_i \text{ i } v_j \text{ susedni,} \\ 0, & \text{ako } v_i \text{ i } v_j \text{ nisu susedni.} \end{cases}$$

Lista susedstva (*adjacency list*, EN) je niz $\text{Adj}(v_i)$, $i = 1, \dots, m$ susedstava čvorova v_i .

Povezan, acikličan graf je **drvo** (*tree*, EN). Acikličan graf je **šuma** (*forest*, EN).

Svaka dva čvora drveta su povezana jedinstvenim putem.

Čvor u drvetu je **viseći** ako je njegov stepen 1. ($d_G(v_i) = 1$)

Netrivijalno drvo sadrži bar dva viseća čvora. Drvo sa m čvorova ima $m - 1$ grana.

Ako je drvo T pokrivaјуći podgraf grafa G , kažemo da je T **pokrivaјуće drvo** grafa G .

Graf ima pokrivaјуće drvo ako i samo ako je povezan.

Orijentisani grafovi

Za orijentisane grafove se analogno neorijentisanim mogu definisati **orijentisana šetnja, staza, put**, poštuјуći orijentaciju.

Orijentisani graf je **orijentisano drvo** ako se ignorisanjem orijentacije dobije drvo. Ako su pritom sve grane usmerene od jednog čvora, kažemo da je to **korensko drvo** i da je čvor od koga su grane orijentisane **koren**. Za orijentisanu granu (u, v) u korenskom drvetu kažemo da je čvor u **roditelj (ili otac)** (*parent*, EN) a v je **dete** (*child*, EN). Korensko drvo u kome svaki čvor ima najviše dva deteta zovemo **binarno drvo**.

Čvorovi u i v su **jako povezani** ako postoje putevi koji spajaju $u \rightsquigarrow v$ i $v \rightsquigarrow u$. Jaka povezanost je relacija ekvivalencije. Njene klase ekvivalencije zovemo **komponente jake**

povezanosti. Jaka povezanost indukuje **graf jako povezanih komponenti:** čvorovi su komponente, grane između komponenti postoje ako postoji grana između elemenata.

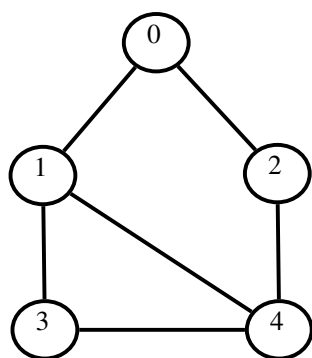
Orijentisani graf je **jako povezan** ako ima tačno jednu komponentu.

Orijentisani aciklični graf (*DAG = Directed Acyclic Graph*, EN) se može **topološki sortirati**, odnosno: čvorovi grafa se mogu poređati u niz tako da ako postoji grana (u, v) , onda, u nizu, u prethodi v .

Strukture podataka za grafove

Matrica susedstva

Graf se u memoriju kompjutera može smestiti u matricu susedstva koja sadrži nule i jedinice. Za prost graf to je simetrična matrica sa nulama na glavnoj dijagonali, pa se radi uštede memorije može koristiti samo gornji trougao. Ako se za graf žele u memoriji smestiti težine grana, može se koristiti matrica koja na mestu (i, j) ima težinu grane između čvorova v_i i v_j , a ako ne postoji grana: specijalni džoker simbol čija vrednost nije dopustiva za težinu grane.



Graf

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Matrica susedstva

u	Adj(u)
0	1, 2
1	0, 3, 4
2	0, 4
3	1, 4
4	1, 2, 3

Lista susedstva

Lista susedstva

Susedstvo $Adj(u)$ čvora u se može čuvati u povezanoj listi. Ako želimo sačuvati redosled dodavanja čvorova u graf u povezanoj listi susedstva, korišćemo tehnologiju dodavanja elemenata u queue: proceduru `enqueue` iz zadatka 48.

Lista susedstva se za graf najčešće pamti ne kao lista listi, već kao **niz** pokazivača na povezane liste suseda za svaki čvor. Na taj način se može brzo doći do svih suseda nekog čvora.

Binarna drva

Binarno drvo u memoriji računara možemo čuvati kao nizove *LC* i *RC* adresa levog i desnog deteta svakog čvora (*LC* = left child, *RC* = right child, *EN*). Koristi se poseban *NULL* simbol ako neki čvor nema levo odnosno desno dete. Pored toga, posebno se čuva adresa korena i niz *K* ključeva (*keys*, *EN*), sa adresama informacionog sadržaja svakog čvora.

50. U programskom jeziku C napisati proceduru koja koristeći sledeći tip podataka za graf

```
typedef struct _node gnode;
typedef gnode *grana;
typedef int nextnode;

struct _node
{
    nextnode data;
    gnode *next;
};
```

- dodaje granu na kraj liste susedstva: enqueue_list,
- oduzima prvu granu iz liste susedstva: dequeue_list,
- oslobađa dinamički alociranu memoriju jedne liste susedstva: clear_list,
- oslobađa dinamički alociranu memoriju grafa: clear_graf,
- štampa tabelu liste susedstva: print_graf,
- štampa matricu susedstva: print_graf_matrix.

```
void enqueue_list(grana **grana_tail_p, nextnode d) {
    grana grana_new = malloc(sizeof(gnode));

    grana_new->data = d;
    grana_new->next = NULL;
    **grana_tail_p = grana_new;
    *grana_tail_p = &(grana_new->next);
}

nextnode dequeue_list(grana *grana_head)
{
    grana grana_temp = *grana_head;
    nextnode d=-1;
```

```

    if (grana_head)
    {
        d = grana_temp -> data;
        *grana_head = grana_temp -> next;
        free (grana_temp);
    }
    return d;
}

void clear_list (grana *adjp)
{
    while (*adjp)
        dequeue_list (adjp);
}

void clear_graph (grana graf[])
{
    int i;
    for (i=0; i<max_cvorova; i++){
        clear_list (&(graf[i]));
    }
}

void print_graph (grana G[], int n)
{
    int i;
    grana gr;
    printf ("\nCvor_ | Adj (Cvor): ");
    printf ("\n-----+\n");
    for (i=0; i<n; i++){
        printf ("%4d | ", i);
        gr = G[i];
        while (gr){
            printf (" %2d, ", gr->data);
            gr = gr->next;
        }
        printf ("\n");
    }
    printf ("-----+\n");
}

void print_graph_matrix (unsigned char M[], int n)
{
    int i, j;
    printf ("\n | ");

```

```

    for (j=0;j<n;j++)
        printf("%2d_", j);
    printf("\n");
    printf("——+");
    for (j=0;j<n;j++)
        printf("——");
    printf("\n");
    for (i=0;i<n;i++){
        printf("%2d_|", i);
        for (j=0;j<n;j++){
            printf("_%1d_", M[i*n+j]);
        }
        printf("\n");
    }
}

```

51. U programskom jeziku C napisati proceduru koja transponuje graf koristeći proceduru enqueue_list iz zadatka 50.

```

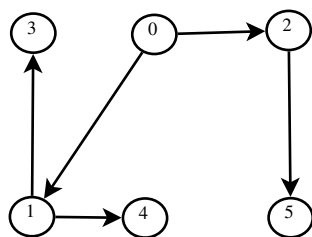
void transpose_graph(grana G[], int n, grana GT[])
{
    int i, j;
    grana gr;
    grana *rear[max_cvorova];

    for (i=0;i<max_cvorova;i++){
        GT[i] = NULL;
        rear[i] = &(GT[i]);
    }

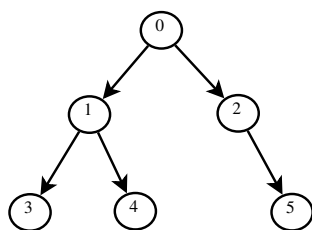
    for (i=0;i<n;i++){
        gr = G[i];
        while (gr){
            j = gr->data;
            enqueue_list(&(rear[j]), i);
            gr = gr->next;
        }
    }
}

```

52. Koristeći procedure iz zadatka 50 i 51 u programskom jeziku C napisati glavni program koji ubacuje graf sa slike u memoriju, štampa listu susedstva, transponuje graf, oslobađa memoriju originalnog grafa, štampa listu susedstva transponovanog grafa i oslobađa memoriju transponovanog grafa.



