Softverski alati u elektroenergetici
(ISPITNA PITANJA I ODGOVORI)

1. Hardver računara (delovi, tipovi, jedinice mere)
2. Softver računara (operativni sistem, aplikacija, multitasking, paralelizam)
3. Distribuirani sistemi (klijent-server model, arhitekture, virtuelizacija)
4. Internet servisi (IP adresa, URI, protokoli)
5. Algoritam (opis, osobine, modeli računanja, pseudokod)
6. Analiza algoritma
7. Asimptotske notacije vremena izvršavanja algoritma
8. Problem pretrage (opis, nabrojati algoritme i njihove osobine)
9. Binarna pretraga (algoritam, osobine, rekurzivni algoritam)
10. Problem sortiranja (opis, nabrojati algoritme i njihove osobine)
11. Selection sort algoritam
12. Insertion sort algoritam
13. Merge sort algoritam
14. Princip "podeli i osvoji" (opis, primena)
15. Quicksort algoritam
16. Heapsort algoritam
17. Heap struktura podataka (opis, metode, namena)
18. Red sa prioritetima (operacije, realizacija, primena)
19. Poređenje kao model računanja (stablo odlučivanja)
20. Algoritmi sortiranja složenosti □(n)
21. Redosledna statistika (opis, min, max, jednovremeni min i max, medijana)
22. Strukture podataka – stek i red (organizacija podataka, osobine, namena)
23. Strukture podataka – liste i stabla (organizacija podataka, osobine, namena)
24. Binarno stablo pretrage
25. AVL stablo
26. Rečnik podataka i heširanje (primene, priheš i heš, kolizije, amortizovano vreme izvršavanja)
27. Heširanje ulančavanjem
28. Otvoreno adresiranje
29. Grafovi (definicija, tipovi, primena, vrste algoritama)
30. Topološko sortiranje grafa
31. Obilazak grafa u širinu
32. Obilazak grafa u dubinu
33. Najkraći put u grafu (definicija, varijante, algoritmi)
34. Dijkstra algoritam
35. Bellman-Ford algoritam
36. Složenost računanja i klase problema (P, NP, EXP, R, problem odlučivanja, primeri)
37. NP problemi (definicija P i NP problema, klase NP problema, redukcija)
38. Sekvencijalni i paralelni algoritmi (opis, Amdalov zakon)
39. Paralelni algoritmi (primeri i poteškoće implementacije)
40. Kriptografija (primena, procesi, šifrovanje i bezbednosne pretnje)
41. Osnovni algoritmi kriptografije (podela, primitivni i napredni algoritmi)
42. RSA algoritam (principi, uključeni algoritmi)
43. Algoritmi sa rad sa nizovima i stringovima (definicije, namene osnovnih algoritama)

1 Hardver računara (delovi, tipovi, jedinice mere)*

-**Računarski hardver** predstavlja skup fizičkih elemenata koji sačinjavaju računarski sistem. Mogućnosti računara u najvećoj meri zavise od hardvera i njegovih kvaliteta.

DELOVI

Osnovni delovi računara se mogu podeliti na:

1. Unutrašnju periferiju računara:
   -matična ploča,procesor, RAM(Random Access Memory), grafička kartica, zvučna kartica, napojna jedinica;
2. Uređaji za skladištenje podataka:
   -hard disk, optički uređaji(CD, CD-ROM, CD-RW, FDD), Fleš memorija;
3. Grafički izlazni uređaji:
   -monitor, štampač;
4. Zvučni ulazno izlazni uređaji:
   -mikrofoni, zvučnici , slušalice;
5. Ulazni uređaji:
   -tastatura, miš, skener, web kamera.

TIPOVI

-Personalni računari, radne stanice
-Laptop, notebook, ultrabook, tablet, smartphone
-Server, grid, superračunari;

JEDINICE MERE

Jedinice za **količinu podataka**:

1024	KiB	**Kibibyte**	1000	kB	**kilobyte**
1024^2	MiB	**Mebibyte**	1000^2	MB	**megabyte**
1024^3	GiB	**Gibibyte**	1000^3	GB	**gigabyte**
1024^4	TiB	**Tebibyte**	1000^4	TB	**terabyte**
1024^5	PiB	**Pebibyte**	1000^5	PB	**petabyte**
1024^6	EiB	**Exbibyte**	1000^6	EB	**exabyte**
1024^7	ZiB	**Zebibyte**	1000^7	ZB	**zettabyte**
1024^8	YiB	**Yobibyte**	1000^8	YB	**yottabyte**

Za **prenos podataka** se koriste jedinice:

bps – bits per second
Kbps – Kilo bits per second
Mbps – Mega bits per second
Gbps – Giga bits per second

Za **brzinu procesora** se koriste jedinice: IPS (*Instructions Per Second*)

-**MIPS** = Mega IPS = istorijski uobičajena mera (10^6 IPS)
-**GIPS** = Giga IPS = 10^9 IPS

FLOPS (*Floating Points Per Secont*) –broj operacija sa pokretnim zarezom u sekundi

-GFLOPS = Giga FLOPS = 10^9 FLOPS

2* Softver računara(OS, aplikacija, multitasking, paralelizam)

- **Softver računara** su programi koji govore računaru kako treba da izvršava određene zadatke. **Softver** je način zapisa algoritama u obliku koji je razumljim računaru.
- Softver se deli na na: **sistemski program**, koji je deo OS-a i **aplikativni program**, a izvršava se u izolovanom računaru ili u mreži kao deo distribuiranog sistema

OPERATIVNI SISTEM – OS

- **OS** je softver potreban za izvršavanje (aplikativnih) programa i za koordinaciju akrivnosti računarskog sistema (osigurava okruženje u kojem ostali programi mogu obavljati koristan posao)
- *Operativni sistem obuhvata:*
  - procedure raspodele resursa računarskog sistema
  - kontrole ulazno izlaznih operacija
  - upravljanje memorijom
  - prevođenja programskih jezika
  ...
- Moduli OS-a:
  - *Kod izvršavanja programa OS obezbeđuje programski interfejs (API) između aplikacija i usluga OS-a.*

-Karnel – jezgro OS-a, realizuje osnovno upravljanje hardverom
-Upravljanje memorijom, CPU, periferijama(*device drivers*)
-Korisnički interfejs
-Rad u mreži
-Bezbednost

MULTITASKING

- **Multitasking** u OS-u omogućava jednovremeno izvršavanje nezavisnih programa-procesa . Procesi dele procesor/e (paralelni algoritmi).
- **Schaduler** (raspoređivač) je deo karnela koji preključuje procese na procesore (CPU-e)

PARALELIZAM - u radu podrazumeva odvijanje više procesa (programa u stanju izvršavanja) u računarskom sistemu. Ovu karakteristiku poseduju oni OS-ovi koji su u stanju da odjednom započnu rešavanje više zadataka i omoguće njihovo istovremeno odvijanje.

3 Distribuiarani sistemi (klijent-server model, arhitekture, virtuelizacija)*

- **Distribuirani sistem -** čine povezani računari koje korisnik doživljava kao skladan sistem. Hardver i računari povezani komunikacionom mrežom rade kao jedan skladan sistem zahvaljujući softveru.
- Tipovi distribuiranih sistema:
  1. **Klaster** – računari su slični (identični), smešteni na jednoj lokaciji
  2. **Grid** – heterogeni udaljeni računari (raštrakani na više lokacija)
  3. **Cloud** – poput klastera kojem se pristupa sa daljine
     -javni klaud
     -privatni klaud

Internet tehnologije omogućiju povezivanje računara koji rade na istom OS-u ili na sličnim OS-ovima

KLIJENT-SERVER MODEL (K-S model)

- K-S model je struktura distribuiranih aplikacija gde se zadaci dele na pružaoce usluga ili resursa(servere) i korisnike usluga resursa (klijenti).
- Server i klijent obično komuniciraju kroz mrežu. Klijent inicira kontakt sa serverom i traži određenu uslugu, a server je ispunjava. Često se uloga računara poistovećuje sa programom koji traži ili pruža uslugu. npr. e-mail

VIRTUELIZACIJA

- **Virtuelizacija** - je stvaranje virtuelne (a ne stvarne) verzije nečega poput: OS-a, uređaja za skladištenje ili mrežnih resursa.
- **Virtuelizacija hardvera** – odnosi se na korišćenja VM, koja se ponaša kao stvaran računar sa OS-om. Softver VM je izolovan od hardvera koji hostuje VM.
  -Host machine – je stvaran hardver na kome se događa virtuelizacija.
  -Guest machine – je Virtuelna mašina
  -Hypervisor (VM menager) – je softver koji izvršava VM na hostu.
- Poznati hipervizori su: VMware, Virtuel Box, Hyper – V

4* Internet servisi (IP adresa, URI, protokoli)

- Internet je međunarodna mreža i naziva se globalna mreža računara ("Mreža nad mrežama")
- Internet servisi u širokoj upotrebi: email, www,chat, video konferencije, FTP.

IP ADRESA (Internet Protocol Address)

- Svaki računar povezan na internet ima svoju IP adresu.
  -IPv4 je u upotrebi (32-bitni broj)
  -IPv6 se uvodi

npr. 147.91.175.145.

- IP je jedinstven broj sličan telefonskom broju koji koriste računari u međusobnom saobraćaju putem interneta, uz korišćenje internet protokola. Ovo dozvoljava mašinama dalje sprovođenje informacije u ime pošiljaoca (kako bi mašina znala gde šalje istu) i kasnije primanje tih informacija (kako bi mašine znale gde dalje da ih pošalju.
- Konvertovanje u ove brojeve iz za ljude čitljivije forme adresa domena poput (www.esi.uns.ac.rs) se vrši preko DNS-a. Proces konverzije je poznat pod imenom **rastavljanje imena domena**.

URI (Uniform Resource Identifier)

- URI je string koji se koristi kao identifikator. URI se može koristiti kao lokacija (URL – Uniform Resource Locator), ime (URN – Uniform Resource Name) ili kombinovano URI = URL + URN.

INTERNET DOMENI (linked to IP)

Domen (Domain) je jedinstveno tekstualno ime web lokacije.

DNS (Domain Name Space) – je distributivna baza podataka i koristi se za lociranje servera.
-Mapira ime domena IP servera (mašine), gde se nalazi (hostuje) web lokacija.

Na najvišem nivou domeni se dele prema:
1. vrsti organizacija ili firmi koje su vlasnici domena.
2. prema državama (rs,fr,du,ba)

DNS serveri izvršavaju servis mapiranja. Oni su hijerarhijski uređeni i sarađuju u izvršavanju distribuiranog algoritma mapiranja.
-Kada se registruje novo ime domena zajedno sa TCP/IP adresom, brojni DNS serveri širom sveta će izvršiti ažuriranje svoje baze podataka.

PROTOKOLI

- **Protokoli** su pravila za formatiranje, predaju i prijem podataka.
- **TCP i IP** su osnovni internet protokoli. Najčešće korišćeni internet protokoli zasnovani na njima su:

1. **HTTP** (*Hyper Text Transfer Protocol*) – vezuje web servis i čitače
2. **HTTPS** (*Source HTTP*) – brine o bezbednoj komunikaciji web servera i čitača (prenos novca)
3. **SMTP** (*Simple Mail Transfer Protocol*) – koristi se za slanje email-a
4. **POP3** (*Post Office Protocol*) – koristi se za prijem email-a
5. **IMAP** (*Internet Messages Access Protocol*) – koristi se za čuvanje i održavanje email-ova
6. **FTP** (*File Transfer Protocol*) - brine o prenosu podataka

5*Algoritam (opis, osobine, modeli računanja, pseudokod)

- **Algoritam** predstavlja skup koraka za obavljanje zadatka.
- Od algoritma očekujemo:
  1. konkretno, tačno rešenje (navigacija nam daje najkraći put, ali negde je moguće dati samo optimalno rešenje)
  2. algoritam treba da bude BRZ!

OSOBINE

-modularnost (izmenjivost)
-funkcionalnost
-robusnost
-lakoća razumevanja
-proširivost
-pouzdanost,
-brzina
-zauzeće memorije

MODELI RAČUNANJA

- Su određeni sa:
  - operacijama koje su dozvoljene
  - "cenu" (trajanje) svake operacije
  - ukupna "cena" algoritma = suma "cena" svih operacija
- Opšti modeli računanja:
  1.**RAM** (Random Access Machine)
  2.**Pointer Machine**
  -Programski jezici obično omogućavaju oba modela računanja

RAM – je apstraktna mašina u opštoj klasi registarskih mašina. Koncept mašine sa slucajnim pristupom počinje sa najjednostavnijim modelom, takozvanim modelom brojačke mašine. Međutim, dve osobine ga razlikuju od brojačke mašine. Prva je indirektno adresiranje, a druga čini da model teži ka više konvencionalnim kompjuterima baziranim na akumulatoru, koji imaju jedan ili više pomoćnih registara, od kojih se najpoznatiji naziva "akumulator". Modeluje se kao dugačak niz reči.

npr. O(1) su operacije:

-pristup reči u RAMu (na bilo kojoj adresi) ili registru (promenljivoj)
-čitanja, pisanja vrednosti, računanja (+,-,*,/,..)

Pointer Machine – Objekti se dinamički formiraju (alociraju). Pristup poljima objekta je $\Theta(1)$. Osobina je da je polje sadrži reč ili pokazivač na objekat (NULL – ništa).

PSEUDOKOD – je apstraktan, formalan jezik za opis algoritm. Tipično struktuiran engleski jezik.
Razumeju ga ljudi - matematičari i ne mora se prevoditi u računaru.

6* Analiza algoritma

- Brzina izvršavanja algoritma zavisi od sledećih stavki:
  -računara na kojem se algoritam izvršava
  -programskog jezika
  -programskog prevodioca
  -ULAZA (n)
- Za analizu vremena izvršavanja algoritma važe sledeće **"*cene*" ili "model machine":**
  1. (1) za operacije koje se događaju unutar algoritma (+,-,*,/, <,>,...)
  2. (1) za operator dodele
  3. (1) za ulazak u funkciju
  4. (1) za return

- primer1 Sum (a, b){
  return a + b; Izvršava se u konstantnom vremenu O(1)
  } (1) + (1) = Tsum = 2

- primer2 SumOfList(A, n){ "cena" broj izvršavanja
  1. total = 0; (c1) 1 (dodela) 1
  2. for i = 0 to n – 1 (c2) 2 (komp. i dodela) n+1(false condition, napušta loop)
  3. total += A (c3) 2 (sabiranje, dodela) n
  4. return total (c4) 1 (return) 1
  }

  TSumOfList = 1 + 2(n+1) + 2n + 1 Tsum = k T(n)
  = 4n + 4 TSumOfList = cn + c' T(n)
  T(n) = 4n + 4 TSumOfMatrix = $an^2 + bn + cn$ $T(n^2)$

- Cene operacija se množe sa brojem izvršavanja i kod polinomskog ishoda se uvek uzima za stepen složenosti algoritma. Treba ispitati sve slučajeve(najgori, prosečan i najbolji).

7*Asimptotske notacije vremena izvršavanja algoritma

- Približno procenjujemo vreme izvršavanja algoritma u zavisnosti od veličine ulaza *n* i to kada je *n* veliko.
  •**Θ-notacija**–opisuje vreme izvršavanja algoritma. Odnosi se na obe granice trajanja algoritma (najkraće i najduže trajanje)

$$\Theta(g(n)) = \{f(n) : \forall n \geq n_0, \exists c_1 > 0, \exists c_2 > 0 \Rightarrow 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

  •***O*-notacija** –koristi se da opiše najgore slučajeve(sa gornje strane)

$$O(g(n)) = \{f(n) : \forall n \geq n_0, \exists c > 0, \Rightarrow 0 \leq f(n) \leq c \cdot g(n)\}$$

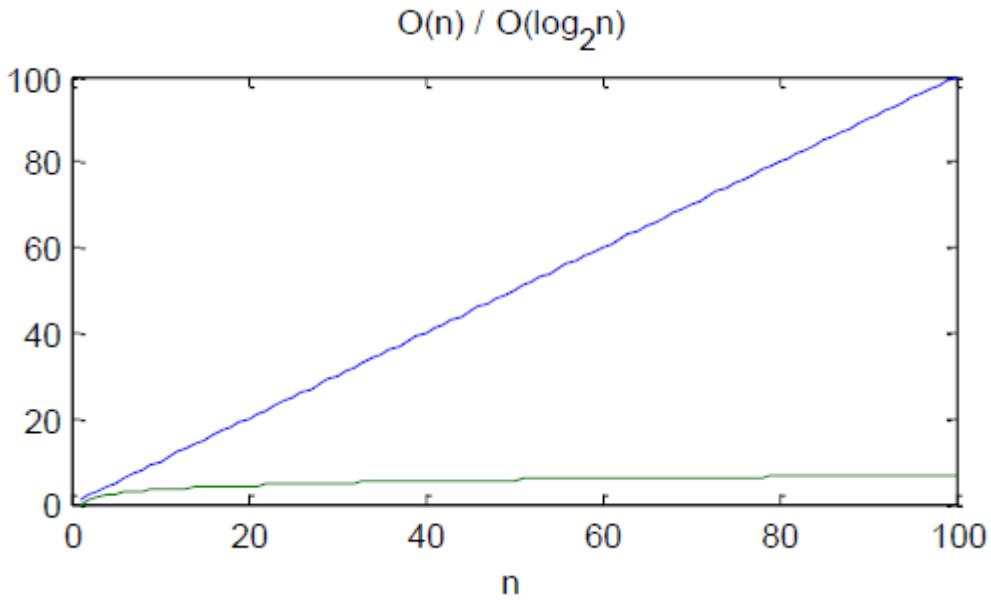
  •**Ω-notacija** –koristi se da opiše najbolje slučajeve(sa donje strane)

$$\Omega(g(n)) = \{f(n) : \forall n \geq n_0, \exists c > 0, \Rightarrow 0 \leq c \cdot g(n) \leq f(n)\}$$

8* Problem pretrage (opis, nabrojati algoritme i njihove osobine)

- Problem pretrage niza elemenata ili strukture se odnosi na traženje ključa u određenom nizu. U zavisnosti od modifikacije algoritama, ključ može biti **broj indeksa prvog elementa** sa zadatom vrednošću odnosno **brojevi indeksa svih elemenata** koji imaju jednaku zadatu vrednost.