Graf



Isti graf kao korensko drvo

u	$Adj(u)$
0	1, 2
1	3, 4
2	5

Lista susedstva

```

int main(void)
{
    grana G[max_cvorova],GT[max_cvorova];
    int i,n;
    for(i=0;i<max_cvorova;i++){
        G[i] = NULL;
    }
    grana *rear [max_cvorova];

    for(i=0;i<max_cvorova;i++){
        rear[i] = &(G[i]);
    }
    enqueue_list(&rear [0],1); enqueue_list(&rear [0],2);
    enqueue_list(&rear [1],3); enqueue_list(&rear [1],4);
    enqueue_list(&rear [2],5); n = 6;
    print_graph(G,n);
    transpose_graph (G, n, GT);
    clear_graph(G);
    print_graph (GT,n);
    clear_graph(GT,n);
    return 0;
}

```

53. Napisati u programskom jeziku C proceduru `adjlist2adjmatrix` koja zapis grafa datog u listi susedstva (Adjacency list) pretvara u zapis matrice susedstva (Adjacency matrix). U matrici susedstva koja je formata $|V| \times |V|$ stavljamo 1 u u -toj vrsti i v -koloni ako postoji grana koja spaja čvorove u i v , inače 0. Napisati i proceduru `adjmatrix2adjlist` koja sa istim ulaznim parametrima radi obrnuto. Dati glavni program koji će testirati rešenje na grafu sa slike na strani 32.

Zadate procedure se mogu pozvati radi testiranja iz programa kao što je ovaj:

```

int main(void) {
    grana G[max_cvorova];
    grana *rear [max_cvorova];
    int n;
    nextnode u;
    unsigned char M[max_cvorova*max_cvorova];
    for(u=0;u<max_cvorova;u++){
        G[u] = NULL;
        rear[u] = &(G[u]);
    }
    enqueue_list(&rear [0],1);
    enqueue_list(&rear [0],2);
    enqueue_list(&rear [1],0);
    enqueue_list(&rear [1],3);
    enqueue_list(&rear [1],4);
    enqueue_list(&rear [2],0);

```

```

    enqueue_list(&rear [2],4);
    enqueue_list(&rear [3],1);
    enqueue_list(&rear [3],4);
    enqueue_list(&rear [4],1);
    enqueue_list(&rear [4],2);
    enqueue_list(&rear [4],3);
    n = 5; // broj cvorova

```

```

    print_graph(G,n);
    adjlist2adjmatrix (G,n,M);
    print_graph_matrix (M,n);
    clear_graph(G);
    adjmatrix2adjlist (G,n,M);
    print_graph(G,n);
    return 0;
}

```

Graf se smešta u niz povezanih listi, matricu smo, na uobičajeni način, smestili po vrstama u niz M veličine $\text{max_cvorova} \times \text{max_cvorova}$.

<pre>void adjlist2adjmatrix(grana G[], int n, unsigned char M[]) { int i; grana gr; for(i=0;i<n*n;i++) M[i] = 0; for(i=0;i<n;i++){ gr = G[i]; while(gr){ M[i*n+gr->data] = 1; gr = gr->next; } } }</pre>	<pre>void adjmatrix2adjlist(grana G[], int n, unsigned char M[]) { int i, j; grana *grp; for(i=0;i<n;i++) { grp = &(G[i]); for(j=0;j<n;j++) if(M[i*n+j]) enqueue_list(&grp, j, 0); } }</pre>
--	---

Kada se pozove dati program, njegov izlaz će biti:

```
Cvor | Adj(Cvor):
-----+-----
0 | 1, 2,
1 | 0, 3, 4,
2 | 0, 4,
3 | 1, 4,
4 | 1, 2, 3,
```

```
 | 0 1 2 3 4
-----+-----
0 | 0 1 1 0 0
1 | 1 0 0 1 1
2 | 1 0 0 0 1
3 | 0 1 0 0 1
4 | 0 1 1 1 0
```

```
Cvor | Adj(Cvor):
-----+-----
0 | 1, 2,
1 | 0, 3, 4,
2 | 0, 4,
3 | 1, 4,
4 | 1, 2, 3,
```

Process returned 0 (0x0) execution time : 0.016 s
Press any key to continue.

54. Za graf sa V čvorova i E grana analizirati asimptotsku složenost procedura `adjlist2adjmatrix` i `adjmatrix2adjlist` iz zadatka 53.

Slično kao u zadatku 61, dobijamo:

za `adjlist2adjmatrix`: $T_1(E, V) = \Theta(V^2)$,

za `adjmatrix2adjlist`: $T_2(E, V) = \Theta(V^2)$.

55. Dati algoritam za pretraživanje grafa uskladištenog u listi susedstva (*adjacency*¹³ list) "u širinu" (*breadth*¹⁴ first search = BFS).

- BFS polazi od izvora: čvor s i prolazi kroz sve čvorove koji su povezani sa s .
- BFS nalazi d , (najkraću) udaljenost od s za svaki čvor, $d = \infty$ ako nije povezan.
- BFS nalazi π , prethodnika u najkraćem putu, dajući "breadth first tree".
- BFS koristi atribut boja (*color*) $\in \{ \text{WHITE, GRAY, BLACK} \}$ za svaki čvor.
- BFS redom otkriva sve čvorove koji su od s udaljeni za k , a potom za $k + 1$.

```

function BFS( $G, s$ )
  ▷ Pretražuje graf  $G$  "u širinu" (breadth first)
  ▷ U nizu  $d$  vraća izračunatu udaljenost od čvora  $s$ 
  ▷ U nizu  $\pi$  vraća prethodnika za čvor
  for each  $u \in V[G] \setminus \{s\}$  do
     $d[u] \leftarrow \infty$ 
     $\pi[u] \leftarrow \text{NULL}$ 
     $color[u] \leftarrow \text{WHITE}$ 
  end for
  MAKENULLQ( $Q$ )
   $d[s] \leftarrow 0$ 
   $\pi[s] \leftarrow \text{NULL}$ 
   $color[s] \leftarrow \text{GRAY}$ 
  ENQUEUE( $Q, s$ )
  while  $\neg \text{ISEMPTYQ}(Q)$  do
     $u \leftarrow \text{DEQUEUE}(Q)$ 
    for each  $v \in \text{Adj}(u)$  do
      if  $color[v] = \text{WHITE}$  then
         $color[v] \leftarrow \text{GRAY}$ 
         $d[v] \leftarrow d[u] + 1$ 
         $\pi[v] \leftarrow u$ 
        ENQUEUE( $Q, v$ )
      end if
    end for
     $color[u] \leftarrow \text{BLACK}$ 
  end while
  return  $d, \pi$ 
end function

```

¹³*adjacency* = susedstvo, EN

¹⁴*breadth* = širina, EN

56. Dati implementaciju u programskom jeziku C procedure BFS iz zadatka 55.

```

#include <stdio.h>
#include <stdlib.h>
#include "queue.h"
#define max_cvorova 50

typedef int cvor;
typedef struct _node gnode;
typedef gnode *grana;

struct _node
{
    cvor data;
    gnode *next;
};

int main(void)
{
    grana G[max_cvorova];
    int p[max_cvorova];
    int i, n, retcode;

    n = ucitaj_graf(G);
    print_graph(G, n);

    bfs(G, n, 1, d, p);
    path(p, 1, 7);
    printf("\n\n");
    for(i=0; i<n; i++){
        printf("d[%2d] = %2d, p[%2d] = %2d\n",
            i, d[i], i, p[i]);
    }

    clear_graph(G);
    return 0;
}

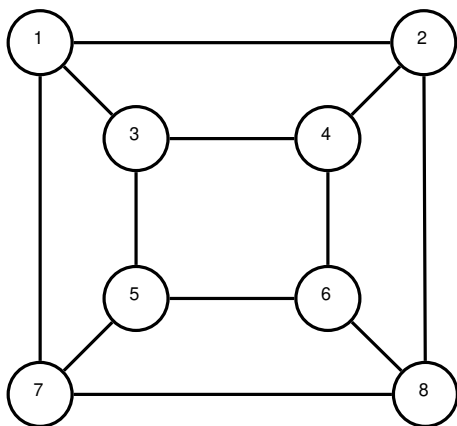
int bfs(grana G[], int n, int s, int d[], int p[])
{
    grana gr;
    queue Q;
    int i, color[max_cvorova];
    cvor u, v;

    for(i=0; i<n; i++){
        d[i] = INT_MAX; //infinity
        p[i] = -1; //no predecessor
        color[i] = 0; //WHITE
    }
    makenullQ(&Q);
    d[s] = 0;
    p[s] = -1;
    color[s] = 1; //GRAY
    enqueue(Q, s);
    while(!isemptyQ(Q)){
        u = dequeue(Q);
        gr = G[u];
        while(gr){
            v = gr->data;
            if(!color[v]){
                color[v] = 1; //GRAY
                d[v] = d[u] + 1;
                p[v] = u;
                enqueue(Q, v);
            }
            gr = gr->next;
        }
        color[u] = 2; //BLACK
    }
    clearQ(Q);
    return 0;
}

```

57. Za graf sa slike dole napisati reprezentaciju listama susedstva, držati se leksikografskog redosleda.

Primeniti na isti graf BFS algoritam polazeći od čvora 1, dati tabelu udaljenosti (broj koraka) od čvora 1 i dati prethodnika u najkraćoj putanji ka čvoru 1.



v	$Adj(v)$	$d(v)$	$p(v)$
1	2, 3, 7	0	-
2	1, 4, 8	1	1
3	1, 4, 5	1	1
4	2, 3, 6	2	2
5	3, 6, 7	2	3
6	4, 5, 8	3	4
7	1, 5, 8	1	1
8	2, 6, 7	2	2

58. Napisati u programskom jeziku C proceduru path koja bi se u zadatku 56 pozvala posle procedure bfs i ispisala putanju od proizvoljnog čvora do čvora broj 1.