- Problem pretrage biva unapređen tako što se niz prvo sortira, pa se od linearne složenosti O(n), dođe do rešenja logaritamske složenosti, što je daleko kvalitetnije $O(\log_2 n)$.



n	$\log_2(n)$
1.024	10
1.048.576	20
1.073.741.824	30

ALGORITMI PRETRAGE

1. ***LINEARNA PRETRAGA PRONAĐI PRVOG*** -vraća **indeks prvog** elementa koji ima zadatu vrednost ako postoji u nizu.
2. ***LINEARNA PRETRAGA PRONAĐI SVE*** – vraća **indekse svih** elemenata u nizu koji imaju zadatu vrednost, ukoliko ih ima u nizu.

Analiza: Linearna pretraga – pronađi prvog

```
LINEAR-SEARCH(A, key)          TRAJANJE   BROJ PROLAZA
1 for k=1 to A.length            c1          1..n+1
2   if A[k] == key               c2          1..n
3     return k                   c3          1..n
4 return nije-nađen              c4          0..1
```

- Najgori slučaj: $T(n) = c_1(n+1) + c_2 n + c_3 n + c_4$
  $T(n) = (c_1 + c_2 + c_3)n + c_1 + c_4 = O(n)$
- Najbolji slučaj: $T(n) = c_1 + c_2 + c_3 = \Omega(1)$

Zaključak: vreme izvršavanja algoritma je $O(n)$.

Analiza: Linearna pretraga – pronađi sve

```
LINEAR-SEARCH(A, key)          TRAJANJE   BROJ PROLAZA
1 rez = []                       c1          1
2 j = 0                          c2          1
3 for k=1 to A.length            c3          n+1
4   if A[k] == key               c4          n
5     j = j + 1                  c5          0..n
6     rez[j] = k                 c6          0..n
7 return rez                     c7          1
```

- Najgori slučaj: $T(n) = c_1 + c_2 + c_3(n+1) + c_4 n + c_5 n + c_6 n + c_7$
  $T(n) = (c_3 + c_4 + c_5 + c_6)n + c_1 + c_2 + c_3 + c_7 = O(n)$
- Najbolji slučaj: $T(n) = c_1 + c_2 + c_3(n+1) + c_4 n + c_7$
  $T(n) = (c_3 + c_4)n + c_1 + c_2 + c_3 + c_7 = \Omega(n)$

Zaključak: vreme izvršavanja algoritma je $\Theta(n)$.

*9*Binarna pretraga (algoritam, osobine, rekurzivni algoritam)*

- Binarna pretraga zahteva sortiran niz kao ulaz. Brzina algoritma je O($\log_2 n$). Kako su podaci(ključevi) sortirani npr. u rastućem redosledu, prvo očitamo vrednost na sredini niza, a zazim:
  1. Ako je to tražena vrednost onda je pretraga gotova.
  2. Ako je vrednost manja od tražene, onda je tražena vrednost u desnoj polovini niza.
  3. Ako je ta vrednost manja od tražene, onda je tražena vrednost u levoj polovini niza

- Time se niz prepolovi, tako da se gleda onaj od interesa i postupak se sprovodi iterativno u podnizovima koji se smanjuju (polove) sve dok ne ostane samo jedan elemenat u podnizu.

Binarna pretraga - algoritam

```
BINARY-SEARCH(A, key)
1  p = 1                    // leva granica
2  r = A.length             // desna granica
3  while p <= r
4     q = ⌊(p + r)/2⌋       // sredina
5     if A[q] == key
6        return q           // bingo
7     elseif A[q] > key
8        r = q - 1
9     else
10       p = q + 1
11 return nije-nađen
```

Binarna pretraga – rekurzivni algoritam

```
BINARY-SEARCH(A, key)
1 RECURSIVE-BINARY-SEARCH(A, 1, A.length, key)

RECURSIVE-BINARY-SEARCH(A, p, r, key)
1  if p > r
2    return nije-nađen
3  else
4    q = ⌊(p + r)/2⌋
5    if A[q] == key
6      return q
7    elseif A[q] > key
8       RECURSIVE-BINARY-SEARCH(A, p, q-1, key)
9    else
10      RECURSIVE-BINARY-SEARCH(A, q+1, r, key)
```

10 Problem sortiranja (opis, nabrojati algoritme i njihove osobine)*

- Sortiranje elemenata se prvenstveno radi u korist lakše pretrage niza ili niza struktura.
- Ulaz u funkciju sortiranja jeste niz/niz struktura (sortira se prema određenom polju) i on se naziva **ključ** (sort key), a kao izlaz dobija se permutacija elemenata u redosledu koji smo hteli da postignemo (opadajući/rastući).
- Složenost algoritama sortiranja je $\Theta(n2)$ ili $\Theta(n \log 2n)$
- Algoritmi za sortiranje: Bubble sort, Selection Sort, Insertion sort, Shell sort, Merge sort, Quicksort, Heap sort, Counting sort, Radix sort, Bucket sort.

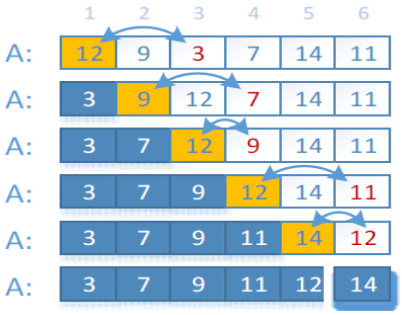
11*Selection sort algoritam

- U zavisnosti od toga da li tražimo rastući ili opadajući niz algoritam traži minimalni ili maksimalni element u nizu i stavlja ga na prvo mesto. Zatim trazi sledeći min/max element i stavlja ga na drugo mesto u nizu i tako dok se ne sortira kompletan niz.
- Algoritam radi u mestu, što znači da će i proći i kroz sortiran niz sa složenosti od $\Omega(n^2)$, jer ne prepoznaje sortiran niz.

Sortiranje izborom (*Selection sort*)

```
SELECTION-SORT(A)
1 for i = 1 to A.length-1
2   indMin = i
3   for j = i+1 to A.length
4     if A[j] < A[indMin]
5       indMin = j
6   A[i] ↔ A[indMin]        // zameni
```

Primer *Selection sort*



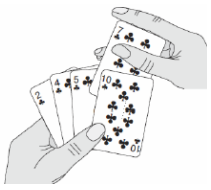
12*Insertion sort algoritam

- U algoritmu se svi elementi se ponašaju kao nesortirani. Kada započne sortiranje, funkcija smešta sortirane brojeve na levu stranu, a nesortirani ostaju na desnoj.
- Za početak sortiranog niza se uzima elemenat na prvom indeksu. Sledeći korak je umetnuti elemenat na drugom indeksu u sortiran niz, tako što ćemo ga porediti sa prvim elementom iz soriranog dela. Svaki sledeći element će biti umetnut u sortiran niz poređenjem sa svim elementima od kojih je manji/ veći +1 u tom delu niza.
- Složenost algoritma u najboljem slučaju će biti $\Omega(n^2)$, kada je niz već sortiran

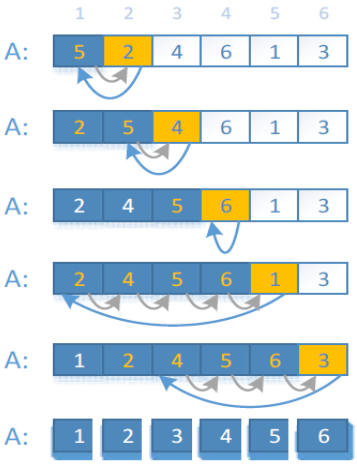
Sortiranje umetanjem (*Insertion sort*)

```
INSERTION-SORT(A)
1 for j = 2 to A.length
2   key = A[j]
3   i = j-1
4   while i>0 and A[i]>key
5     A[i+1] = A[i]
6     i = i-1
7   A[i+1] = key
```

- Ideja algoritma je slična postupku ređanja karata u ruci.



Primer *Insertion sort*



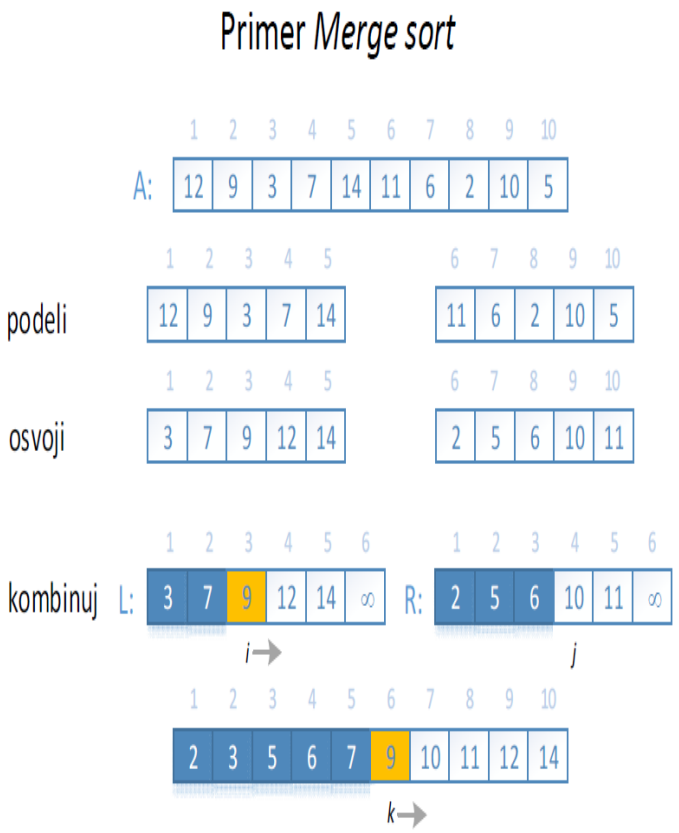
13* Merge sort algoritam

• Ovaj algoritam se razlikuje od *Selection* i *Insertion sort*-a:
– Njegovo vreme izvršavanja je Θ(*n* log2*n*), što je znatno brže kada se gledaju najgori slučajevi druga dva algoritma *O*(*n2*)
– Konstantan faktor u asimtotskoj notaciji je veći nego kod drugih algoritama – sporiji je za malo *n*
– Ne radi „u mestu“. Ne može da pomera elemente u nizu A, nego radi sa kopijama niza (što zahteva dodatnu memoriju).

• *Merge sort* primenjuje algoritamsku paradigmu „podeli i osvoji“:

1. **Podeli** niz na dva dela (u zavisnosti od implementacije forsira levu ili desnu strannu u slučaju neparnog broja elemenata): - prvi podniz A[p,....,q]
   - drugi podniz A[q+1,....,r], p <= q <= r
2. **Osvoji** – svaki od delova se nezavisno sortira upotrebom Merge sort. Svaka polovina se dalje deli dok se ne dobije podniz sa jednim elementom. (base case) Algoritam se vraća unazad rekurzivno poredeći prvo pojedinačne elemente, sve dok ne dođe do dva matična podniza.

3. **Kombinuj** – dva sortirana podniza se objedinjuju (merge) principom “**dva prsta**” u sortiran niz



Merge sort – Algoritam (1)

```
MERGE-SORT(A)
1 MERGE-SORT-STEP(A, 1, A.length)

MERGE-SORT-STEP(A, p, r)
1  if p < r
2     q = ⌊(p + r)/2⌋
3     MERGE-SORT-STEP(A, p, q)
4     MERGE-SORT-STEP(A, q+1, r)
5     MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
1  n1 = q - p + 1  // # elem. u levom podnizu
2  n2 = r - q      // # elem. u desnom podnizu
3  for i = 1 to n1           // kopiraj levi
4    L[i] = A[p + i - 1]
5  for j = 1 to n2           // kopiraj desni
6    R[j] = A[q + j]
7  L[n1 + 1] = ∞  // dodaj ∞ da bude > R[n2]
8  R[n2 + 1] = ∞  // dodaj ∞ da bude > L[n1]
9  i = 1          // indeks u levom
10 j = 1          // indeks u desnom podnizu
11 for k = p to r // “spoji” levi i desni
12   if L[i] ≤ R[j]
13     A[k] = L[i]
14     i = i + 1
15   else A[k] = R[j]
16     j = j + 1
```

14*Princip podeli i osvoji (opis, primena)

OPIS

- **Podeli i osvoji** (Devide and Conquer) je princip rada određenih algoritama u kojem se nesortiran niz deli na dva podniza. Svaki od tih podnizova se deli metodom odabranog algoritma do pojedinačnih elemenata (base cases), koji se dalje rekurzijom sortiraju poređenje i takvi se vraćaju u matični podniz (Merge sort) ili su elementi samim dovođenjem na *base cases*, elementi automatski sortirani pa se vraćaju u matični podniz rekurzijom trenutnog stanja položaja elemenata. (Quick sort). Na kraju se dva podniza kombinuju u jedan sortiran niz principom ***“dva prsta”****(u slučaju Merge sort-a)*

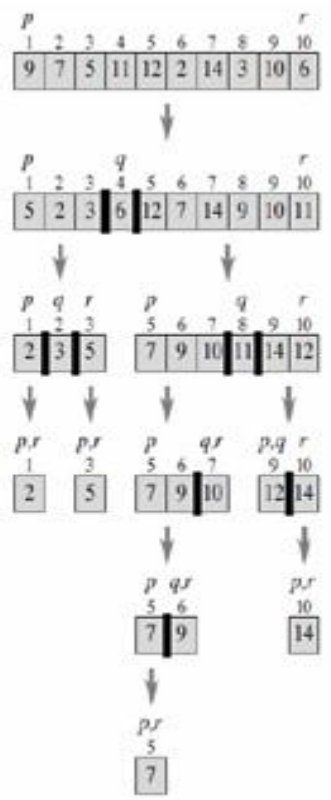
PRIMENA - *Quicksort i Merge sort*

15*Quicksort algoritam

• Quicksort (takođe) primenjuje algoritamsku paradigmu „podeli i osvoji“.
• Radi u mestu.
• Vreme izvršavanja je u najgorem slučaju kada je niz sortiran $O(n2)$, ali je u prosečnom slučaju Θ $(n \log 2n)$.
• Konstantan faktor u asimtotskoj notaciji je manji nego kod *Merge sort* algoritama, i za razliku od njega ne pravi kopije podnizova.

1. Podeli – poslednji elemenat iz niza se uzima za pivot i stavlja se na sredinu niza, tako da se sa leve strane nalaze elementi manji od pivota, a sa desne strane svi veći od pivota. Algoritam isti postupak radi i sa podnizovima dok ne dođe do *base cases.* Tada su elementi već sortirani, ali ih treba spojiti nazad u niz.
2. Osvoji - odvija se rekurzivno spajanje elemenata u jedan niz.