<pre> void path(int p[], int dovde, int odavde) { if(p[odavde]==-1){ printf("\nNo path!\n"); return; } printf("\n%2d-> ", odavde); while(!(p[odavde]==dovde)){ printf("%2d-> ", p[odavde]); odavde = p[odavde]; } printf("%2d", dovde); } </pre>	<pre> void path1(int p[], int dovde, int odavde) { if(p[odavde]==-1){ printf("\nNo path!\n"); return; } if(p[odavde]==dovde){ printf("\n%2d-> %2d", dovde, odavde); return; } else{ path1(p, dovde, p[odavde]); } printf("-> %2d", odavde); } </pre>
---	---

59. Napisati u programskom jeziku C proceduru path1 koja bi se u zadatku 56 pozvala posle procedure path sa istim ulaznim parametrima i ispisala putanju od čvora broj 1 do proizvoljnog čvora. (Rešenje zadatka 58 unazad.)

Pošto program treba da iskoristi listu prethodnika istu kao i program zadatka 58, rešenje možemo realizovati primenom steka za obrtanje redosleda ispisa, ili elegantnije, iskoristiti rekurziju. Pored rešenja zadatka 58, gore, data je realizacija rekurzijom.

60. Napisati u Programskom jeziku C proceduru koja za graf smešten u Adjacency list vraća stepen svih čvorova i funkciju koja vraća diameter grafa. Stepen je broj suseda, diameter je najveće najkraće rastojanje između dva čvora u grafu.

<pre> 1 void stepen(grana G[], int n, int s[]) 2 { 3 int i; 4 5 grana gr; 6 7 for(i=0;i<n;i++){ 8 gr = G[i]; 9 s[i] = 0; 10 while(gr){ 11 (s[i])++; 12 gr = gr->next; 13 } 14 } 15 16 } </pre>	<pre> int diameter(grana G[], int n) { int diam = 0; int d[max_cvorova], p[max_cvorova]; int i, j; for(i=0;i<n;i++){ bfs(G, n, i, d, p); for(j=0;j<n;j++){ if(d[j]>diam){ diam = d[j]; } } } return diam; } </pre>
---	--

61. Ispitati asimptotski red složenosti procedure stepen iz zadatka 60.

Standardno sa c_k obeležavamo vreme potrebno da se izvrši linija k . Neka graf ima n čvorova i m grana. Za neusmereni graf, kod koga se grana (i, j) smešta i u listu susedstva čvora v_i i listu susedstva čvora v_j , vreme izvršavanja procedure stepen je $T(n, m) =$

$$= c_4 + c_5 + (n + 1) \cdot c_7 + n \cdot c_8 + n \cdot c_9 + \sum_{i=0}^{n-1} (|Adj(v_i)| + 1) c_{10} + \sum_{i=0}^{n-1} |Adj(v_i)| (c_{11} + c_{12}),$$

gde je $|Adj(v_i)|$ broj suseda čvora v_i . Kako je $\sum_{i=0}^{n-1} |Adj(v_i)| = 2m$, sledi

$$T(n, m) = n(c_7 + c_8 + c_9 + c_{10}) + 2m(c_{10} + c_{11} + c_{12}) + c_4 + c_5 + c_7 = \Theta(n + m).$$

Standardno se obeležava broj čvorova grafa V , a broj grana E , složenost algoritma stepen je $\Theta(V + E)$.

62. Analizirati asimptotsku složenost BFS algoritma.

Označavamo broj čvorova grafa V i broj grana grafa E .

Priprema praznih listi koje se vrše za sve čvorove traje $\Theta(V)$, priprema za petlju $\Theta(1)$.

Pozivanje operacije ENQUEUE i DEQUEUE traje $\Theta(1)$. Pošto svaki povezan čvor prođe kroz queue, ukupno se za te operacije potroši $O(V)$ vremena.

Elementi listi susedstva se u algoritmu obrade najviše jednom, kad se čvor skida sa queue.

Za njihovu obradu treba $O(E)$ vremena, jer je njihov broj $\Theta(E)$.

Ukupno vreme izvršavanja BFS algoritma je $O(V + E)$.

63. Napisati algoritam za pretraživanje grafa uskladištenog u liste susedstva "u dubinu" (*depth*¹⁵ *first search* = DFS).

- DFS prolazi kroz sve čvorove u koje nije posetio.
- DFS rekurzivno nastavlja kroz sve grane čiji su susedi v neistraženi.
- kad DFS istraži sve čvorove koji su susedi od v , backtrack¹⁶ postupkom se vraća u čvor iz kojeg je stigao u v .
- kad DFS istraži sve grane iz polaznog čvora, nastavlja sa neistraženim čvorovima.
- kad DFS dođe od čvora u do čvora v , upisuje da je predecessor¹⁷ od v čvor u .
- DFS koristi atribut boja (*color*) $\in \{ \text{WHITE, GRAY, BLACK} \}$ za svaki čvor. U početku su svi WHITE. Kad se otkrije, čvor postaje GRAY, kad završi sa njim, postaje BLACK.
- DFS za svaki čvor u zapisuje *timestamps*¹⁸ $d[u]$ i $f[u]$ ($d[u] < f[u]$) momenta kad je otkrio u (*discovery*) i kad je završio sa u (*finish*).
- Vremenske oznake *timestamps* su iz skupa $\{1, 2, \dots, 2 \cdot |V|\}$.
- Čvor u je WHITE od momenta 1 do $d[u]$, GRAY od $d[u]$ do $f[u]$ i BLACK posle $f[u]$.
- U pseudokodu koji sledi promenljiva *time* je globalna promenljiva. Radi jednostavnosti, za DFS-VISIT globalne promenljive su i G (graf), d, f , kao i π .
- Rezultat primene algoritma zavisi od redosleda kojim su numerisani čvorovi i redosleda kojim su čvorovi uneti u *Adj* liste.

```

function DFS( $G$ )
  for each  $u \in V[G]$  do
     $color[u] \leftarrow \text{WHITE}$ 
     $\pi[u] \leftarrow \text{NULL}$ 
  end for
   $time \leftarrow 0$ 
  for each  $u \in V[G]$  do
    if  $color[u] = \text{WHITE}$  then
      DFS-VISIT( $u$ )
    end if
  end for
  return  $d, f, \pi$ 
end function

```

```

procedure DFS-VISIT( $u$ )
   $color[u] \leftarrow \text{GRAY}$ 
   $time \leftarrow time + 1$ 
   $d[u] \leftarrow time$ 
  for each  $v \in Adj(u)$  do
    if  $color[v] = \text{WHITE}$  then
       $\pi[v] \leftarrow u$ 
      DFS-VISIT( $v$ )
    end if
  end for
   $color[u] \leftarrow \text{BLACK}$ 
   $time \leftarrow time + 1$ 
   $f[u] \leftarrow time$ 
end procedure

```

¹⁵*depth* = dubina, EN

¹⁶*backtrack* = vratiti se istim putem, EN

¹⁷*predecessor* = prethodnik, EN

¹⁸*timestamps* = vremenske oznake, EN

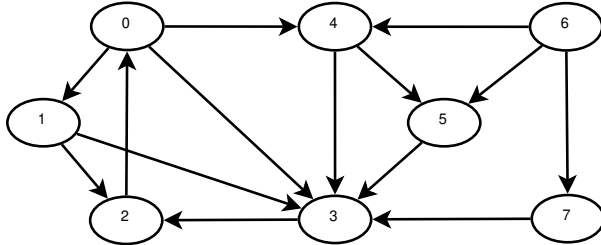
64. Dati implementaciju u programskom jeziku C procedure DFS iz zadatka 63.

Radi elegantnosti, modularnosti i ubrzanja rekurzije procedura DFS i DFS-VISIT se izvajaju u poseban ulazni modul `dfs.c` čiji je heder fajl `dfs.h`. Onda je u C-u moguće realizovati globalne promenljive kao što je urađeno u pseudokodu u zadatku 63.

Pretpostavljamo da su osnovne procedure za definisanje tipa liste susedstva i osnovnih manipulacija sa grafom date u fajlovima `graf.c` i `graf.h`.

graf.h	dfs.c
<pre>typedef int nextnode; typedef struct _node gnode; typedef gnode *grana; struct _node { nextnode data; gnode *next; };</pre>	<pre>#include <stdio.h> #include <stdlib.h> #include "graf.h" #include "dfs.h" #define max_cvorova 50 int n; grana *G; int *d; int *f; int *p; unsigned char color[max_cvorova]; int time; void dfs_visit(int); void dfs(grana G1[], int n1, int d1[], int f1[], int p1[]) { n=n1; G=G1; d=d1; f=f1; p=p1; int u; for(u=0;u<n;u++){ color[u] = 0; //WHITE p[u] = -1; //no predecessor } time = 0; for(u=0;u<n;u++) if(!color[u]) dfs_visit(u); } void dfs_visit(int u) { grana gr; color[u] = 1; //GRAY time++; d[u] = time; gr = G[u]; while(gr){ if(!color[gr->data]){ p[gr->data] = u; dfs_visit(gr->data); } gr = gr->next; } color[u] = 2; //BLACK time++; f[u] = time; }</pre>
dfs.h	
<pre>void dfs(grana [], int, int [], int [], int []);</pre> <p>U heder fajlu <code>dfs.h</code> se prikazuje samo procedura DFS, dok globalne promenljive i rekurzivna procedura DFS-VISIT ostaju lokalne za modul <code>dfs.c</code>.</p> <p>Radi testiranja napisane procedure se mogu pozvati iz glavnog programa kao što sledi.</p>	
main.c	
<pre>#include <stdio.h> #include <stdlib.h> #include "graf.h" #include "dfs.h" #define max_cvorova 50 int main(void) { grana *G[max_cvorova]; int d[max_cvorova], f[max_cvorova], p[max_cvorova]; int n; for(i=0;i<max_cvorova;i++){ G[i] = NULL; } n = ucitaj_graf(G); print_graph(G,n); dfs(G, n, d, f, p); printf("\n\n"); for(i=0;i<n;i++){ printf("d[%2d]=%2d, f[%2d]=%2d, \ p[%2d]=%2d\n", i, d[i], i, f[i], i, p[i]); } clear_graph(G); return 0; }</pre>	

65. Primeniti algoritam DFS na graf sa slike, uzimajući čvorove i grane leksikografski.



U čvorove upisati d i f vrednosti.
 Napraviti tabelu zagrada.
 Opisati šumu ovog DFS.
 Označiti tipove grana kad se prvi put otkriju.

Tipovi grana:

T - tree edge, grana drveta iz DFS šume, pronalazi novi čvor drveta,

F - forward edge, (u,v) je grana unapred ako pronalazi čvor koji već pripada drvetu, t.j. ako je v potomak od u .

B - back edge, (u,v) je grana unazad ako je u je potomak od v .

C - cross edge, poprečne grane, su sve ostale grane.

Na sledećoj slici vidimo levo: DFS šumu i desno: d i f vrednosti u čvorovima i oznaku tipa grana na samim granama.

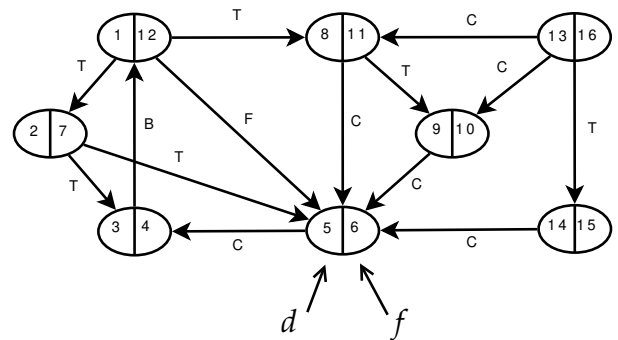
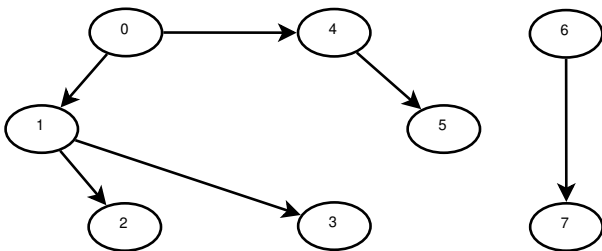


Tabela zagrada:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	()				
1		()									
2			()												
3					()										
4							()								
5								()							
6										()					
7													()		

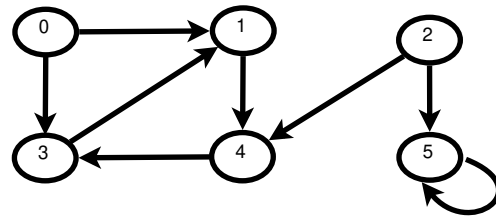
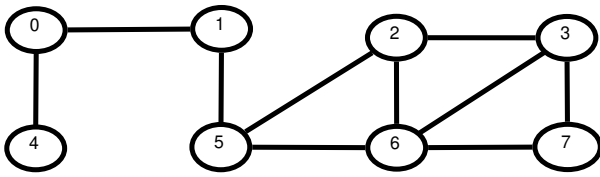
Zagrada se otvara kad se otkrije čvor, zatvara kad se završi sa čvorom.

Za dva različita čvora u i v nije moguće $d(u) < d(v) < f(u) < f(v)$.

Za DFS nekog grafa čvor v je potomak čvora u ako i samo ako $d(u) < d(v) < f(v) < f(u)$.

Za DFS neusmerenog grafa grane (u,v) i (v,u) su ista grana, klasifikuje se po prvom kriterijumu koji zadovolji. Za neusmereni graf sve grane su ili tree edge ili back edge.

66. Sve iz zadatka 65 za graf sa slike dole levo.



67. Sve iz zadatka 65 za graf sa slike gore desno.

68. Modifikovati DFS algoritam tako da za usmerene grafove za svaku granu identifikuje koji je tip.

Vidi rešenje zadatka 74.

69. Opisati potrebne modifikacije iz zadatka 68 za neusmerene grafove.

70. Modifikovati DFS algoritam tako da prebroji povezane komponente i da svakom čvoru pridruži redni broj povezane komponente kojoj pripada.

Vidi rešenje zadatka 74.

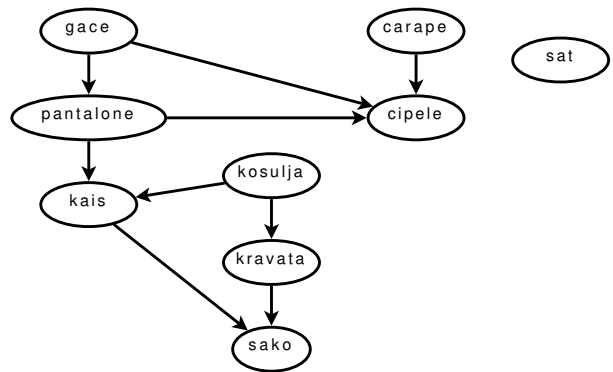
71. Za usmereni graf važi: graf je acikličan \Leftrightarrow proizvoljna DFS šuma nema Back edges. Dokazati.

72. Profesor Rasejanko je napravio graf kojim opisuje koji odevni predmet treba da se obuče pre kojeg.

Izvršiti DFS algoritam na datom grafu i ustanoviti da je acikličan.

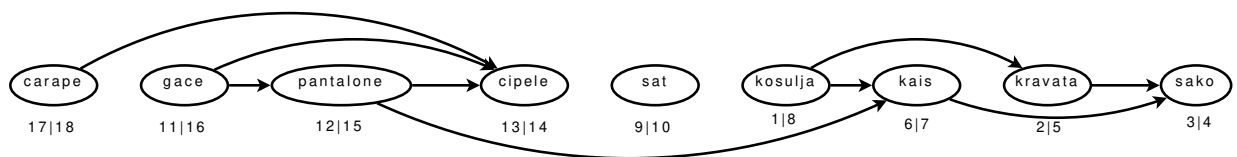
Uraditi topološko sortiranje datog grafa i napraviti plan kojim Profesor treba da se obuče.

Da li je rešenje jedinstveno?



Do rešenja se dolazi kad se primeni DFS algoritam na dati graf. Lako se ustanovljava da u grafu nema *back edges*, pa je acikličan. Potom se *finish* vremena čvorova $f(u)$ sortiraju opadajuće i po tom redosledu se poređaju čvorovi odevnih predmeta.