Quicksort – Algoritam (1)

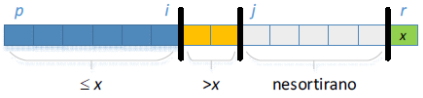


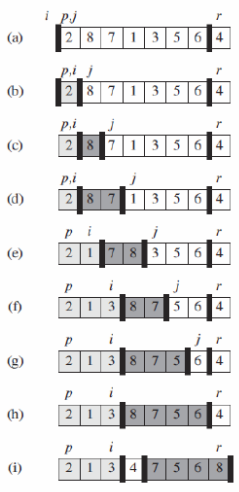
```
QUICKSORT(A)
1 QUICKSORT-STEP(A, 1, A.length)

QUICKSORT-STEP(A, p, r)
1 if p < r
2    q = PARTITION(A, p, r)
3    QUICKSORT-STEP(A, p, q-1)
4    QUICKSORT-STEP(A, q+1, r)
```

Quicksort – Algoritam (2)

```
PARTITION(A, p, r)
1 x = A[r]          // pivot
2 i = p - 1
3 for j = p to r-1
4   if A[j] ≤ x
5     i = i + 1
6     A[i] ↔ A[j]
3 A[i+1] ↔ A[r]
4 return i+1
```





Zaključak

- Algoritmi pretrage

Algoritam	Najgori slučaj	Najbolji slučaj	Zahteva sortiran niz?
Linearna pretraga	$\Theta(n)$	$\Theta(1)$	Ne
Binarna pretraga	$\Theta(\log_2 n)$	$\Theta(1)$	Da

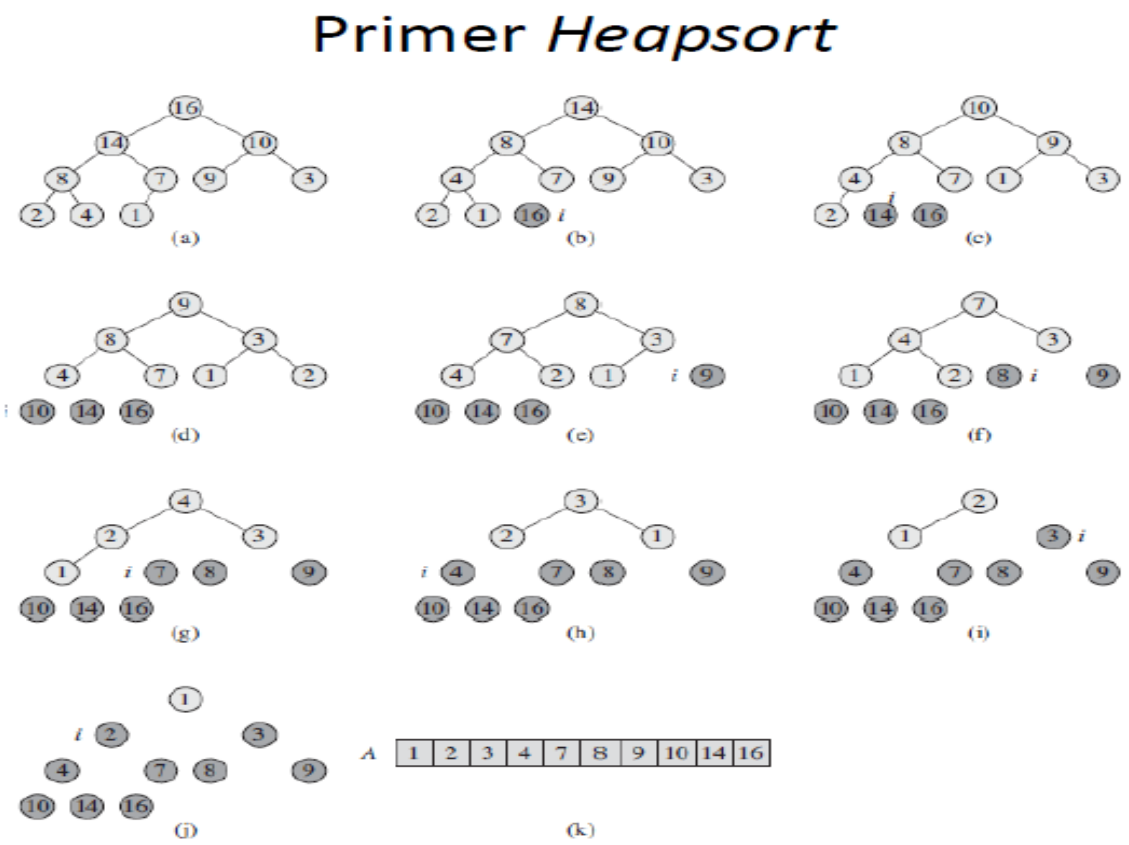
- Algoritmi sortiranja

Algoritam	Najgori slučaj	Najbolji slučaj	Broj zamena (najgori slučaj)	Radi u mestu?
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	Da
Insertion sort	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	Da
Merge sort	$\Theta(n\log_2 n)$	$\Theta(n\log_2 n)$	$\Theta(n\log_2 n)$	Ne
Quicksort	$\Theta(n^2)$	$\Theta(n\log_2 n)$	$\Theta(n^2)$	Da

16* Heap sort algoritam

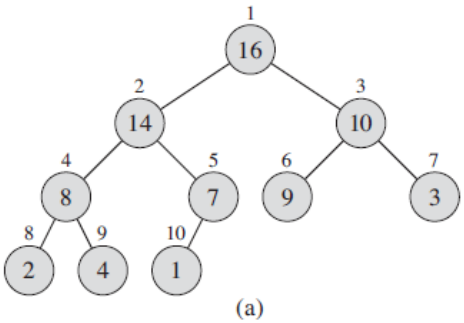
- Koristi se posebna struktura podataka *heap* (“hip”), po kojoj je algoritam dobio ime.
  - Heap je pogodan za implementaciju efikasnog niza sa prioritetima (Priority Queue)
- Slozenost algoritma iznosi $n\log_2 n$
  - Brzina algoritma je odlična, ali je od njega bolji dobro implementiran Quicksort
- Algoritam radi u mestu jer se promene mesta elemenata odvijaju u istom fizičkom nizu.

- Algoritam započinje Build-max-heap metodom. Nadalje najveći elemenat u nizu (koji je u korenu hipa) zamenjuje sa poslednjim elementom niza A, skraćuje niz za 1 i koriguje poredak (poziv Max-Hipify (A, 1)). Nastavlja se sa prethodnim korakom dokle god ima elemenat u nizu A.

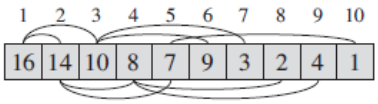


17* Heap struktura (opis, metode, namena)

- **Heap** je niz elemenata koji se može predstaviti kao binarno stablo.
  - Binarno stablo je skoro kompletno tj. moguće je da poslednji nivo nije popunjen do kraja
- Vrste heap-a:
  - Max-heap (A[Parent(i)] >= A[i]), najveći elemenat je u korenu
  -Min-heap (A[Parent(i)] <= A[i]), najmanji elemenat je u korenu
- Primer pravilno popunjenog Max-heap-a:



(a)



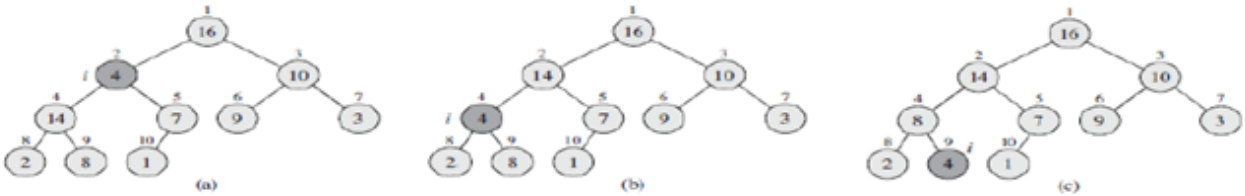
(b)

OSOBINE HEAP STRUKTURE

- Za Heapsort algoritam se koristi Max-heap
- Visina heap-a je broj nivoa, a ona iznosi $\log_2 n$
  Operacije su sa heap-om:
  -**MAX- HIPIFY** - održava Max-heap osobinu (složenost $O(\log_2 n)$)
  -**BUILD – MAX- HEAP –** pravi Max-heap na osnovu nesortiranog ulaznog niza O(n)
  -**HEAPSORT** sortiran niz u mestu (složenost $O(\log_2 n)$)
  - MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, HEAP-MAXIMUM služe za implementaciju priority queue-a (složenost $O(\log 2n)$).

MAX- HIPIFY METODA

- Primenjuje se za izgradnju Heap-a. Ako je u korenu podstabla vrednost manja negošto je u "deci", preoslediti tu vrednost na "dole", tako da se održi osobina Heap-a.
- Promene koje sprovodi MAX-HEAPIFY(A, 2)




```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4     largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7     largest = r
8  if largest ≠ i
9     A[i] ↔ A[largest]
10    MAX-HEAPIFY(A, largest)
```

- Zamena vrednosti u korenu sa nekim od dece je O(1) operacija, ali se vrednost iz korena može prebacivati u dobinu rekurzivnim pozivima Max-Hipify
- Za podstablo od n elemenata maksimalna veličina grane je 2n/3 (najgori slučaj je kada je poslednji nivo popunjen do pola – vidi prethodni primer: leva grana 6, a desna 3 elementa)
- Trajanje rekurzivnog poziva T(n) = O($\log_2$n), takođe ovo vreme se može iskazati preko <u>dubine stabla **h**</u> T(n) = O (n).

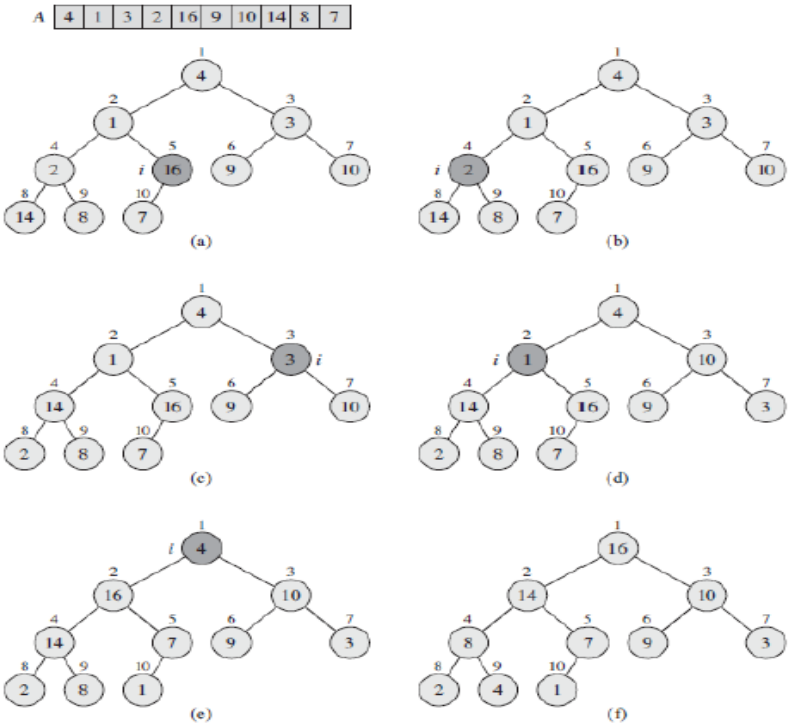
<u>BUILD – MAX – HEAP METODA</u>

Za izgradnju Max-Heap-a na osnovu niza A[1,2,...,n] koristi se Max Heapify metoda tako što se primeni(unazad) na svim elementima koji nisu lišće.

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Primer

BUILD-MAX-HEAP



18* Red sa prioritetima (operacije, realizacija, primena)

- Struktura podataka ***Priority Queue*** organizuje skup podataka S gde svaki elementima ima udružen prioritet kao "ključ".
- Max Priority Queue podržava operacije:
  - **Insert (S,x)-** dodaje elemenat x u skup S
  - **Maximum(S)-** vraća elemenat iz S sa najvećim ključem
  - **Increase-Key(S, x, k) –**povećava vrednost ključa elementa x na k (pod uslovom k>x)
  - **Extract-Max(S) –** Uklanja i vraća elemenat sa najvećim ključem iz S
- Primer upotrebe: raspoređivač zadataka u OS-u Max Priority Queue, čuva zadatke spremne za izvršavanje.
- Slično je sa Min Priority Queue*: Insert, Extract-Min, Extract-Max, Decrease-Key.*

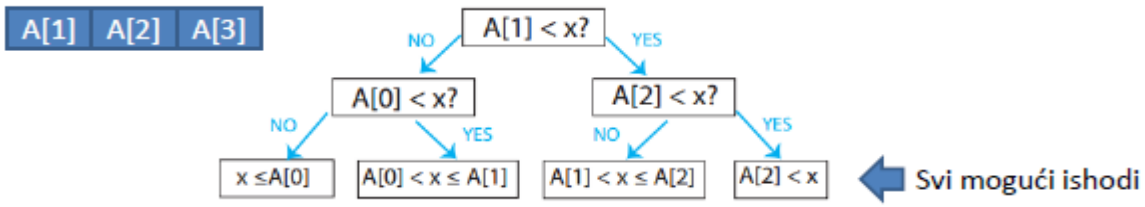
19*Poređenje kao model računanja (Stablo odlučivanja)

- Elementi su apstraktni tipovi podataka (ADT – *Abstract Data Types*).
- Postoji operacija poređenja elemenata (<,>....)
- Trajanje se iskazuje brojem poređenja

STABLO ODLUČIVANJA

- Stablo odlučivanja jeste klasifikacijski algoritam u formi stablaste strukture u kojoj se razlikuju dva tipa čvorova povezanih granama:
  1. **a krajnji čvor** ("leaf node") - kojim završava odredjena grana stabla. Krajnji čvorovi definišu klasu kojoj pripadaju primjeri koji zadovoljavaju uslove na toj grani stabla;
  2. **a čvor odluke** ("decision node") - ovaj čvor definiše uslov u obliku vrednosti određenog atributa (varijable), iz kojeg izlaze grane koje zadovoljavaju određene vrednosti tog atributa.
- Svaki algoritam gde se uporeduje, model poređenja može se prikazati kao stablo u svim mogućim ishodima poređenja (za dato n).

- Primer: Binarna pretraga (n=3)



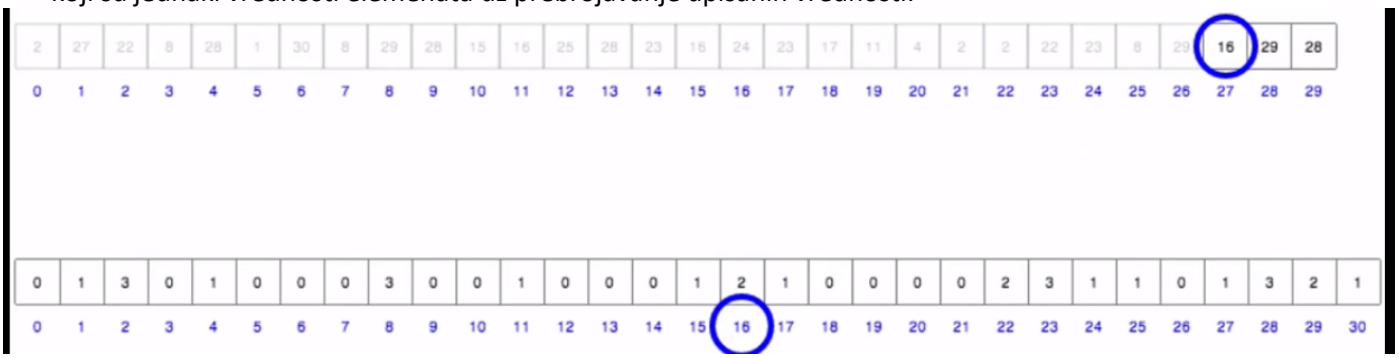
- Unutrašnji čvor = binarna odluka
- List = izlaz (algoritam je gotov)
- Putanja od korena do lista = izvršavanje algoritma
- Dužina putanje (dubina) = vreme izvršavanja
- Visina stabla = najgori slučaj izvršavanja alg.

- Niz se svaki put polovi u pretrazi, tako da je složenost pretrage $\log_2 n = h$ stabla, u najgorem slučaju.