Dobije se graf kao na slici:



Druga rešenja bi se dobila da su čvorovi i susedi u listama susedstva drugačije poređani.

73. Dokazati da postupak iz zadatka 72 daje topološko sortiranje grafa.

74. Modifikovati DFS algoritam tako da za acikličan graf daje listu topološki sortiranih čvorova.

Ovde dajemo rešenja zadataka 68, 70, 74. Modifikacije su date u odgovarajućoj boji.

Osnova je algoritam DFS sa procedurom DFS-VISIT iz zadatka 63.

```

function DFS( $G$ )
  for each  $(u, v) \in E[G]$  do
     $type[(u, v)] \leftarrow \text{NULL}$ 
  end for
  for each  $u \in V[G]$  do
     $color[u] \leftarrow \text{WHITE}$ 
     $\pi[u] \leftarrow \text{NULL}$ 
  end for
   $time \leftarrow 0$ 
   $dag \leftarrow \text{TRUE}$ 
   $MAKENULL(S)$ 
   $ccounter \leftarrow 0$ 
  for each  $u \in V[G]$  do
    if  $color[u] = \text{WHITE}$  then
      DFS-VISIT( $u$ )
       $ccounter \leftarrow ccounter + 1$ 
    end if
  end for
  if  $dag$  then
    while  $\neg \text{ISEMPTY}(S)$  do
       $WRITELN(\text{POP}(S))$ 
    end while
  else
     $CLEAR(S)$ 
  end if
  return  $d, f, \pi, type, cc, dag$ 
end function

procedure DFS-VISIT( $u$ )
   $color[u] \leftarrow \text{GRAY}$ 
   $cc[u] \leftarrow ccounter$ 
   $time \leftarrow time + 1$ 
   $d[u] \leftarrow time$ 
  for each  $v \in Adj(u)$  do
    if  $type[(u, v)] = \text{NULL}$  then
      if  $color[v] = \text{WHITE}$  then
         $type[(u, v)] \leftarrow \text{TREE}$ 
      else if  $color[v] = \text{GRAY}$  then
         $type[(u, v)] \leftarrow \text{BACK}$ 
         $dag \leftarrow \text{FALSE}$ 
      else if  $color[v] = \text{BLACK}$  then
        if  $d[v] > d[u]$  then
           $type[(u, v)] \leftarrow \text{FORWARD}$ 
        else
           $type[(u, v)] \leftarrow \text{CROSS}$ 
        end if
      end if
    end if
    if  $color[v] = \text{WHITE}$  then
       $\pi[v] \leftarrow u$ 
      DFS-VISIT( $v$ )
    end if
  end for
   $color[u] \leftarrow \text{BLACK}$ 
   $PUSH(S, u)$ 
   $time \leftarrow time + 1$ 
   $f[u] \leftarrow time$ 
end procedure

```

Po završetku, osim vremena otkrića d i završetka f za čvor, prethodnika π u DFS šumi, algoritam vraća i redni broj komponente kojoj čvor pripada cc , tip grane $type$ i tačno/ne-tačno da li je graf acikličan u dag .

Topološko sortiranje se dobija ispisivanjem steka S u proceduri DFS (ako je graf acikličan).

75. Dati asimptotsku analizu složenosti DFS algoritma.

Kao u zadatku 62 uradićemo celokupnu analizu. Sama procedura DFS bez ulaska u DFS-VISIT se izvršava za svaki čvor jednom, stoga traje $\Theta(V)$.

Procedura DFS-VISIT se za svaki čvor v poziva tačno jednom. Unutar nje je petlja koja se izvršava $|Adj(v)|$ puta. Kako je

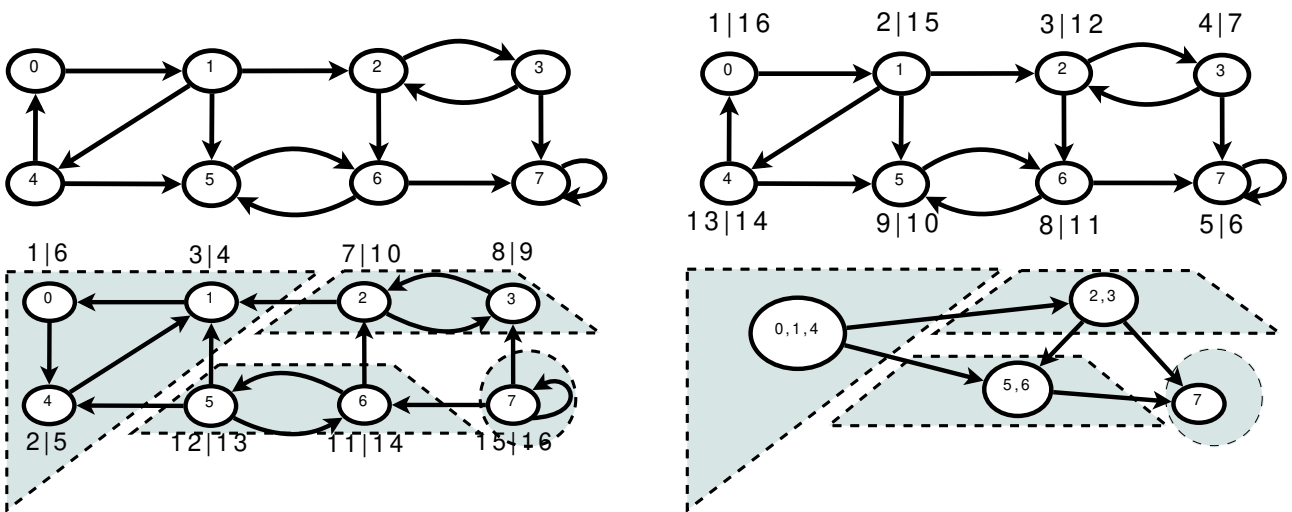
$$\sum_{v \in V} |Adj(v)| = \Theta(E),$$

sledi da je ukupno vreme izvršavanja DFS algoritma $\Theta(V + E)$.

76. Primeniti algoritam JAKO_POVEZANE_KOMPONENTE na graf sa slike.

```

function JAKO_POVEZANE_KOMPONENTE( $G$ )
  call DFS ( $\bar{G}$ ) da bi izračunao finish vremena  $f(u)$ 
  call TRANSPOSE_GRAPH ( $G, G^T$ )
  call DFS ( $G^T$ ) uzimajući čvorove  $u$  glavnoj petlji u opadajućem redosledu po  $f(u)$ 
  komponente povezanosti postaju čvorovi  $G_1$ 
  grane  $G$  postaju grane  $G_1$  ako povezuju komponente povezanosti
  return  $G_1$ 
end function
  
```



77. Dokazati da algoritam iz zadatka 76 daje graf jako povezanih komponenti.

78. Dokazati da je graf jako povezanih komponenti usmereni acikličan graf (DAG).

79. Implementirati u programskom jeziku C algoritam za nalaženje jako povezanih komponenti.

80. Na grafu $G = (V, E)$ je data težinska funkcije $w : E \rightarrow \mathbb{R}$. Dati Kruskalov algoritam za nalaženje minimalnog pokrivajućeg drveta (MST = minimum spanning tree, EN).

```

function KRUSKAL( $G, w$ )
   $A \leftarrow \emptyset$ 
  for each  $v \in V[G]$  do
    MAKE-SET( $v$ )           ▷ za svaki element skup koji ga sadrži
  end for
  sort( $E, w$ )              ▷ sortiraj grane iz  $E$  neopadajuće po  $w$ 
  for each  $(u, v) \in E[G]$  do ▷ redom, neopadajuće po  $w$ 
    if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ ) then
       $A \leftarrow A \cup \{(u, v)\}$  ▷ dodaj granu
      UNION( $u, v$ )           ▷ spoji skupove
    end if
  end for
  return  $A$ 
end function
  
```

Procedura MAKE-SET(v) pravi skup koji sadrži samo element v .

Procedura FIND-SET(v) nalazi skup u kojem je sadržan element v . Taj skup sadrži sve čvorove koji su do tad otkriveni i povezani su sa v dotad formiranim delom pokrivajućeg drveta.

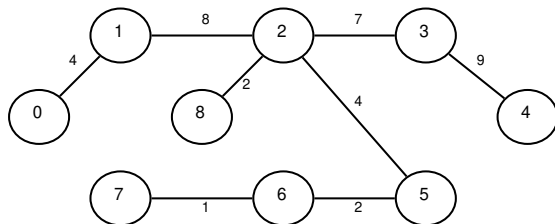
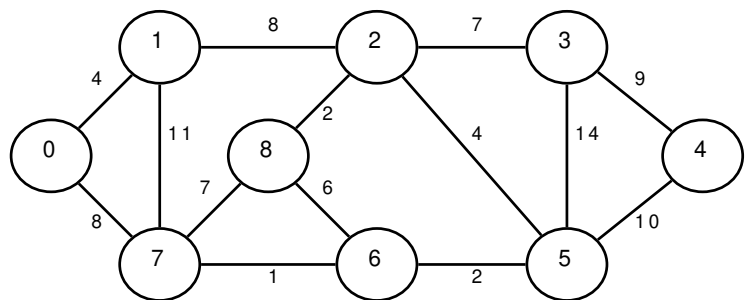
Procedura UNION(u, v) spaja skupove čvorova povezanih sa u i v , jer kad se grana (u, v) doda u pokrivajuće drvo, čvorovi iz tih skupova postaju povezani.

Kruskalov algoritam polazi od praznog skupa. U for-petlji dodaje grane sa težinama po neopadajućem redosledu na pokrivajuće drvo.

81. Implementirati Kruskalov algoritam iz zadatka 80 u programskom jeziku C.

82. Primeniti Kruskalov algoritam na graf sa slike desno.

Napisati redosled kojim su grane dodavane na pokrivajuće drvo i naći ukupnu težinu pokrivajućeg drveta.



	1	2	3	4	5	6	7	8	
u	6	5	2	2	0	2	1	3	
v	7	6	8	5	1	3	2	4	Σ
w	1	2	2	4	4	7	8	9	37

83. Na grafu $G = (V, E)$ je data težinska funkcije $w : E \rightarrow \mathbb{R}$. Dati Primov algoritam za nalaženje minimalnog pokrivajućeg drveta.

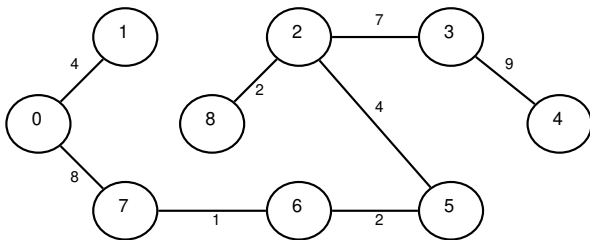
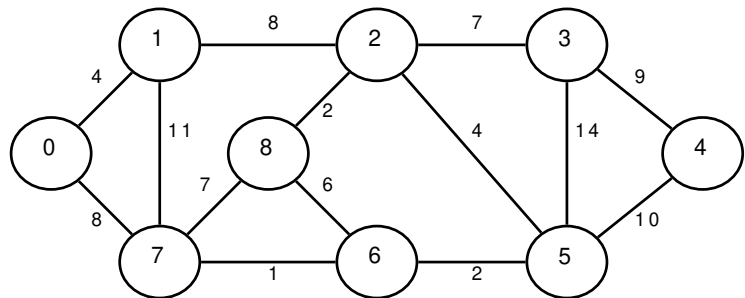
```

function PRIM( $G, w, r$ )
  for each  $u \in V[G]$  do
     $key[u] \leftarrow \infty$ 
     $\pi[u] \leftarrow \text{NULL}$ 
  end for
   $key[r] \leftarrow 0$ 
   $Q \leftarrow \text{PQ\_BUILD}(V[G], key)$            ▷ lista svih čvorova postaje priority queue
  while  $\neg \text{PQ\_ISEMPTY}(Q)$  do
     $u \leftarrow \text{PQ\_EXTRACT\_MIN}(Q, key)$    ▷ isto kao DEQUEUE sa najmanjim key
    for each  $v \in \text{Adj}(u)$  do
      if  $\text{PQ\_ISMEMBER}(v, Q) \& (w(u, v) < key[v])$  then
         $\pi[v] \leftarrow u$                    ▷ ako budemo odabrali  $v$ , prethodnik je  $u$ ,
         $key[v] \leftarrow w(u, v)$            ▷ onda će grana  $w(u, v)$  ući u MST
      end if
    end for
  end while
   $A \leftarrow \emptyset$ 
  for each  $u \in V[G] \setminus \{r\}$  do
     $A \leftarrow A \cup (\pi[u], u)$            ▷ u listi prethodnika implicitno imamo MST
  end for
  return  $A$ 
end function

```

84. Implementirati Primov algoritam iz zadatka 83 u programskom jeziku C.

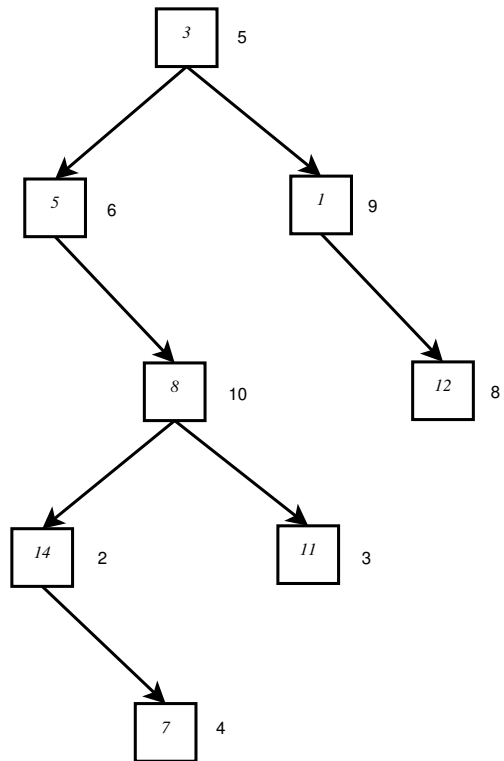
85. Primeniti Primov algoritam sa $r = 0$ na graf sa slike desno. Napisati redosled kojim su grane dodavane na pokrivajuće drvo i naći ukupnu težinu pokrivajućeg drveta.



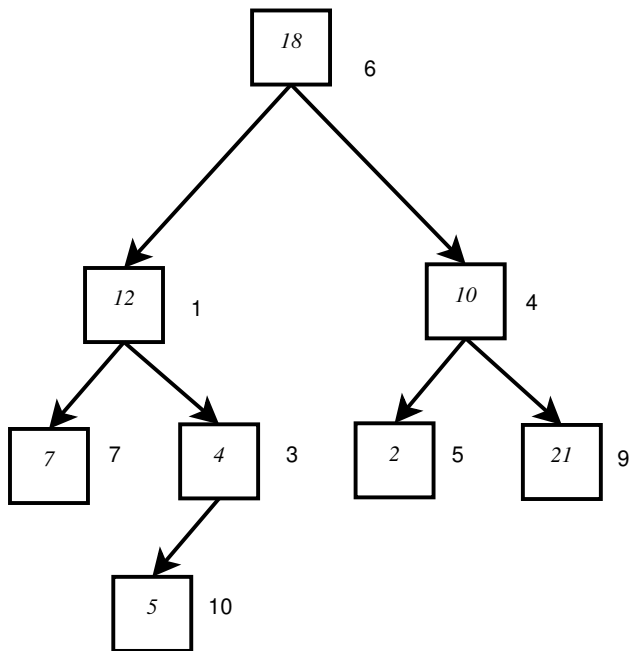
	1	2	3	4	5	6	7	8	
u	0	0	7	6	5	2	2	3	
v	1	7	6	5	2	8	3	4	Σ
w	4	8	1	2	4	2	7	9	37

86. Rekonstruisati binarno drvo dato u LC-RC reprezentaciji sa korenom na adresi 5.

I	K	LC	RC
1	9	7	-
2	14	-	4
3	11	-	-
4	7	-	-
<u>5</u>	3	6	9
6	5	-	10
7	4	6	1
8	12	-	-
9	1	-	8
10	8	2	3