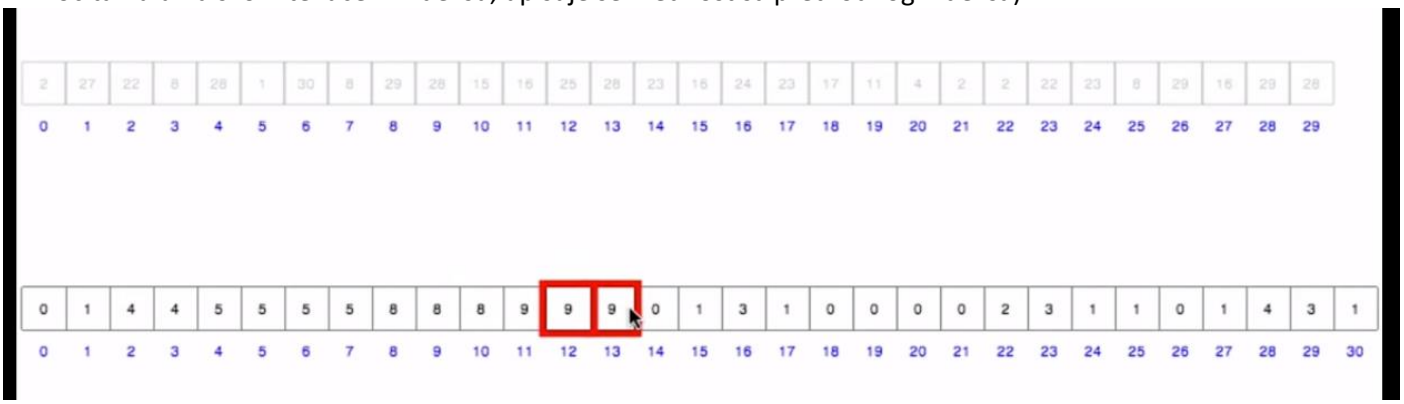
20*Algoritmi sortiranja složenosti O(n)

COUNTING SORT

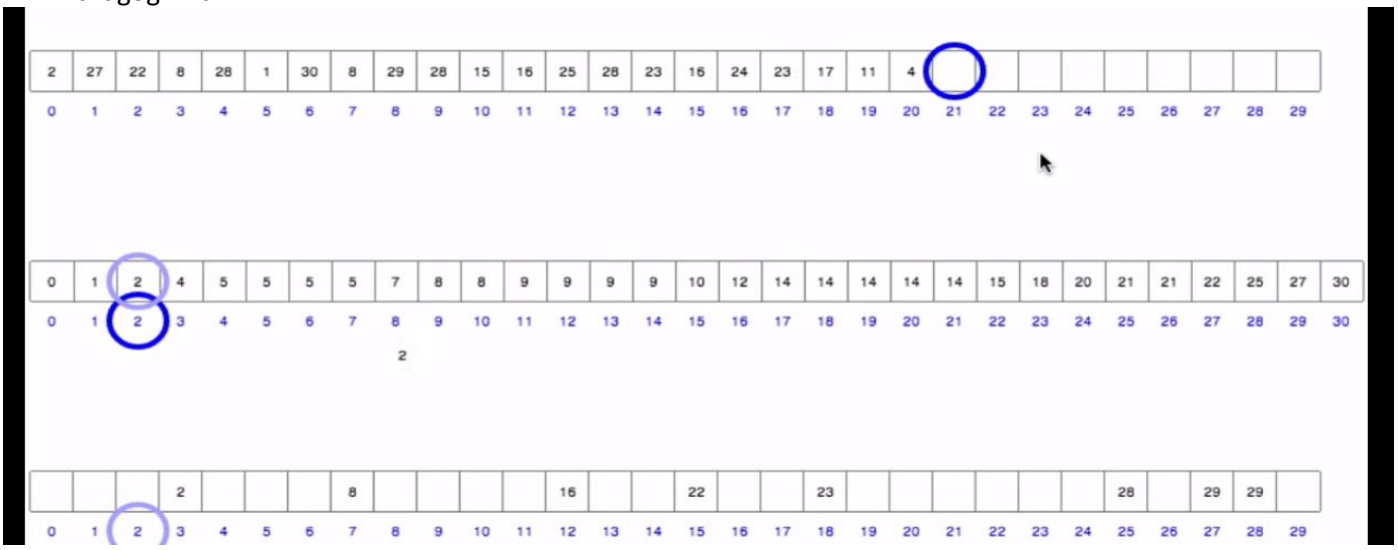
- **Radi samo sa pozitivnim celim brojevima i stabilan je(brojevi se pojavljuju u redosledu u kom su bili u matičnom nizu), ne radi u mestu.**
- Prvo se elementi nesortiranog niza redom smeštaju u novi podniz "sortiranih" elemenata na mesta indeksa koji su jednaki vrednosti elemenata uz prebrojavanje upisanih vrednosti.



- Sledeći korak je pomeraj dvostruke petlje kroz niz, gde druga petlja sabiranjem broja upisanih vrednosti na svom tekućem indeksu sa vrednošću koja je u na prethodnom indeksu prolazi kroz niz (ako druga petlja očita nulu na svom tekućem indeksu, upisuje se vrednost sa prethodnog indeksa).



- Sledeći korak je da se kreće sa poslednje pozicije matičnog niz. Vrednost koja se nalazi na tekućem indeksu matičnog niza će preuzeti vrednos indeksa u drugom nizu. Elemenat u drugom nizu koji se nalazi na tom mestu(indeksu), postaje vrednost indeksa u trećem nizu, gde biva upisan, broj indeksa tog tekućeg elementa iz drugog niza.



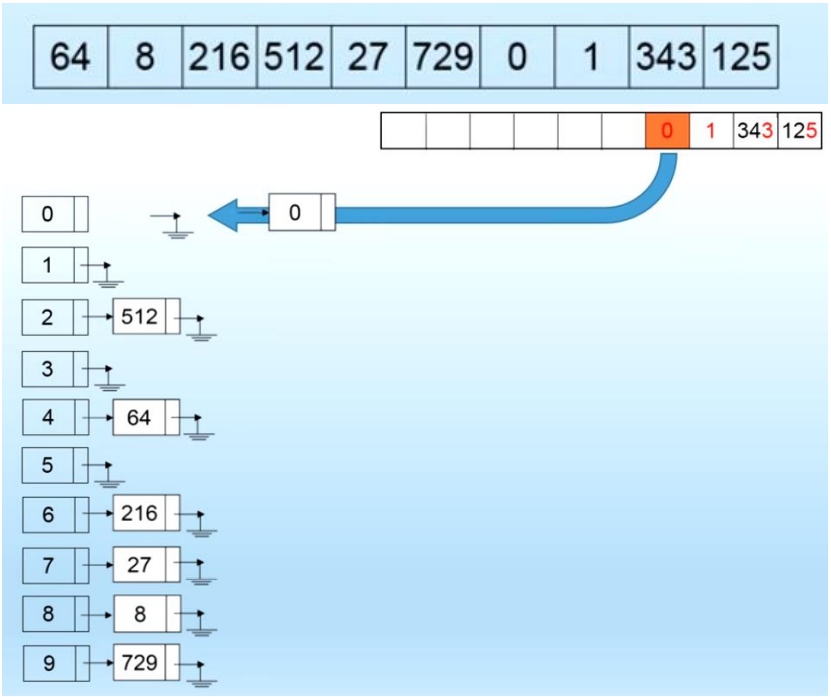
- Ukoliko ima više elemenata iz matičnog niza u drugom nizu upisanih na određeni indeks, kada se u ovom poslednjem koraku vrednost iz drugog niza iskoristi za sortiran niz (3. niz) vrednost upisana na tom indeksu u drugom nizu će se smanjiti za jedan, a element biva smešten na indeks koji je za jedan manji od vrednosti iz drugog niza.

Counting sort - algoritam

```
COUNTING-SORT(A, B, k)
1  for i = 0 to k
2    C[i] = 0
3  for j = 1 to A.length
4    C[A[j]] = C[A[j]] + 1
5  for i = 1 to k
6    C[i] = C[i] + C[i-1]
7  for j = A.length downto 1
8    B[C[A[j]]] = A[j]
9    C[A[j]] = C[A[j]] - 1
```

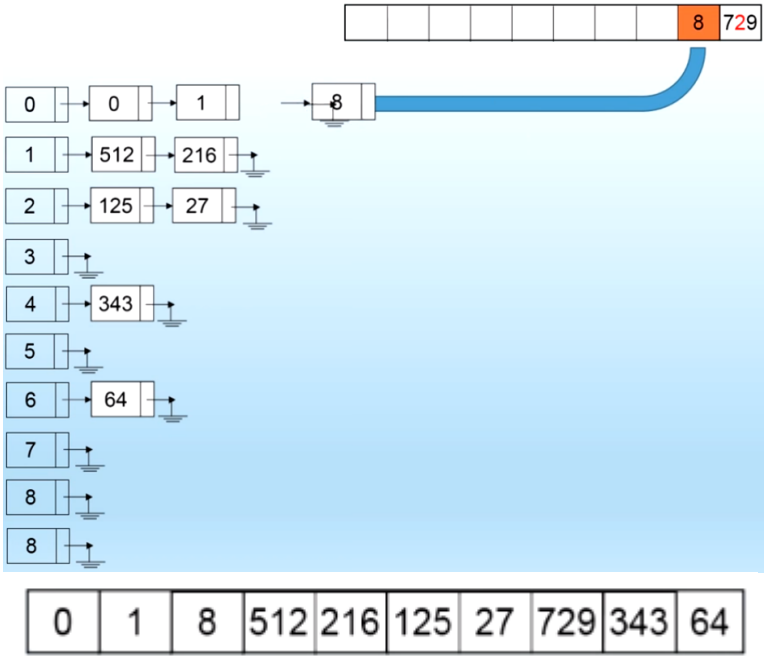
RADIX SORT

- U prvom prolazu Radix sort elemente nesortiranog niza postavlja u novi podniz, tako da ih smešta na broj indeksa koji su jednaki poslednjoj cifri broja koji se obrađuje. Zatim se izgradi novi niz od tih elemenata.



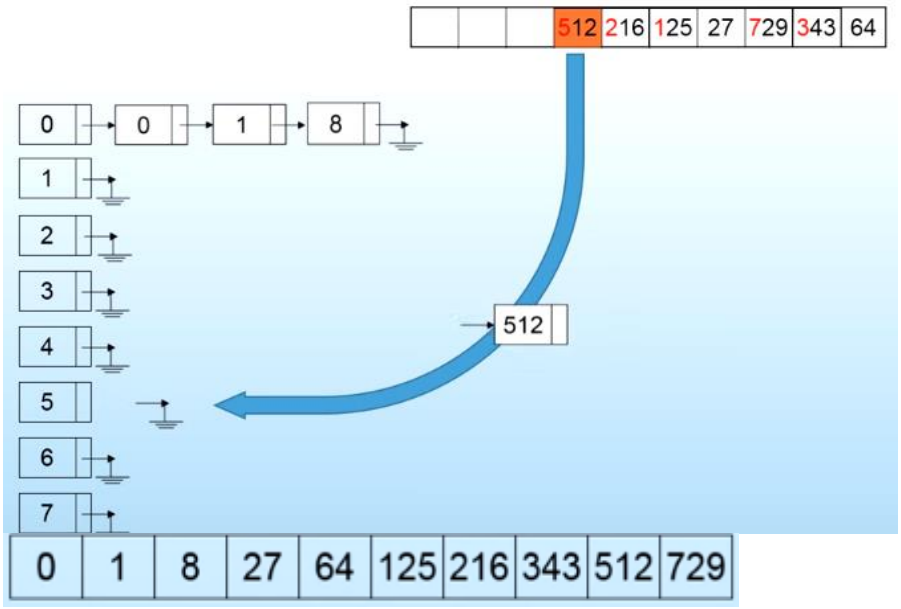
0	1	8	27	64	125	216	343	512	729

- U drugom prolazu kroz novi niz cifre se sortiraju prema deseticama. Ako postoje dva broja čije se desetice poklapaju, ulančavaju se jedan na drugi preko liste na tom indeksu. Sada se građenje novog niza sastoji od ubacivanja elemenata sa svakog indeksa istim redosledom kako su ubacivani u liste.



0	1	8	512	216	125	27	729	343	64

- U trećem prolazu se sortiraju prema stotinama, te se istim principom ulančavaju u liste na indeksima sa istom vrednošću stotina elemenata drugog podniza. SORTIRAN niz se događa kada se izgradi novi niz od ovako sortiranih lemenata izvučenih iz liste istim principom kao i u drugom nizu.



0	1	8	27	64	125	216	343	512	729

***Radix sort* ili *Quicksort*?**

• Složenost *Radix sort*-a Θ (*n*), a *Quicksort*-a je Θ(*n* log2*n*)

– Θ (*n*) je bolje od Θ (*n* log2*n*), ali ...

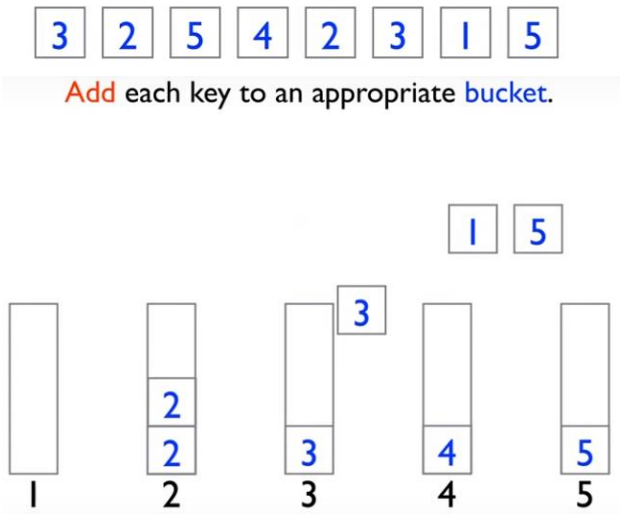
– Konstantan faktor *Radix sort*-a je lošiji od *Quicksort*-a

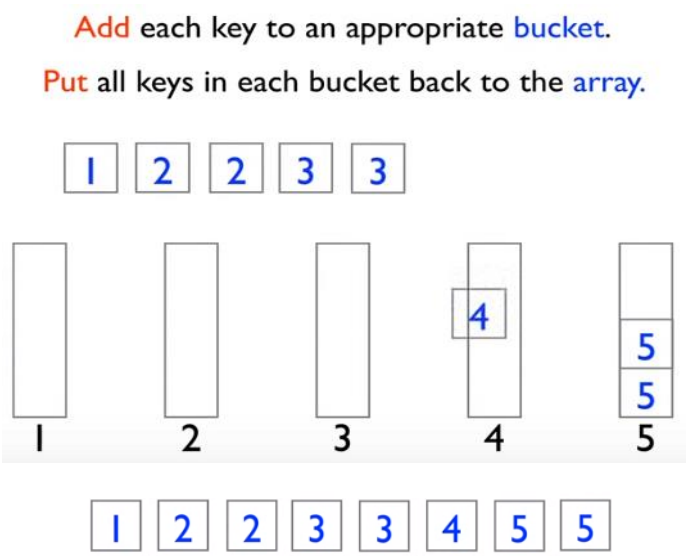
– Dobre implementacije *Quicksort*-a su brže od *Radix sort*-a

```
RADIX-SORT(A, d)
1 for i = 1 to d
2   Sortiraj cifru i stabilnim sort alg.
```

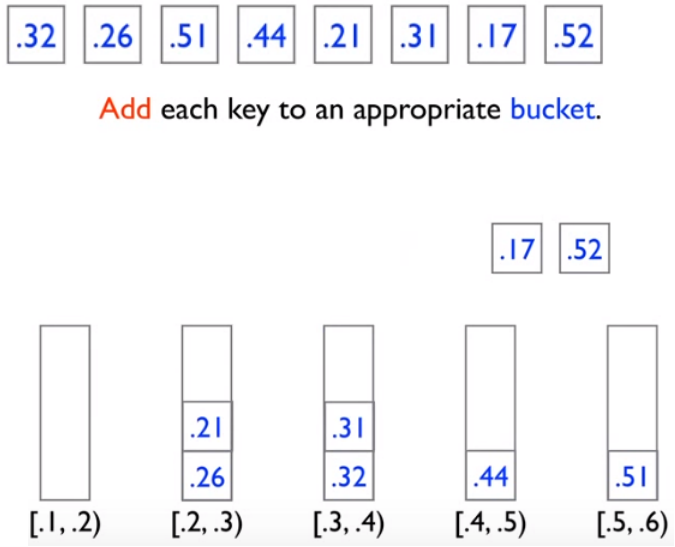
BUCKET SORT

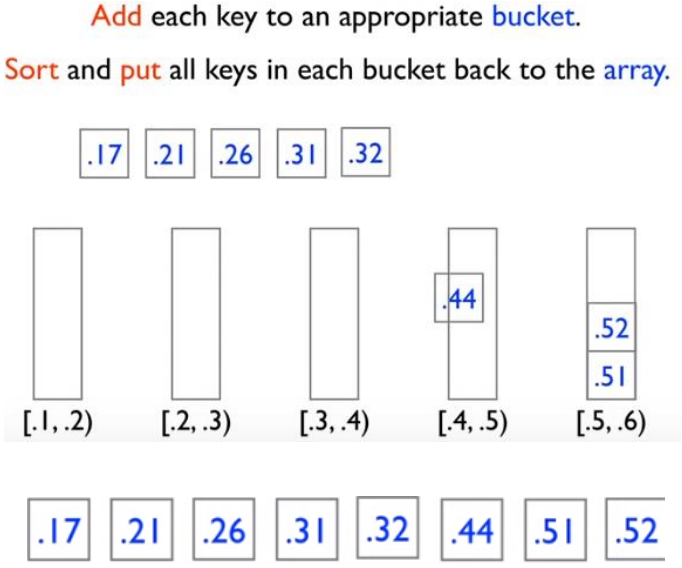
- U radu sa **celim brojevima** niz se sortira tako što se prave "posude"(liste) sa intervalima u koje se ubcuju elementi, zatim se sortiraju u rastućem redosledu u okviru intervala. Zatim se pravi sortiran niz tako što se redom "prazne" liste sa intervalima, redosledom kako su elementi ubacivani.