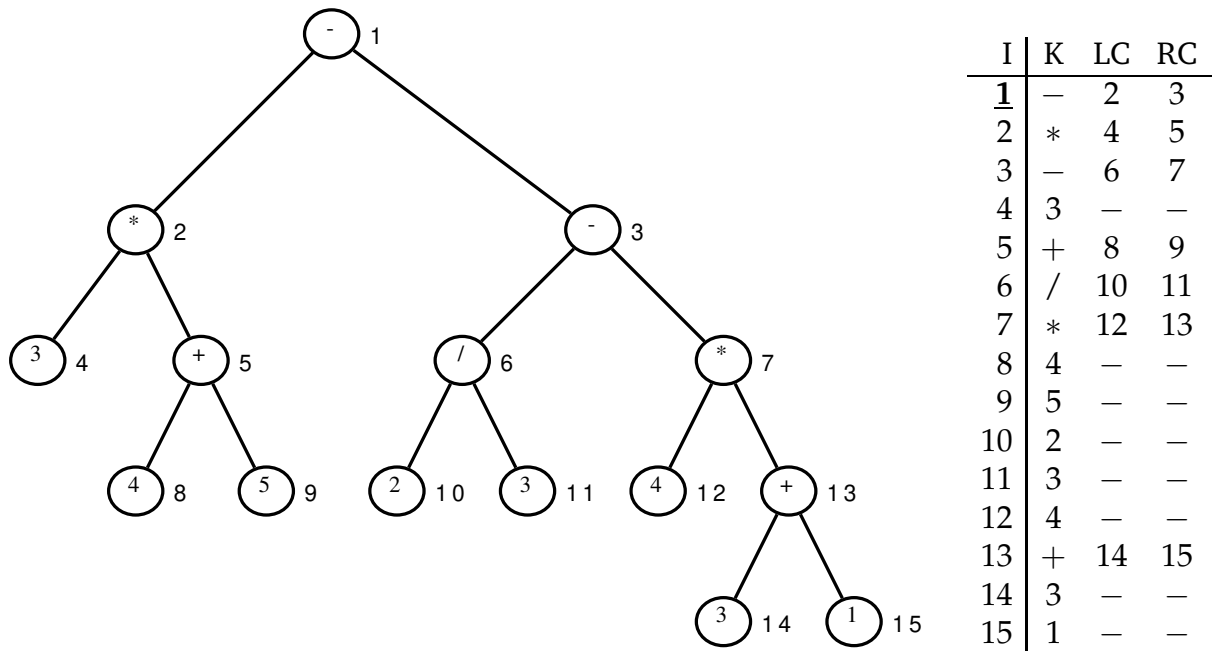


87. Dati tabelu LC-RC reprezentacije grafa sa slike.



I	K	LC	RC
1	12	7	3
2			
3	4	10	-
4	10	5	9
5	2	-	-
<u>6</u>	18	1	4
7	7	-	-
8			
9	21	-	-
10	5	-	-

88. Nacrtati drvo terma izraza $3 \cdot (4 + 5) - (2/3 - 4 \cdot (3 + 1))$ i dati LC-RC reprezentaciju.



89. Napisati rekurzivnu proceduru koja ispisuje elemente drveta iz zadatka 86 u infiksnom redosledu i rekurzivnu proceduru koja dodaje parent polje svim čvorovima drveta.

```

#include <stdio.h>
#include <stdlib.h>
int key[] = {0, 9,14,11, 7, 3, 5, 4,12, 1, 8};
int LC[] = {0, 7,-1,-1,-1, 6,-1, 6,-1,-1, 2};
int RC[] = {0,-1, 4,-1,-1, 9,10, 1,-1, 8, 3};
int parent[]={0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};

void infix_print(int root) {
    if(root>-1){
        infix_print(LC[root]);
        printf("%3d,",key[root]);
        infix_print(RC[root]);
    }
}

void find_parent(int root, int p) {
    if(root>-1){
        parent[root] = p;
        find_parent(LC[root],root);
    }
}

find_parent(RC[root],root);
}

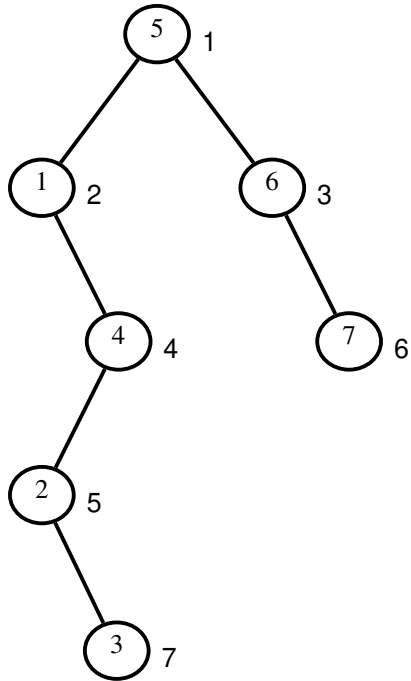
int main() {
    int i, root = 5;
    printf("Infix_print:\n");
    infix_print(root);
    printf("\n");
    printf("Parent:\n");
    for(i=1;i<11;i++)
        printf("%3d,",i);
    printf("\n");
    find_parent(5,-1);
    for(i=1;i<11;i++)
        printf("%3d,", parent[i]);
    printf("\n");
    return 0;
}

```

Infix print:
 5, 14, 7, 8, 11, 3, 1, 12,
 Parent:
 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
 -1, 10, 10, 2, -1, 5, -1, 9, 5, 6,

90. Za ulazni niz [5,1,6,4,2,7,3] kreirati binarno drvo tako da se element ako je manji od korena ubacuje u levo poddrvo, inače u desno poddrvo.

Čvorove ubacivati redom kojim su dati.



```

#include <stdio.h>
#include <stdlib.h>
#define max 10
int ulaz []={5,1,6,4,2,7,3};
int key[max];
int LC[max];
int RC[max];
int iskorisceno = -1;

int ubaci(int koren, int broj) {
    if(koren>-1){
        if(broj<key[koren])
            LC[koren]=ubaci(LC[koren], broj);
        else
            RC[koren]=ubaci(RC[koren], broj);
        return koren;
    }
    else{
        iskorisceno++;
        key[iskorisceno] = broj;
        LC[iskorisceno] = -1;
        RC[iskorisceno] = -1;
        return iskorisceno;
    }
}

int main() {
    int i, koren;
    koren = ubaci(-1,ulaz[0]);
    for(i=1;i<7;i++)
        koren = ubaci(koren, ulaz[i]);
    infix_print(koren);
    return 0;
}
  
```

91. Napisati proceduru ubaci koja ubacuje čvorove iz niza u drvo kao u zadatku 90. Reprerentacija drveta neka bude LC-RC. Neka ih glavni program potom ispiše u infiksnom redosledu dajući sortirani niz.

Rešenje je gore.

92. Trener plivačke reprezentacije ima za štafetu 4X100m na raspolaganju četiri plivača čija su vremena na 100m po stilovima: slobodno, leđno, prsno, baterflaj, data u tabeli.

	S	L	P	B
A	57	61	64	62
B	55	63	65	64
C	59	64	66	63
D	56	62	67	64

Kako da sastavi najbolju štafetu?

Ovo je problem angažovanja $n = 5$ radnika na $n = 5$ poslova, *Assignment Problem, EN*. To je specijalni slučaj transportnog problema koji je specijalni slučaj problema linearnog programiranja.

Kao problem linearnog programiranja, moguće je ovaj problem rešiti simplex metodom. Ipak, za rešavanje ćemo koristiti mađarsku metodu, *Hungarian Method, EN*.

1. korak: od elemenata matrice se oduzmu minimumi po vrstama:

$$1. \begin{bmatrix} 0 & 4 & 7 & 5 \\ 0 & 8 & 10 & 9 \\ 0 & 5 & 7 & 4 \\ 0 & 6 & 11 & 8 \end{bmatrix} \quad 2. \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 4 & 3 & 5 \\ 0 & 1 & 0 & 0 \\ 0 & 2 & 4 & 4 \end{bmatrix}$$

2. korak: od elemenata matrice se oduzmu minimumi po kolonama.

3. korak: nule se precrtaju minimalnim brojem k vertikalnih i horizontalnih linija:

$$3. \begin{bmatrix} \cancel{0} & \cancel{0} & \cancel{0} & \cancel{1} \\ 0 & 4 & 3 & 5 \\ \cancel{0} & \cancel{1} & \cancel{0} & \cancel{0} \\ 0 & 2 & 4 & 4 \end{bmatrix} \quad 4. \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 1 & 3 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 2 & 2 \end{bmatrix}$$

4. korak: zato što je $k = 3 < n = 4$, minimalni neprecrtan broj $m = 2$ se oduzima od neprecrtanih elemenata i dodaje dvaput precrtanim elementima i vraćamo se na 3.

3'. korak: nule se sad mogu precrtati sa minimalno $k = 4$ vertikalnih i horizontalnih linija, zato prelazimo na korak 5.

5. korak: Angažovanje se vrši izborom nule koja je sama u vrsti. Ako nema, bira se nula sama u koloni. Ako ni to nema, biramo proizvoljnu nulu.

Pri svakom izboru eliminišu se preostale nule u izabranoj vrsti/koloni.

$$5. \begin{bmatrix} 2 & \emptyset & \underline{0} & 1 \\ \underline{0} & 2 & 1 & 3 \\ 2 & 1 & \emptyset & \underline{0} \\ \emptyset & \underline{0} & 2 & 2 \end{bmatrix} \quad \text{Konačno:} \quad \begin{bmatrix} 57 & 61 & \underline{64} & 62 \\ \underline{55} & 63 & 65 & 64 \\ 59 & 64 & 66 & \underline{63} \\ 56 & \underline{62} & 67 & 64 \end{bmatrix}$$

Kad se izabrana angažovanja prenesu na polaznu tabelu vremena, dobija se optimalna štafeta čije "vreme" je $64 + 55 + 63 + 62 = 244$.

93. Softverska kompanija je zaposlila 5 pripravnika (A, B, C, D, E). Pripravnik će biti angažovani u 5 departmana kompanije (1, 2, 3, 4, 5). Da bi odredili koji pripravnik će se angažovati u kojem departmanu, uradili su test iz veština koje se koriste u odgovarajućem departmanu.

U tabeli su dati poeni osvojeni na testu.

	1	2	3	4	5
A	121	160	130	115	124
B	132	162	140	125	128
C	118	150	142	122	120
D	110	148	129	117	115
E	130	155	135	120	118

Naći optimalno angažovanje.

U ovom problemu optimalno angažovanje se dobija za maksimalni zbir bodova na testu. Maksimalni osvojeni broj bodova kod svih pripravnika na svim testovima je za B-2: 162. Stoga ćemo napraviti novu tabelu u koju ćemo uneti koliko na pojedinom testu fali do maksimalnih 162. Optimalno angažovanje je, onda, rešenje minimalnog angažovanja za novu tabelu.

Rešenje dobijeno mađarskom metodom je naznačeno u matrici:

$$\begin{bmatrix} 41 & \underline{2} & 32 & 47 & 38 \\ 30 & 0 & 22 & 37 & \underline{34} \\ 44 & 12 & \underline{20} & 40 & 42 \\ 52 & 14 & 33 & \underline{45} & 47 \\ \underline{32} & 7 & 27 & 42 & 44 \end{bmatrix}$$

Kada se to rešenje prenese na početnu tabelu dobija se maksimalno angažovanje: $160 + 128 + 142 + 117 + 130 = 677$.

94. Dati matematičku formulaciju problema angažovanja radnika $1, \dots, n$ na poslove $1, \dots, n$, ako su u matrici $C = [c_{i,j}]_{n \times n}$ data vremena $c_{i,j}$ koja su potrebna da radnik i obavi posao j .

Dodelićemo vrednost $x_{i,j} = 1$ ako se radnik i angažuje na posao j , $x_{i,j} = 0$ inače.

$$\zeta = \sum_{i=1}^n \sum_{j=1}^n c_{i,j} x_{i,j} \rightarrow \min$$

$$0 \leq x_{i,j} \leq 1, i, j \in \{1, 2, \dots, n\}, x_{i,j} \in \mathbb{Z},$$

$$\sum_{i=1}^n x_{i,j} = 1, j \in \{1, 2, \dots, n\},$$

$$\sum_{j=1}^n x_{i,j} = 1, i \in \{1, 2, \dots, n\}.$$

95. Neka je dat kompletan graf sa n čvorova i nad njegovim granama funkcija težina matricom $C = [c_{i,j}]_{n \times n}$. Težine $c_{i,j}$ predstavljaju dužinu ili cenu putovanja od mesta i do mesta j .
Dati matematičku formulaciju problema trgovačkog putnika¹⁹.

Problema trgovačkog putnika je nalaženje zatvorenog Hamiltonovog puta sa minimalnom ukupnom cenom, odnosno dužinom. Put (staza u kojoj se čvorovi ne ponavljaju) koji sadrži sve čvorove grafa je Hamiltonov put.

U kompletanom grafu nema petlji. Da bi se pojednostavio zapis problema, pretpostavljamo da je $c_{i,i} = \infty$, $i = 1, 2, \dots, n$. Na taj način smo sigurni da u optimalnom rešenju nema petlji. Planove putovanja ćemo definisati vrednostima $x_{i,j} = 1$ ako se putuje od mesta i do j , $x_{i,j} = 0$ inače. Tražimo:

$$\zeta = \sum_{i=1}^n \sum_{j=1}^n c_{i,j} x_{i,j} \rightarrow \min$$

$$0 \leq x_{i,j} \leq 1, i, j \in \{1, 2, \dots, n\}, x_{i,j} \in \mathbb{Z},$$

$$\sum_{i=1}^n x_{i,j} = 1, j \in \{1, 2, \dots, n\},$$

$$\sum_{j=1}^n x_{i,j} = 1, i \in \{1, 2, \dots, n\},$$

uz uslov nepostojanja zatvorenog podputa.

Ako se uslov nepostojanja podkonture izostavi, dobija se problem angažovanja, koji nazivamo **relaksirani problem** koji odgovara TSP.

Dopustivo neoptimalno rešenje se dobija metodom **najbližeg suseda**: polazi se od nekog čvora i ide u najbliži neposećen čvor sve dok se ne obiđu svi čvorovi, a tad se vraća u polazni čvor.

Optimalno rešenje se može naći Branch & Bound²⁰ metodom:

Reši se relaksirani problem. Ako se dobije rešenje bez podputova, to je optimum.

Ako postoje podputovi vrši se **grananje** mogućnosti da pojedina grana Hamiltonovog puta pripada ili ne pripada rešenju. U svakoj grani se rešava odgovarajući problem. U početku je donja granica optimum relaksiranog problema, posle se donja granica ažurira kako se pronade novo rešenje relaksiranog problema.

Gornja granica je u početku rešenje najbližeg suseda. Kad se u Branch & Bound drvetu pronade sledeće dopustivo rešenje ažurira se gornja granica. Tako ažurirana **granica** se koristi da prekine traženje rešenja u nekom poddrvetu kad se dobije da je relaksirano rešenje lošije od gornje granice.

¹⁹Travelling salesman problem (TSP), EN

²⁰Grananje i ograničavanje, EN

96. U tabeli su date udaljenosti između 5 gradova.

	1	2	3	4	5
1	-	28	115	45	110
2	28	-	87	30	93
3	100	87	-	75	115
4	45	30	75	-	135
5	120	93	110	135	-

- (a) Definirati problem trgovačkog putnika.
- (b) Polazeći od čvora 1, metodom najbližeg suseda naći približno rešenje problema trgovačkog putnika.
- (c) Za isti problem naći mađarskom metodom angažovanje koje je rešenje relaksiranog problema trgovačkog putnika.
- (d) Znajući rešenja (b) i (c), u kojim granicama se nalazi optimalno rešenje problema trgovačkog putnika?

- (a) Polazeći od jednog grada obići sve ostale gradove tačno jednom, vratiti se u početni grad tako da ukupan pređeni put bude minimalan.

Ili: naći zatvoreni Hamiltonov put minimalne dužine.

- (b) $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 5 \rightarrow 1$

Dužina puta $28 + 30 + 75 + 115 + 120 = 368$.

$1 \rightarrow 2$

$2 \rightarrow 4$

- (c) Mađarska metoda daje rešenje relaksiranog problema $4 \rightarrow 1$, čija dužina puta je

$3 \rightarrow 5$

$5 \rightarrow 3$

$28 + 30 + 115 + 45 + 110 = 328$.

- (d) Rešenje pod (c) nije optimalno jer ima podputove $1 \rightarrow 2 \rightarrow 4 \rightarrow 1$ i $3 \rightarrow 5 \rightarrow 3$. Zaključujemo da je optimalno rešenje između rešenja pod (b) i (c): $328 \leq \zeta^* \leq 368$.

$1 \rightarrow 2$

$2 \rightarrow 5$

Uzgred, optimalno rešenje je $5 \rightarrow 3$, dužina puta je $28 + 93 + 110 + 75 + 45 = 351$.

$3 \rightarrow 4$

$4 \rightarrow 1$

Spisak pitanja

1. Bubble sort
2. Insertion sort
3. Selection sort
4. Merge sort
5. Quick sort
6. Složenost algoritama i asimptotske oznake
7. Rekurzija
8. Pregled algoritama za sortiranje
9. Matrice
10. Povezane liste
11. Stack pomoću povezanih listi
12. Queue pomoću povezanih listi
13. Stack pomoću nizova
14. Queue pomoću nizova
15. Grafovi, osnovne definicije i teoreme
16. Strukture podataka za grafove
17. Breadth first search (BFS)
18. Algoritmi nad grafovima (stepen, diametar, path, ...)
19. Depth first search (DFS)
20. DFS forest, tipovi grana
21. DFS forest, topološko sortiranje
22. DFS forest, jako povezane komponente
23. Minimalno pokrivaјуće drvo, Kruskal
24. Minimalno pokrivaјуće drvo, Prim
25. Binarna drva, LC-RC reprezentacija
26. Binary search tree sort
27. Problem angažovanja, mađarska metoda
28. Problem trgovačkog putnika