- U radu **sa realnim brojevima** prave "posude"(liste) sa intervalima u koje se ubcuju elementi za dati interval. Zatim svaki interval zasebno treba sortirati. Na kraju se sortiran niz formira tako što se sortirani intervali vraćaju u niz redom.






```
BUCKET-SORT(A, B, k)
1 n = A.length
2 for i = 0 to n-1
3   B[i] = ∅
4 for i = 1 to n
5   ubaciti A[i] u grupu B[⌊n·A[i]⌋]
6 for i = 0 to n-1
7   Sortirati grupu B[i]    ← Insertion sort
8 spojiti grupe B[0], B[1], … B[n-1]
```

21* Redosledna statistika (opis, min, max, jednovremeni min i max, medijana)

OPIS

- *Statistika i-tog redoslednog elementa iz skupa n elemenata je i-ti najmanji elemenat:*
  -Minimum = redosledno prvi elemenat i = 1
  -Maksimum = redosledno n-ti elemenat i = n
  - Medijana = redosledno na sredini minimuma i maksimuma
    *Za neparan broj elemenata je i = n/2
    *Za paran broj elemenata
    -Niža medijana i = $\lceil(\mathrm{n}\ +\ 1)/2\rceil$
    -Viša medijana i = $\lfloor(n\ +\ 1)/2\rfloor$

- **Problem: Odrediti *i*-ti redosledni elemenat na skupu od n različitih elemenata**
  -Ulaz: skup A sa n različitih elemenata, celobrojna ***i (i ≤ i ≤ n)***
  -Izlaz: elemenat koji je veći od i – 1 drugih elemenata
- Ovo je **problem selekcije** koji se može rešiti u dva koraka:
  1. Sortiranjem niza A, i
  2. Izborom i-tog elementa
- Prethodno rešenje ima složenost O(*n* log2*n*), i asimptotski je neefikasno
  -Traže se brža rešenja, složenosti O(n)

MINIMUM I MAKSIMUM

- Najmanji broj operacija poređenja neophodno za pronalaženje minimuma je n – 1 (najmanji broj se mora uporediti sa svakim drugim brojem) –nije efikasno

- Algoritmi:

```
MINIMUM(A)
1 m = A(1)
2 for i=2 to A.length
3   if A[i] < m
4     m = A[i]
5 return m
```

```
MAXIMUM(A)
1 m = A(1)
2 for i=2 to A.length
3   if A[i] > m
4     m = A[i]
5 return m
```

- **Nije potrebno 2(n-1) poređenja kako je gore naznačeno već 3 $\lfloor n\ /2\rfloor$ poređenja.**
- ALGORITAM:
  -Umesto poređenja (jednog) i-tog elementa sa tekućom najmanjom i najvećom vrednosti, treba posmatrati po 2 elementa (parove elemenata) iz niza A:

1. međusobno se porede elementi u paru 2. manji se poredi sa tekućim minimumom, 3. veći se poredi sa tekućim maksimumom

Ukupno 3 poređenja
za svaka dva elementa

MEDIJANA

- **Problem selekcije** i-tog elementa je teži od traženja minimuma ili maksimuma.
- Posmatramo dva algoritma:

  1. Randomized-Select – je moifikovana verzija Quicksort algoritma, jer se rekurzivno particionisanje odnosi samo na jednu stranu (polovinu) niza A gde se nalazi traženi elemenat

  ___ To čini da je očekivana složenost algoritma O(n), podrazumevajući da su svi elementi različiti.

  ___ Najgori slučaj ima složenost $O(n^2)$, ali samo kada se particionisanje dešava oko najvećeg(ili) najmanjeg elementa. Slučajnost izbora pivota (koja je posledica poziva **Random** funkcije) niti jedan poseban ulaz ne forsira najgori slučaj.

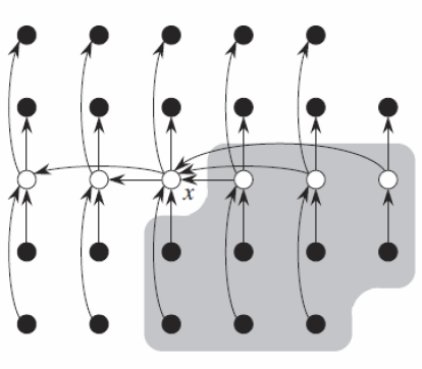
RANDOMIZED-SELECT algoritam

```
RANDOMIZED-SELECT(A,p,r,i)
1  if p==r
2    return A[p]
3  q = RANDOMIZED-PARTITION(A,p,r)
4  k = p-q+1
5  if i==k
6    return A[q]
7  elseif i < k
8    return RANDOMIZED-SELECT(A,p,q-1,i)
9  else return RANDOMIZED-SELECT(A,q+1,r,i-k)

RANDOMIZED-PARTITION(A,p,r)
1  i = RANDOM(p,r)
2  A[r] ↔ A[i]
3  return PARTITION(A,p,r)    // vidi Quicksort
```

2. Select – algoritam vrši selekciju u linearnom vremenu za najgori slučaj.

1. Podeli A na $\lceil n/5 \rceil$ grupa, gde svaka grupa ima 5 elemenata (poslednja može imati manje elemenata)
2. Nađi medijanu u svakoj od grupa upotrebom insertion sort-a
3. Upotrebi **Select** rekurzivno na novi niz sastavljen od medijana svih grupa iz prethodnog koraka nađi **x**
4. Primeni modifikovan **Partition** oko medijane-medijana (**x** iz koraka 3) koja je k-ti najmanji elemenat
   -Levu grupu čine elementi 1..k-1
   -Desnu grupu čine elementi k k+1...n
5. Ako je **i == k** kada je rešenje pronađeno, inače nastavi rekurzivno sa levom grupom ako je **i < k**, ili traži (i – k)-ti elemenat u desnoj grupi



Kružići su elementi
Beli kružići su medijane grupa
x je medijana-medijana
Strelice pokazuju na veći elemenat

22 Strukture podataka – stek i red (organizacija podataka, osobine, namena)*

• **Stek i red** su dinamički skupovi gde se elementi uklanjaju unapred određenim redosledom
– Kod steka Delete briše poslednji (najmlađi) dodat elemenat. Nazivamo ga LIFO procesiranje (*last-in, first-out*)
– Kod reda Delete briše prvi (najstariji) dodat elemenat. Nazivamo ga FIFO procesiranje (*first-in, first-out*)

STEK

• Operacije:
– **Push** – dodavanje elementa (insert)
– **Pop** – preuzimanje (brisanje) elementa (delete)
– Provera: da li ima elemenata?
• Podaci: niz S[1..S.*top*] sadrži elemente
– S[1] je elemenat na dnu, a S[S.*top*] je na vrhu steka.
– Ako je S.*top* == 0, onda je stek prazan
– Greška je kada se pozove Pop na praznom steku (greška tipa *underflow*)
– Greška je kada se pozove Push na steku sa *n* elemanata (*overflow*)

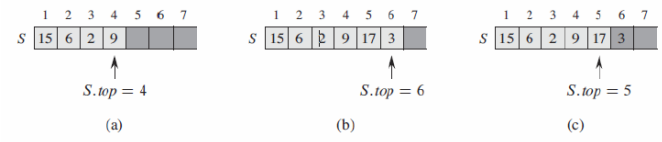
Stek primer

- Primer par operacija sa stekom...

  Push(S,3)

  Push(S,17)

  a = Pop(S)



Stek operacije

- Sve operacije su brze.
- Svaka traje $O(1)$

```
STACK-EMPTY(S)
1  if S.top == 0
2    return True
3  else return False

PUSH(S,x)
1  S.top = S.top + 1
2  S[S.top]= x

POP(S)
1  if STACK-EMPTY(S)
2    error "underflow"
3  else S.top = S.top - 1
4    return S[S.top + 1]
```

RED

• Operacije:

– **Enqueue** – dodavanje elementa (insert)

– **Deqeueu** – preuzimanje (i brisanje) elementa (delete)

– Provera: da li ima elemenata?

• Podaci:

– „glava“ (*head*) pokazuje na prvi elemenat reda.Preuzima se (dequeue) element sa početka – glave reda.

– „rep“ (tail) pokazuje na poslednji elemenat reda.Dodavanje elementa ga smešta na kraj – rep reda.

• Implementacija (jedan način):

– niz Q[1..*n*] je prostor za najviše *n*-1 elemenata reda

– Q se koristi kao kružni bafer

Operacije sa redom

```
ENQUEUE(Q,x)
1  Q[Q.tail] = x
2  if Q.tail==Q.length
3     Q.tail = 1
4  else Q.tail=Q.tail+1
```

```
DEQUEUE(Q,x)
1  x = Q[Q.head]
2  if Q.head==Q.length
3     Q.head = 1
4  else Q.head=Q.head+1
5  return x
```

- U gornjim operacijama nedoastaju provere
  - Greška je kada se pozove Dequeue na praznom redu (greška tipa *underflow*)
  - Greška je kada se pozove Enqueue na punom redu (*overflow*)

Primer reda

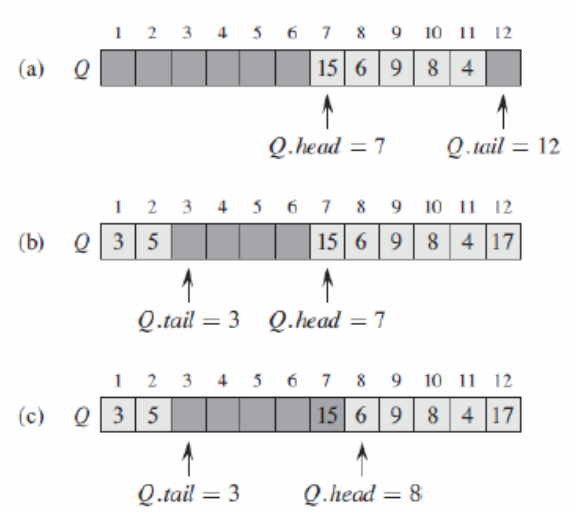
- Primer par operacija sa redom...

  Enqueue(Q,17)

  Enqueue(Q,3)

  Enqueue(Q,5)

  a = Dequeue(Q)



23* Strukture podataka – liste i stabla (organizacija podataka, osobine, namena)

POVEZANE LISTE

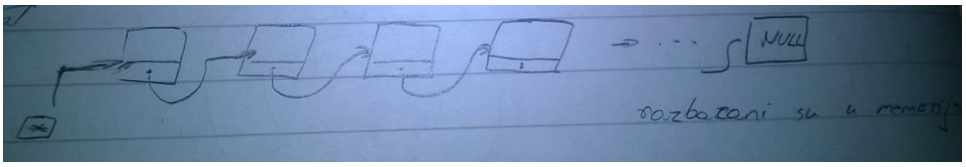
• Liste su strukture podataka gde su elementi uređeni u linearnom poretku.
• Za razliku od nizova gde je poredak uređen preko indeksa, kod lista imamo pokazivače na susedne elemente.
• Pogodna su za:
– Česta dodavanja i brisanja elemenata
– Česta posećivanja elemenata niza – iteriranje kroz listu.
• Nisu pogodne za pretrage

Tipovi lista

• Razlikuju se:
– **Jednostruko spregnute liste** gde element pokazuje na naredni element.
– **Dvostruko spregnute liste** gde element pokazuje i na naredni i na prethodni elemenat.
– Cirkularne liste – elementi su u „prstenu“ (prvi pokazuje na poslednji, a poslednji na prvi)
• Dodatno, lista može biti sortirana na osnovu ključa
– Prvi elemenat je sa najmanjim ključem, a poslednji sa najvećim

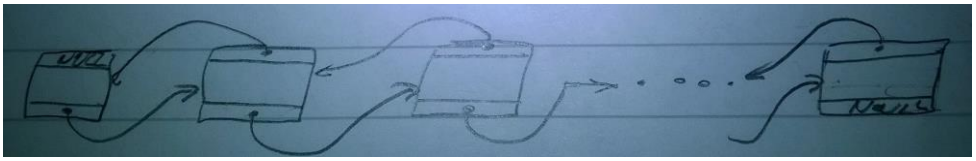
Jednostruko spregnuta lista

• Jednostavnija je od dvostruko spregnute liste
– Elementi nemaju *prev* polje koje pokazuje na prethodni elemenat
– Nema L.*tail*
• Omogućava kretanje od početka (L.*head*) ka narednim elementima.
• Nema mogućnost kretanja ka prethodnom elementu.
– Jedini način da se dođe do prethodnog elementa je ponovno kretanje od L.*head*.



Dvostruko spregnuta lista

• Svaki elemenat sadrži
– *next* – pokazivač na naredni elemenat (== Nil kada je poslednji)
– *prev* – pokazivač na prethodni elemenat (== Nil kada je prvi)
– *key* – ključ, pristan kod sortiranih lista
– drugi podaci (nisu prikazani na slici)
• Pokazivač na prvi elemenat – L.*head*
– Ako je L.*head* == NIL lista je prazna
• Pokazivač na poslednji elemenat – L.*tail*



Brisanje elementa iz liste

```
LIST-DELETE(L,x)
1  if x.prev ≠ Nil
2     x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ Nil
5     x.next.prev = x.prev
```

- (slika pre i posle)

Dodavanje elementa u listu

```
LIST-INSERT(L,x)
1  x.next = L.head
2  if L.head ≠ Nil
3     L.head.prev = x
4  L.head = x
5  x.prev = Nil
```

- (slika pre i posle)

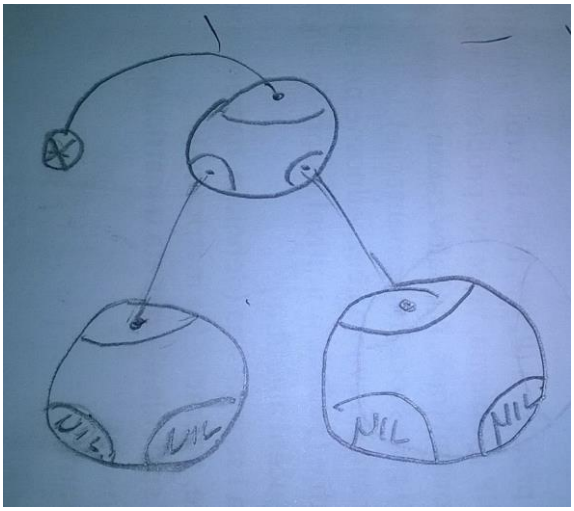
Pretraga u listi

```
LIST-SEARCH(L,k)
1  x = L.head
2  while x ≠ Nil and x.key ≠ k
3     x = x.next
4  return x
```

- Složenost $\Theta(n)$ u najgorem sllučaju jer je traženi elemenat na kraju liste.

24. Binarno stablo pretrage (BSP)

- **Binarno stablo pretrage** je sortirano binarno stablo. Svaki čvor u stablu ima uporedivi ključ čija je vrednost:
  -veća od ključeva u levom podstablu (>=)
  -manja od ključeva u desnom podstablu (<=)
- Osobine:
  (+) pretraga, ubacivanje i brisanje elemenata se može uraditi efikasno
  (-) stablo može izgledati degenerisano ili izgledati kao spregnuta listaž



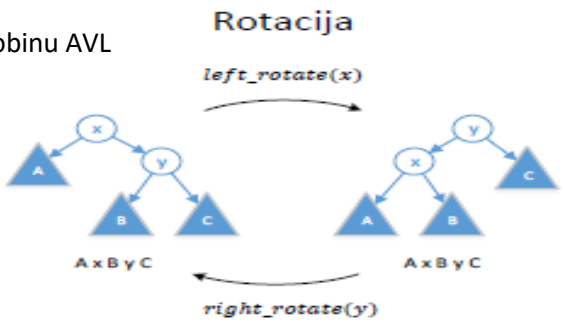
• Svaki elemenat stabla sadrži pokazivače na:
– Roditelja *p,*
– Levo dete *left,* i
– Desno dete *right.*
• Element u korenu stabla *x* ima polje *x.p* = NIL
• Element koji nema dece (ili ima samo jedno) polja *left* i/ili *right* postavlja na NIL.
• Struktura binarnog stabla sadrži pokazivač na koren stabla T.*root*
– Stablo bez elemetana ima T.*root* = NIL

Stablo - uopšteno

• Svaki roditelj može imati više dece što se može predstaviti na razne načine.
• Ukoliko uglavnom svi roditelji imaju jednak broj dece onda se elemenat stabla može proširiti poljima: *child1*, *child2*, ..., *childk* (umesto polja *left* i *right*)
• Ukoliko broj dece varira od elementa do elementa – tada se mogu upotrebiti samo dva pokazivača:
  – na levo (prvo) dete - *left*
  – na brata/sestru – *sibling*

25* AVL stablo

- Anderson – Velskz & Landis-ono stablo = samobalansirajuće BSP. Za svaki čvor visina levog i desnog podstabla se razlikuje maksimalno za +/- 1.
  -Svaki čvor pamti svoju visinu.
- Najgori slučaj je da je levo/ desno stablo za 1 čvor više od desnog/levog podstabla.
- Operacije sa stablom:
  -**AVL insert** - ubaciti čvor u BSP i popraviti osobinu AVL
  - **rotacija (rotation)**

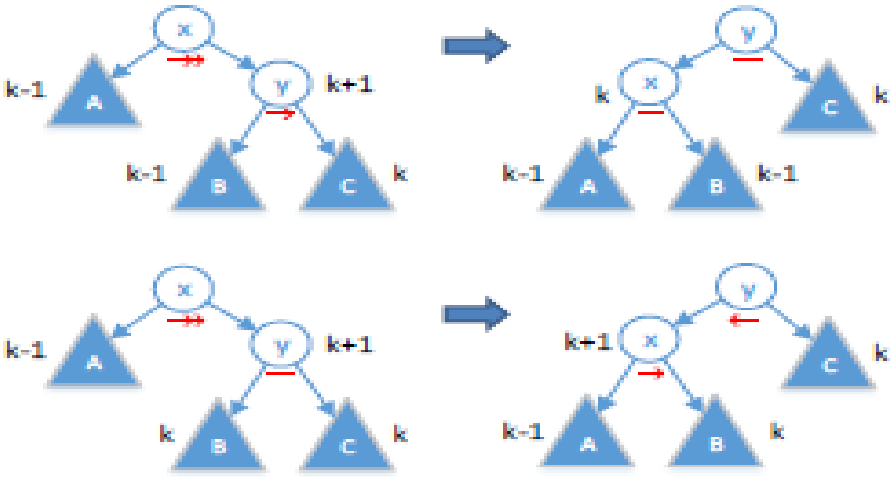


ALGORITAM ZA ISPRAVAK AVL OSOBINE

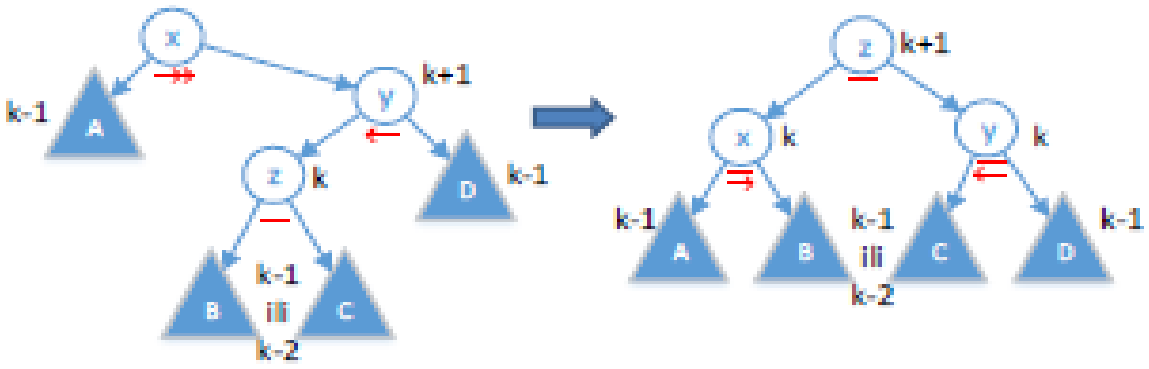
- Neka je x najniži čvor gde je narušena AVL osobina.
- Neka je desna strana od x koja je teža.
- Ako je desna strana od y teža ili je y balansiran, uraditi left_rotate, u suprotnom (leva strana od y je z i ona je teža) uraditi dve rotacije: right-rotate(y) i left rotate(x)

Nastaviti ispravke sa "dedom" od x i tako sve do vrha – dokle god je narušena AVL osobina

$left\_rotate(x)$



$right\_rotate(y)$ & $left\_rotate(x)$



26* Rečnik podataka i heširanje (primene, priheš, heš, kolizije, automatizovano vreme izvršavanja)

- **Rečnik podataka** (dictionary) je tip podataka(ADT-Abstract Data Type). Sadrži i održava skup elemenata gde svaki ima ključ(key).
  - Elemenat (item) se može osmatrati kao uređeni par {ključ, vrednost}
- ***Dictionary se koristi kada su potrebne brze operacije insertion, deletion i find-element, jer u teoriji te operacije se mogu izvesti u vremenu O(1).***

PRIMENE (verovatno najčešće upotrebljavana struktura podataka **)**

- baze podataka
- prevodioci: imena → promenljive
- rutiranje mrežnog saobraćaja: IP adresa → žica
- virtuelna memorija: virtuelna memorija → fizička lica
- pretraga podstringova (Google search)
- sinhronizacija sadržaja datoteka
- kriptografija
- IMPLEMENTIRAN u savremenim programskim jezicima
- Heš funkcije se mogu koristiti za skladištenje svih tipova podataka

PRIHEŠ - je funkcija koja uzima deo podataka kao ulaz, mi ćemo to zvati ključ (key), a kao izlaz daje ceo broj, poznatiji kao heš vrednost (hash-value). Praktično, ključ se prevodi u indeks. Dok je elemenat u tabeli, njgova priheš funkcija h_p se ne sme menjati, jer ga onda ne možemo pronaći.

$$h_p: k \rightarrow i$$

HEŠ - redukuje potencijalno velike vrednosti brojeva i mapira naš ključ u određeni indeks u heš tabeli. Na početku mi koristimo heš funkciju da odredimo gde u heš tabeli, da smestimo određeni ključ. Kasnije ćemo koristiti istu heš funkciju da odredimo gde u tabeli da tražimo zadati ključ. Iz ovog razloga potrebno je da se heš funkcija ponaša dosledno i da kao izlaz daje istu heš vrednos za identične ključeve.

- Problem sa kojim se suočavamo u formiranju rečnika jeste kolizija. **KOLIZIJA** je pojava kada se dva ključa mapiraju na isti indeks u tabeli.

Metode rešavanja kolizije:

1. **Heširanje ulančavanjem**
2. **Heširanje otvorenim adresiranjem**

Primer: Radimo sa stringovima.
- Recimo da naša heš funkcija daje heš vrednost koja je bazirana na osnovu prvog slova ključa
- reč “Alen” počinje, tako da je mapirana na poziciji 0 heš tabele
-slično “Boris”, je mapiran na indeks 1, a “Ciceron” je mapiran na indeks 2
- ako se pitamo da li je reč “Dragan” u tabeli, stavićemo istu u heš funkciju i kao izlaz ćemo dobiti indeks 3, pošto nema ništa smešteno na tom indeksu, i možemo reći da “Dragan” nije u tabeli, iako smo proverili samo jedan od 26 indeksa u tabeli
- U slučaju da u tabelu želimo da smestimo reč “Anđela”, heš funkcija će nam vratiti vrednost 0, kao za reč “Alen”. Ovo je primer za **koliziju**. To znači da se se rezultati dva ključa hešuju na isti indeks. Čak iako je naša heš tabela veća nego nego naš skup podataka i izabrali smo dobru heš funkciju, još uvek **treba da nađemo rešenje za rešavanje kolizije.**

27* Heširanje ulančavanjem

- Kod ove metode je u stvari niz pokazivača u spregnutoj listi. Kada se kolizija pojavi, ključ može biti ubačen u konstantnom vremen na glavu odgovarajuće spregnute liste. Kada želimo sada da potražimo reč "Alen" u našoj listi, u najgorem slučaju moramo da prođemo celu uvezanu listu, koja počinje na indeksu 0. Vreme izvršavanja za zadati ulaz u najgorem slučaju je O(n/k), gde je k veličina heš tabele. Iako je se n/k svodi na n, u stvarnom svetu je O(n/k) veliko poboljšanje u odnosu na O(n).

- **Jednostavno uniformno heširanje** – pp. svaki ključ ima podjednaku šansu da se mapira(heširа) u bilo koji red u tabeli, nezavisno od mesta gde se heširaju ostali ključevi. Definiše se faktor popunjenosti tabele (load factor) $\alpha = \mathrm{m}/\mathrm{n}$
  - **n** – broj elemenata koje smeštamo u T
  - **m** - broj mesta(redova) u T
  - α – određuje očekivani broj elemenata po mestu = očekivana dužina ulančane liste.

Preformanse: očekivano trajanje pretrage je $\Theta\ (1 + \alpha)$

- 1 za primenu heš funkcije i pristup redu tabele, a α za prolaz kroz listu.
- Ako je m = Ω(n), tada je $\alpha =$ O(1), pa je trajanje pretrage O(1)

- **Koliko treba da je m?** Ako je m veliko to je bačen prostor. m je promenljivo, po pokretanju malo, a može se po potrebi povećavati. **Kada je m = n tabelu treba duplirati.**

AMORTIZOVANO VREME IZVRŠAVANJA

- Operacija ima amortizovano vreme izvršavanja T(n), ako je za k operacija trajanje ≤ k*T(n). Grubo gledano amortizovano vreme je prosečno vreme za ponovljene operacije.

Operacije sa heš tabelom:

1. **Dodavanje –** amortizovano vreme izvršavanja je O(1). Kada se tabel
2. **Pretraga –** O(1) jer se održava m = Θ (n) i tada je α konstanta (ova osobina za α važi za uniformno
3. **Brisanje –** amortizovano vreme izvršavanja je O(1).
   -Kada n padne ispod m/4 tabelu treba prepoloviti
   -Ako se tabela prepolovi kada je n = m/2, a duplira kada je n = m+1, tada za n = m i niz operacija *insert, delete, insert, delete* ... dobijamo linearno vreme izvršavanja

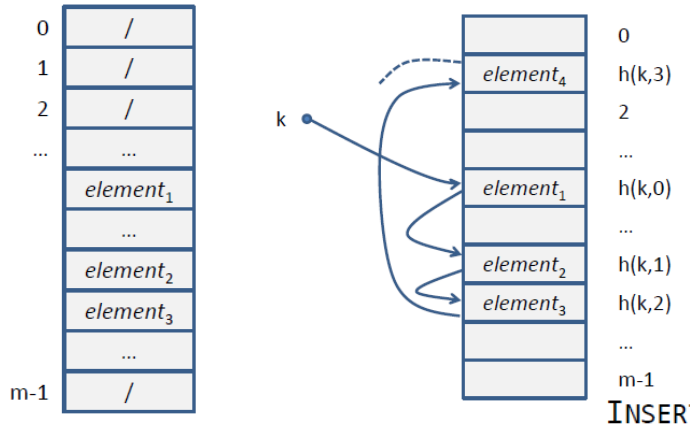
28* Heširanje otvorenim adresiranjem.

- Nema ulančanih elemenata. Najviše ide jedan elemenat po redu u T, tj. m >= n (n – broj elemenata, m – broj redova). Heš funksija je "proširena": **pored ključa koristi i broj pokušaja heširanja**.

h: U x {0,1,...,m - 1} --> {0,1,..., m - 1}

-h je funkcija koja mapira par <ključ, pokušaj>, na vrstu u tabeli T.

-Ideja: ako se pokušaj **i** povećava h(k, i) (a k se ne menja) biće adresirani svi redovi u tabeli.



Dodavanje elementa – uporno probati dok se ne pronađe prazan red, zatim dodati elemenat u red.

```
INSERT(k,v)
1 for i = 0 to m-1
2 if T[h(k,i)]==None        // prazan slot?
3   T[h(k,i)] = <k,v>       // sačuvaj el.
4   return
5 raise "puna tabela"
```

Pretraga – okušavati primenu heš funkcije dok se u mapiranom slotu nalazi vrenost različita od traženog ključa ili se ne natrči na prazan slot

```
SEARCH(k)
1 for i = 0 to m-1
2 if T[h(k,i)]==None          // prazan slot?
3   return None               // kraj „lanca"
4 elseif T[h(k,i)].key==k     // u slotu je ključ?
5   return T[h(k,i)]          // vrati elemenat
6 return None                 // pretražen ceo T
```

Brisanje – zbog zamišljenog ulančavanja koji su posledica kolizije ne može se jednostavno ukloniti elemenat iz reda jer "prekida zamišljeni lanac i tada pretraga neće raditi.

Rešenje: uvodi se posebna oznaka sa značenjem "obriši me" za svaki red(slot) u tabeli T

-dodavanje ignoriše oznaku(tretira je kao None), ali je pretraga tumači

-kod smanjivanja tabele označeni redovi se brišu.

Strategija heširanja sa "pokušajima"

- **Linearno dodavanje pokušaja (linear probing)**
  h(k,i) = h(h'(k) + i) mod m
  gde je h'(k) "obična heš funkcija.
  -Problem: dovodi do zauzeća uzastopnih redova – prave se zauzeti blokovi koji vremenom postaju sve veći.

- **<u>Duplo heširanje</u>**
  $h(k,i) = (h_1(k) + i* h_2(k)) \mod m$
  gde su $h_1(k)$ i $h_2(k)$ obične heš funkcije

PREDNOSTI I MANE OTVORENOG ADRESIRANJA I ULANČAVANJA

- Otvoreno adresiranje (OA) zauzima manje memorije(ne zahteva pokazivače)
- Ulančavanje je manje osetljivo na izbor heš funkcije

<u>Osobine dobre heš funkcije</u>

1.Koristi sve informacije koje daje ključ – u cilju da da maksimalni broj heš vrednosti (na primer kod reči je potrebno iskoristiti sva slova reči “vode”, “vodenast”, ako heš funkcija izbroji samo 3,4 slova pojaviće se kolizija, jer želimo da te dve reči stavimo na dve različite pozicije u tabeli)
2. Heš vrednosti treba da se šire jednako preko heš tabele – Ovo će smanjiti dužinu uvezanih lista
3.Mapira slične ključeve u što različitije heš vrednosti
4. Korišćenje samo brzih operacija – umetanje, brisanje i traženje elementa.

29 Grafovi (definicija, tipovi, primena, vrste algoritama)*

DEFINICIJA

- **Grafovi (graphs)** – se izučavaju u teoriji grafova (oblast diskretne matematike i računarske nauke). Graf je matematička strktura za modelovanje odnosa između parova objekata.
- **Graf se sastoji od**:
  - čvorova (objekata) – nodes, vertices &
  -grana (koje povezuju parove objekata) – edges
- **Matematička definicija**
  - Graf G = (V,E) se sastoji od skupa čvorova V, i skupa grana E.
  - Grane su dvoelementni podskup od V
  - Red grafa je broj čvorova |V|
  - Veličina grafa je broj grana |E|

TIPOVI

- Grafovi se dele na dva tipa:
  1. sa stanovišta usmerenosti grana:
     - neusmereni,
     - usmereni,
     - mešoviti (samo deo grana je orijentisan)
     - aciklični (nema "petlje"- kružne putanje)
  2. sa stanovišta parametara grana:
     - direktni
     - težinski

PRIMENA GRAFOVA

- Grafovi se upotrebljavaju za modelovanje mnogih praktičnih problema.
- Tipično modeliraju relacije i dinamiku procesa u fizičkim, biološkim, socijalnim i informatičkim sistemima.
- U računarstvu se grafovi koriste za predstavljanje računarske mreže, organizacije podataka i sl.

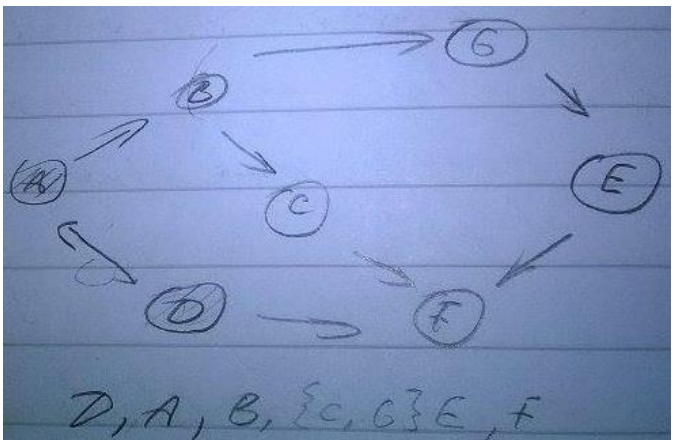
30 Topološko sortiranje*

- **Topološko soritranje usmerenog grafa** – je linearno uređivanje čvorova, tako da je čvor **u** ispred čvora **v**, kada ih spaja grana (u,v). Poredak ne mora biti jedinstven.

PRINCIP

1. Svakom čvoru, pridružujemo **broj ulaza = broj grana koje završavaju u čvoru**
2. Biramo čvor bez ulaza (broj ulaza je 0). Postavimo ga na kraj sortiranog reda, uklonimo ga iz grafa

- **Topološko sortiranje direktnog acikličnog grafa (DAG)** se može sprovesti na osnovu DFS-a.



31 BFS – pretraga u širinu*

- **BFS algoritam** je jedan od najednostavnijih algoritama za pretragu grafova. Predstavlja osnovu drugim algoritmima.

- Pretraga polazi od izvornog čvora i pokušava da dopre do svakog čvora koji je dostupan, dajući informaciju njgove udaljenosti (polje d u strukturi - distance) od izvornog čvora.
- Nazvan je po načinu rada gde se "front" pretrage širi tako da nakon obilaska čvorova na rastojanju **k** od izvora nastavlja sa otkrivanjem čvorova na rastojanju **k + 1.**
- Ako neki čvor iz matrice susedstva nema vezu sa ostalim čvorovima (predstava ostrva) taj čvor neće ni ulaziti u obradu.
- Za svaki čvor daje najkraći put od izvornog čvora.
- Algoritam radi sa orijentisanim i neorijentisanim grafovima.

PRINCIP RADA

- Posle inicijalizacije, algoritam nikad ne beli čvor. Na osnovu uslova u Queue se dodaju samo beli čvorovi.
- Ukupan broj operacija Enqueue i Dequeue složenosti O(1) je n, gde je n broj čvorova.
- Kada se čvor ubaci u **Q** onda se skenira njegova lista susedstva. To se radi za svaki dostupan čvor na istoj udaljenosti od predhodnog, te se skeniraju sve grane u grafu. (bez ostrva)
- Ukupno vreme izvršavanja algoritma je O (V + E).

32 DFS – pretraga u dubinu*

- **DFS algoritam** je koncipiran tako da pretražuje u dubinu sve grane koje se nalaze u grafu, bile one usmerene ili neusmerene algoritam radi. Pošto pretražuje sve čvorove grafa, za razliku od BFS-a pretražuje i ostrva, koja nisu u relaciji sa izvornim čvorom. (što je i jedna od primena DFS-a)
- Vremenske značke (polja) niza struktura V strukture G:
  G. – struktura
  |____Adj.matrix n x n matrix – matrica susedstva
  |____v 1xn – niz struktura, liste susedstva reda n
  |____(.d) - kada je otkriven čvor (discovery time)
  |____(.f) – kada je obrada čvora završena (finish time)
  |____(.color) – trenutno stanje (bela, siva, crna)
  |____(.pred) – ID predhodnog čvora

  PRINCIP RADA

- DFS za usmeren graf forsira preodiranje u dubinu po principu izbora manjeg ID –a u slučaju da jedan čvor ima više suseda. Kada stigne do kraja (neki čvor nema više suseda), taj čvor postaje crn i njegovo vreme se registruje u polju (.f), tada prelazi u suseda od (.pred) sa najmanjim ID – om. Gleda sve njegove susede i onda se rekurzivno vraća unazad, dok svi ne postanu crni. Prvi siv u (.d) biva poslednji crn u (.f) za tekuće ostrvo, zbog osobine rekurzije ovog algoritma. Kada obradi celo ostrvo prelazi na sledeće sa početnim najmanjim ID – om.

- Klasifikacija grana na osnovu DFS –a na usmerenom grafu:
  - Grane stabla (edges)
  - Preskočna grana (farward edge)
  - Povratne grana (backward edge)
  - Unakrsne grana (cross edge)

  CIKLUS

- Detekciju kružne grane vrši DFS – kružna putanja postoji ako postoji povratna grana.

33*Najkraći put u grafu (definicije, varijante, algoritmi)

DEFINICIJE

1. *Netežinski grafovi:*

- **BFS** – pronalazi najkraći put u netežinkom grafu – razmatra se broj grana

2. *Težinski grafovi:*

- Posmatra se usmereni težinski graf **G = (V,E,w)** gde su težine zadate funkcijom ***w : E --> R***
- Težina **w(p)** putanje **p = (v_0, v_1, v_2,....,v_k)** je suma težina grana te putanje, a cilj je pronaći **p** sa najmanjom težinom.

- **Dijkstra algoritam** – ograničava težine na nenegativne težine grana. Složenost je O(V*$\log_2$(V+E)).
- **Belman Ford** – graf može imati i pozitivne i nenegativne težine grana. Omogućava detekcije kružne putanje nenegativnog pojačanja. Složenost je O (V*E) i ona ne zavisi od težina w.

3. *DAG – umseren aciklični graf (linked topological-sort)*

- Algoritam dozvoljava grane negativne težine, ali to ne utiče na algoritam, jer DAG po definiciji nema ciklus (negativne) backward edges.

VARIJANTE NAJKRAĆEG PUTA

1. Najkraći put od zadatog čvora, do svih ostalih čvorova u grafu. (osnovni problem)
2. Najkraći put do zadatog čvora od svih ostalih čvorova u grafu (problem se rešava kao osnovni problem)
3. Najkraći put između zadatih čvorova i odredišta (rešava se kao osnovni problem)
4. Najkraći put između svih čvorova u grafu (može se rešiti na principu br. 2 za svaki čvor, ali se rešava posebnim algoritmom)

POTEŠKOĆE

- Grane sa negativnim težinama
  -Zašto imamo grane sa negativnim težinama?
  -Kako ih predstaviti kao pozitivne?
- Kružne putanje sa negativnim pojačanjem
  - Da li je moguće da se algoritam nikada ne završi jer se d.v stalno smanjuje?

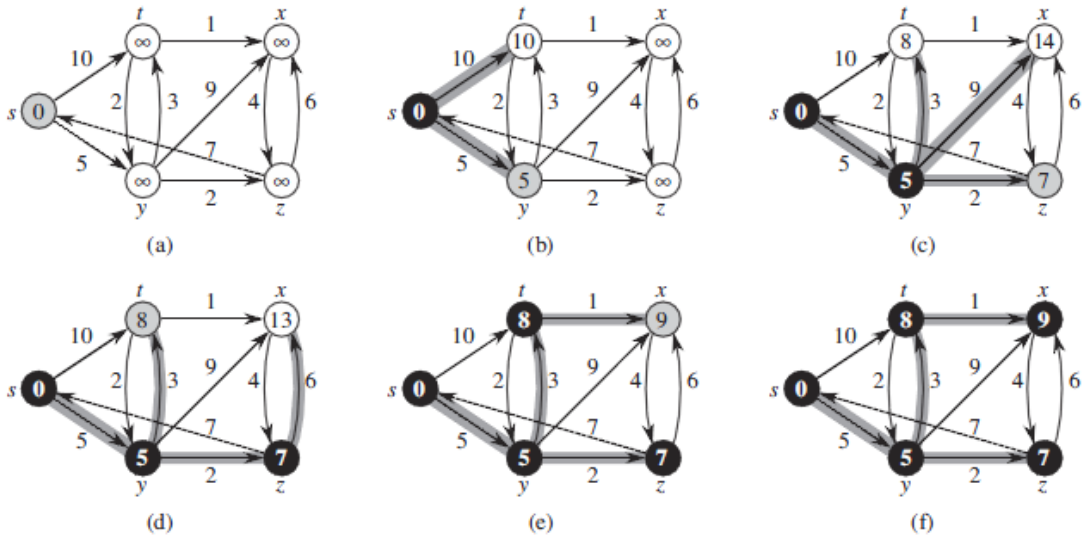
34* Dijkstra algoritam

- **Dijkstra algoritam** nalazi najkraći put id zadatog čvora u težinskom usmerenom grafu gde sve težine grana nisu nenegativne (ima negativnih). **Vreme izvršavanje je bolje od Belman- forda.**

PRINCIP RADA

- Algoritam održava skup čvorova **S** gde su rastojanja od polaznog čvora određena (neće se menjati). Zatim bira čvor **u ϵ(V\S)** sa najmanjim rastojanjem od **u.d**(prethodnog), dodaje ga u **S** i koriguje (relaksira) rastojanja do svih čvorova preko grana koje izlaze iz **u**.

- Inicijalizacija na početku:
  -početni čvor ima težinu 0,
  -ostali čvorovi su daleko ∞.
- Kretanje se dogaža prema svim susedima i koriguju se njihove trenutna rastojanja na nova (*relax*). Bira se komšija sa najmanjom težinom (udaljenošću) od početnog čvora i stavlja se u skup S. Gledaju ponovo svi njegovi susedi i radi se *relax*. Ako su neki čvorovi već obrađeni, a imaju veće težine od one koja bi trebale da dobiju od čvora koji se trenutno obrađuje, ponovo se radi njihov *relax* i koriguje se ratojnje.



VREME IZVRŠAVANJA

- Algoritam održava red sa minimalnom prioritetima (min – priority queue)
  |V|=n |E|=m

1. Implementacija Q nizom indeksiranim rednim brojem čvora (1, 2, ...,n):
   - Operacije:
     - Insert = O(1)
     - Extract – min = O(n) – jer se pretražuje ceo niz
     - Decrease key = O(1)
   - Vreme izvršavanja: $O(n^2)$
2. Implementacijom binarnog min-heap sve operacije traju $O(\log_2 n)$, a vreme izvršavanja je $O((m+n)*\log_2 n)$

- U slučaju da je graf redak bolji je drugi način

35* Bellman Ford algoritam

- Rešava problem traženja najkraćih puteva za svaki čvor i kada postoje nenegativne težine grana otkriva postojanje kružnih putanja nenegativnog pojačanja.
- Radi u polinomskom vremenu O(V*E)

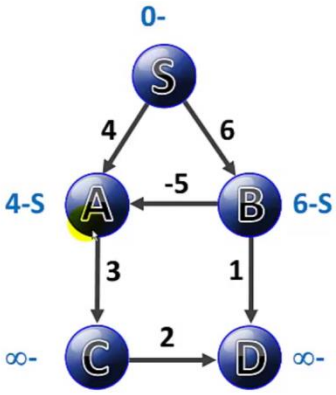
PRINCIP RADA

- Radi tako što stalno koriguje(relaksira) sve grane u grafu, odnosno testira da li može da poboljša najbolju putanju do traženog čvora kroz iteracije u grafu. Za jednu iteraciju algoritam je potrebno da prođe kroz sve putanje koje su određene usmerenjima i težinama grana i u iteracijama se teži poboljšanju trenutnih stanja čvorova. Iteracija ima onoliko koliko ima čvorova.

primer

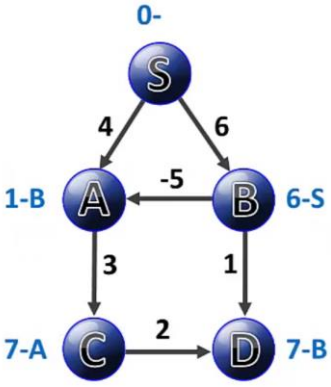
1.

A-C	3
B-A	-5
B-D	1
C-D	2
S-A	4
S-B	**6**



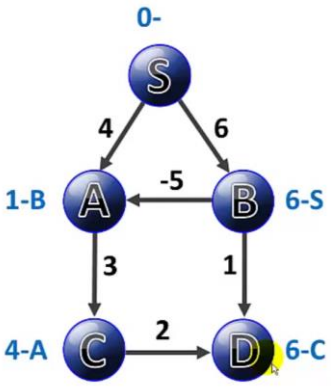
2.

A-C	3
B-A	-5
B-D	1
C-D	2
S-A	4
S-B	**6**



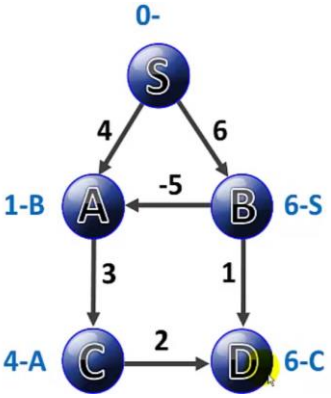
3.

A-C	3
B-A	-5
B-D	1
C-D	2
S-A	4
S-B	**6**



4.

A-C	3
B-A	-5
B-D	1
C-D	2
S-A	4
S-B	**6**



- Dogodila bi se još jedna iteracija provere ali je u ovom slučaju nepotrebna tako da može da se napusti loop ranije.

- Na kraju se proveravaju negativne kružne putanje, jer ako one postoje onda algoritam nije dobar.

$d_C \leq d_A + w \mid 4 \leq 1 + 3$
$d_A \leq d_B + w \mid 1 \leq 6 - 5$
$d_D \leq d_B + w \mid 6 \leq 6 + 1$
$d_D \leq d_C + w \mid 6 \leq 4 + 2$
$d_A \leq d_S + w \mid 1 \leq 0 + 4$
$d_B \leq d_S + w \mid 6 \leq 0 + 6$

36* Složenost računanja klase problema (P, N ,EXP, R, problem odlučivanja, primeri)

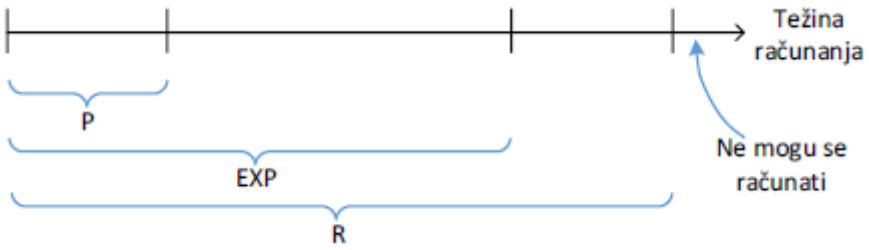
- Probleme ćemo za početak podeliti u dve "interne" kategorije:
  1. **Jednostavni problemi** – do sada smo posmatrali samo njih. To su pretrage sortiranja, obilazak grafa,.. Za jednostavne probleme postoje efikasni algoritmi lija je vremenska složenost O(P(n)) ograničena nekim polinomom P(n), zavisan od veličine ulaza n. Klasu efikasnih algoritama označavamo sa P jer imaju polinomsko vreme izvršavanja($\sim n^c$).
     -U ovu klasu problema spadaju i problemi čija je složenost npr. $O(n^{10})$, $O(n^{100})$,... To nisu brzi algoritmi, a i izmišljeni su!
     -U praktičnoj primeni je obično mali stepen vremenske složenosti p algoritama (npr. n^2)

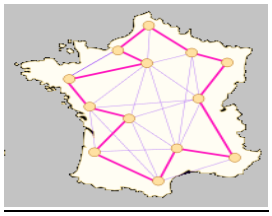
  Takođe, rešenje P problema se može jednostavno proveriti

  2. **Složeni problemi** - su svi oni problemi koji nemaju rešenje u obliku jednostavnih algoritama, što znači ne mogu se rešiti u polinomskom vremenu. Postoji dosta algoritama za čije rešenje se ne zna ni jedan algoritam polinomske složenosti.

OSNOVNE KLASE PROBLEMA

- **P –** problemi rešivi u **polinomskom vremenu** ($\sim n^c$)
- **EXP** – problemi rešivi u **eksponencijalnom** vremenu ($(\sim 2^n)^c$)
- **R** – problemi rešivi u **konačnom** vremenu
- **Nerešivi problemi**





PRIMERI SLOŽENIH PROBLEMA

1. **Problem "Trgovačkog putnika".**Preduzeće koje isporučuje pošiljke koristi vozila za prevoz. Svako vozilo u radnom danu treba da preveze n paketa na n lokacija i da se vrati u garažu.
   -To znači da vozilo treba da poseti n + 1 lokaciju.
   -Predpostavimo da se za prevoz iz svake u svaku drugu lokaciju poznaju troškovi prevoza.
   (tabela sa (n + 1)*(n+1) vrednosti)
   -Treba odrediti rutu koja polazi i završava se u garaž, a da su ukupni troškovi prevoza što je moguće manji!
   - Koliko je težak ovaj problem? Koliko ima različitih putanja? Broj putanja je n! **(faktorijel)**
   - Realni podaci: vozilo u proseku prevozi 170 paket od kojih se nekoliko isporuči na istu adresu. Neka se radi o svega 20 različitih adresa, što daje 20! = 2.432.902.008.176.630.000 putanja.
     -Ako program generiše/procesira 10^{12} (nerealno velik broj) putanja svake sekunde treba mu mesec dana da obradi svoje putanje!
     -Ako program generiše/procesira 10^{9} (i dalje veliki broj) putanja svake sekunde treba mu oko 1000 meseci!
     - Ta obrada se odnosi samo na jedno vozilo i jedan dan. Veliki isporučioci imaju desetine hiljada vozila i rade svaki dan u nedelji.
   - Rešavanje problema analizom svih mogućih putanja nema smisla, jer dok pronađemo odgovarajuću koja daje najmanje troškove – nije praktičan algoritam!
   - **Postoji polinomsko rešenje ovog algoritma, ali još ne postoji algoritam. Isto tako, nije dokazano ni da ne postoji algoritam. Mi kao inženjeri uzimamo rešenje koje nije najbolje ali je dovoljno dobro.**
     - Za ovu klasu problema postoje brojni aproksimativno algoritmi. Npr. aproksimativni algoritam za problem trgovačkog putnika radi u polinomskom vremenu, a daje putanju koja je 50% duža od optimalne.

2. **Otkrivanje zatvorene putanje u usmerenom grafu čije je pojačanje negativno ϵ P.**
3. **Šah na tabli n x n $\in$ EXP (ne pripada $\notin$ P)**
4. **Tetris $\in$ EXP, ali se ne zna da li je $\in$ P (Da li se sekvenca datih oblika može preživeti)**
5. **Halting problem $\notin$ R**
   - Da li se neki kompijuterski program završava ili se beskonačni izvršava?
   - Ni jedan algoritam ovo ne rešava korektno u konačnom vremenu za proizvoljni dati program(ulaz)
   - Ujedno je i problem odlučivanja čiji je izlaz {Da, Ne}
6. **Traženje najkraćeg/najdužeg puta u grafu.**
   - Traženje najkraćeg puta u grafu G(V,E) od datog čvora je algoritam složenosti O((m + n)*$\log_2$n)
   -Traženje nadužeg puta u grafu između dv čvora je NP – kompletan problem
7. **Ojlerova putanja/Hamiltonov ciklus u grafu**
   -Ojlerova putanja u grafu je kružna putanja koja prolazi kroz svaku granu grafa(tačno jednom) i pri tome dozvoljava višestruke posete istom čvoru. Grane ojlerove putanje se mogu odreditu u O(m) vremenu.
   -Hamiltonov ciklus je zatvorna putanja koja sadrži svaki čvor grafa sadrži tačno jednom

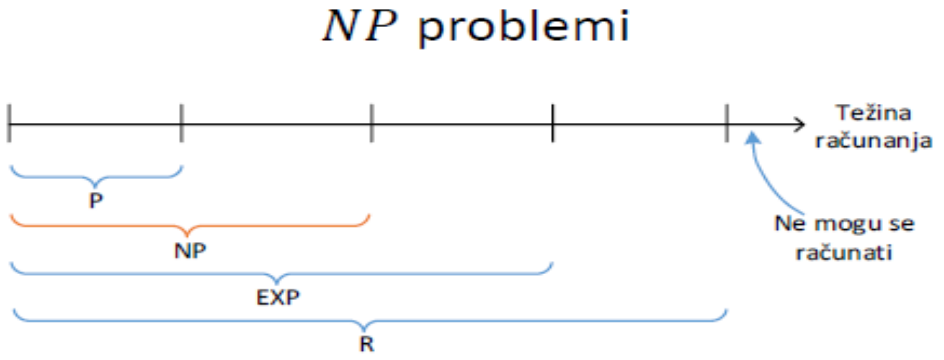
PROBLEM ODLUČIVANJA

- Problem odlučivanja je funkcija $f: \mathbb{N} \to \{0,1\}$
  - Ulaz je binarni string ~ celobrojan nenegativan broj $\mathbb{N}$
  -Izlaz je {Da, Ne} {0,1}
- Ovo se može posmatrati kao mapiranje gde se svakom mogućem ulazu dodeljuje 1 bit. Time se dobija beskonačan niz bita (0.1101101001...) što je definicija realnog broja, tj. problema odlučivanja ima $|\mathbb{R}|$.
  - Ako posmatramo sve moguće programe (program ~ konačan binarni string) i sve moguće probleme ispada da je broj programa mnogo manji od problema $|\mathbb{N}|<<|\mathbb{R}|$. Ovo je teorijski interesantno, ali srećom, većina praktičnih problema ima rešenje.

37* NP problemi (definicija P i NP problema, klase NP problema, redukcija)

DEFINICIJA

- NP problem je podskup **problema odlučivanja.** Rešiv je u polinomskom vremenu uz upotrebu **nedeterminističkog modela računanja** (nedetrminističke mašine).
  -Gde svako računanje ima grananje i ona se mogu jednovremeno sračunati.
- Nedeterministički model računanja se mže sprovoditi na determinističkoj mašini(~klasičan računar), ali sporo(rekurzivno, bez mogućnosti jednovremenog računanja). Deterministička mašina bi radila fikasno kao i nedeterministička kada bi kod svakog odlučivanja "srećno" pogodila granu (put) koji vodi ka rešenju (bez probanja svih opcija kod grananja).
- **Kod NP problema se ne zna algoritam za "brzo" dobijanje rešenja (rešenje se ne dobija u polinomskom vemenu), ali se dobijeno rešenje može proveriti pomoću determinističke mašine u polinomskom vremenu.**
  Primeri:
  1. **Problem sume podskupa.** Dat je skup celih brojeva. (Da li prostoji neprazan podskup čija je suma nula?
     -Na skupu {-2, 3, 15, 14, 7, -10} postoji podskup {-2,-3,-10,15} čija je suma 0
     -Jednostavno se proverava – u linearnom vremenu (treba samo sabrati sve elemente podskupa).
     -Teško se pronalazi jer treba probati $2^n - 1$ kombinacija (što je eksponencijalno vreme).
  2. **Tetris ∈ NP (za datu listu ulaza određuje da li će igrač "preživeti")**
     -Nedeterministički algoritam – odluka kod svakog poteza
     -Dokaz za "preživljavanje" je jednostavan – treba imlementirati ponašanje poteza i primeniti ih.



- **Svaki P problem je ujedno i NP problem je ne samo da se za dato rešenje P problema može proveriti da je ono konkretno, nego se za P vreme može i pronaći takvo rešenja.**

- **Milenijumski problem: P = NP ili P ≠ NP.** Da li za svaki problem za koji neki algoritam može brzo da se proveri dato rešenje (u polinomskom vremenu) takođe postoji algoritam koji brzo pronalazi takvo rešenje (u polinomskom vremenu) tj. da li su svi NP problemi ujedno i P?
  - Većina smatra da je P≠ NP ali za to nema dokaz.
  - U načelu traženje rešenja problema (ili dokazivanje da je rešenje korektno) je teže od njegove same provere.
  - Jedan od 7 matematičkih problema definisan 2000. godina od strane Clay Mathematics instituta za korektno rešenje donosi nagradu od 1 milion $.

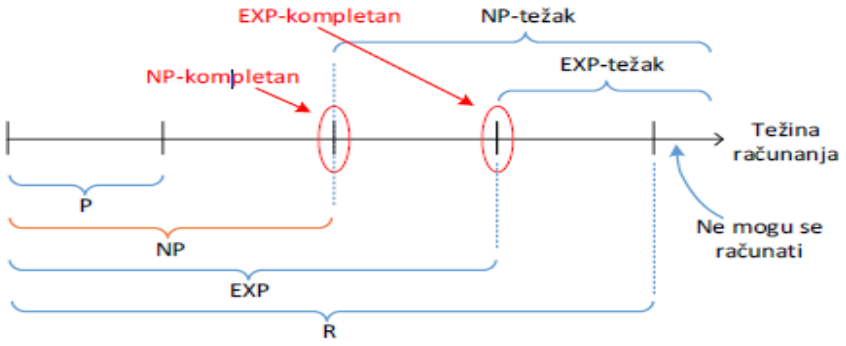
KLASE NP PROBLEMA

1. **NP – težak (NP –hard)** je problem težak kai svaki NP problem. Ne mora biti NP problem jer se njihova rešenja mogu proveriti u polinomskom vremenu.
   VRATI SE
2. **NP – kompletni prolemi (NP – complete)** su " najteži u NP klasi problema.
   Problem je tipa NP – kompletan ako zadovoljava 2 uslova:
   - **a)** On je podklasa NP problema (rešenje se može proveriti u polinomskom vremenu).
   - **b)** Ako za problem postoji algoritam koji se izvršava u polinomskom vremenu, onda postoji način da se svaki NP problem konvertuje u taj problem na način da se svi oni izvršavaju u polinomskom vremenu, tj. efikasan algoritam za NP – kompetan problem postoji akko za svaki NP – kompletan problem postoji efikasan algoritam.

Rasprostranjeno je verovanje da NP – kompletni problemi nemoju efikasan algoritam, ali to nije dokazano, a ni obrnuto. Danas se za dosta problema zna da su klase NP – kompletni.
Postoje u brojnim oblastima: *bulova logika, aritmetika, dizajn mreža, skupovi i particionisanje, skladištenje i očitavanje podataka, algebra i teorija brojeva, teorija igara, slagalice, automati, optimizacija, biologija, hemija, fizika,...*
NP – kompletni problemi se rešavaju tehnikama koje daju približno rešenje – rešenja nisu optimalna (najbolja moguća).
-Približno rešenje je bolje nego nikakvo.
-Približno rešenje je obično dovoljno dobro.

Primeri NP –kompletnih problema

1) Problem ranca (Knapsack problem) – je problem kombinatorne optimizacije, takav da u ranac poznate nosivosti staviti što je veći broj stvari (maksimalno) mokuće.
2) Hamiltonov put – problem teorije grafova. Da li postoji putanja u grafu koja u kojoj se svaki čvor posećuje samo jednom?
3) Trgovački putnik – problem kombinatorne optimizacije, gde je dat spisak gradova i mežusobnih rastojanja. Koja je najkraća putanja gde se svaki grad posećuje samo jednom i vraća se u polaznu tačku.
4) Izomorfizam podgrafova – data su dva grafa. Da li je drugi sadržan u prvom (da li je njegov podgraf jednak drugom podgrafu)?
5) Problem sume podskupa – Dat je skup celih brojeva. Da li postoji neprazan podskup čija je suma nula?
6) Klik problem – problem teorije grafova. Traži se maksimalan podgraf gde su svi čvorovi međusobno povezani saki sa svakim.
7) Bojenje grafa – teorija grafova. Traži se minimalan broj boja u grafu kojima možemo obojiti čvorove tako da susedni čvorovi budu različitih boja.
8) Najduža zajednička podsekvenca od n stringova (DNK).
9) Puzzle igre

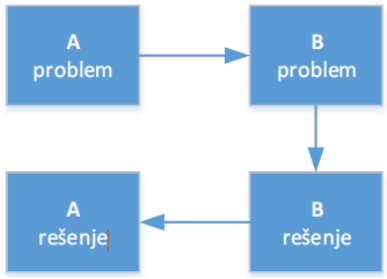


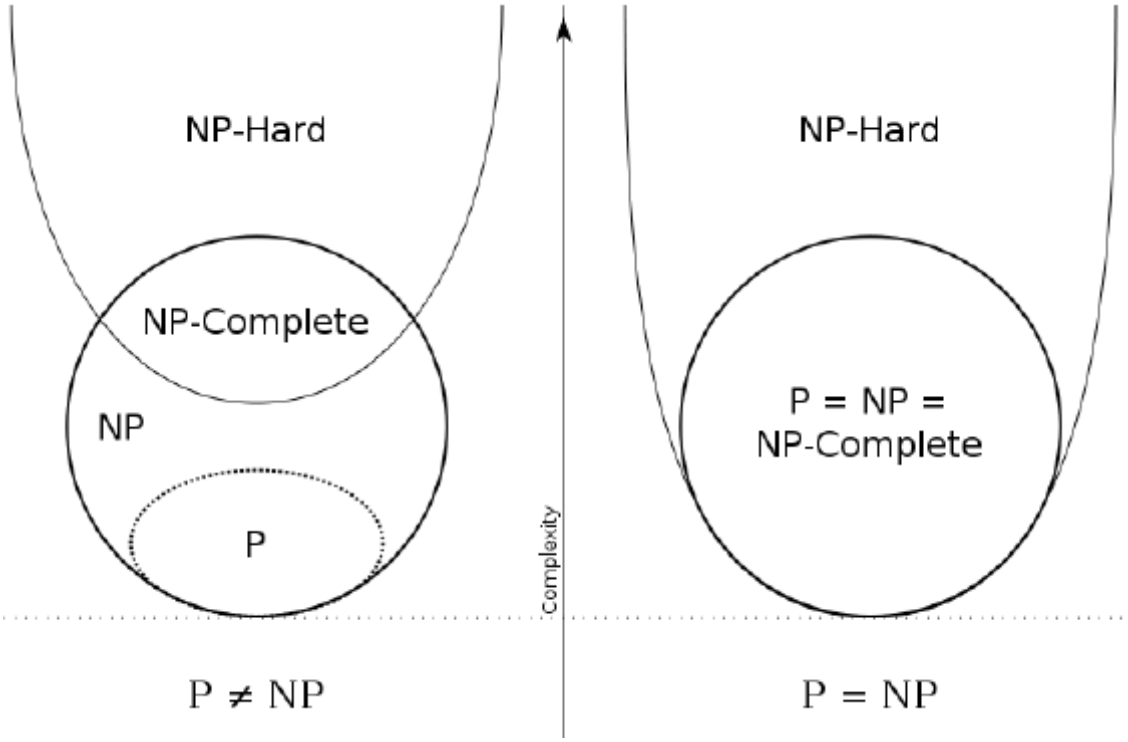
- **<u>Kako znamo da je NP problem kompletan?</u>**
  - Kada određujemo da je problem NP – kompletan, onda definišemo koliko je problem težak, uz ogradu da se radi o problemima odlučivanja. Pokušavamo da dokažemo da je za očekivati da ne postoji efikasan algoritam koji rešava problem

  - *<u>Tri koncepta se koriste da se pokaže da je problem NP – kompletan:</u>*
    1) Problem se svodi na problem odluke (rešenje problema svodi na odgovor"da/ne").
    2) Problem se **<u>redukuje</u>** (u polinomskom vremenu) na drugi problem za koji se predpostavlja da je iz klase NP i tako redom dok se transformacijama ne svede na "*prvi NP – kompletan problem*".
    3) na "*prvi NP – kompletan problem*" je problem za koji je dokazano da je NP – kompletan

  - Kada se za neki problem proglasi da je NP – kompletan, onda je to dovoljan dokaz da ga za sada ne možemo efikasno rešiti. Ne traži se uzaludno brz algoritam, već se energija usmerava ka razvoju algoritma za aproksimativno rešenje. Mnofi često sretani i interesantni problemi se čine od problema sortiranja, graf pretrage i sl., ali su NP – kompletni problemi

<u>REDUKCIJA</u> – je konvertovanje problema koji se rešava, u problem za koji imamo rešenje. Najčešće upotrebljavana tehnika rešavanja problema.





*38*Sekvencijalni i paralelni algoritmi(opis, Amdalov zakon)*

OPIS

1. **Paralelni algoritmi** – u poslednje vreme je jedan od glavnih pravaca razvoja računarstva, gde se u računaru sa procesorom od više jezgara i/ili više procesora. Razvijaju se različite tehnike i modeli izračunavanja koje u osnovi omogućavaju jednovremeno izvršavanje delova algoritma.
2. **Sekvencijalni algoritmi** – su svi algoritmi sortiranja i pretrage grafova koje smo do sada radili. Osnovne mere složenosti sekvencijalnih algoritama su:
   -vreme izvršavanja
   -veličina upotrebljene memorije.
   Dok, kod paralelnih algoritama se gledaju isti indikatori složenosti, al i se vodi računa i o drugim resursima: npr. broj angažovanih procesora.

- *Nisu svi problemi pogodni za rešavanje paralelnim algoritmima!* Suštinski sekvencijalni problemi se ne mogu brže izvršavati ni kada imamo neograničen broj procesora.
- Većina algoritama se može makar delom "paralelizovati", ali ne u svim delovima
  npr. problem koji je idealan za paralelizovanje će se 2 x brže izvršavati upotrebom dva procesora u odnosu na upotrebu jednog procesora(to nije realno očekivati!).

AMDALOV ZAKON (Amdahl's law)

- Ovaj zakon daje teorijski maksimalno ubrzanje obrade algoritma kada se upotrebi više procesora. Vreme izvršavanja upotrebom više procesora po zakonu iznosi:

$$T(n) = T(1)\left(\mathrm{B} + \frac{1}{n} + (1 - B)\right)$$

n – broj procesora (niti izvršavanja),
B - ϵ [0,1] – deo algoritma koji je sekvencijalan,
T(1) – vreme izvršavanja upotrebom jednog procesora.

Odnosno ubrzanje iznosi: $S(n) = \frac{T(1)}{T(n)} = \frac{1}{\mathrm{B} + \frac{1}{n} + (1-B)}$

Primer amdalovog zakona

- Neka je za izvršavanje algoritma na jednom procesoru potrebno 30 min. Taj algoritam ima deo od 3 min koji se ne može paralelizovati, dok se preostalih 27 min izvršavanja (90%) može paralelizovati.
- Maksimalno ubrzanje ne može biti bolje od 10 puta:

$$S(n) = \frac{1}{\left(0.1 + \frac{1}{n}\ 0{,}9\right)} = 10$$

- Ubrzanje sa 4 procesora je 3.08 puta:

$$S(4) = \frac{1}{\left(0.1 + \frac{1}{4}\ 0{,}9\right)} = 3.08$$

39 Paralelni algoritmi (primeri i poteškoće implementacije)*

PRIMERI

1) Suma elemenata niza (srednja vrednost niza brojeva)
2) Traženje maksimuma (ili minimuma)
3) Sortiranje – merge sort
4) Elementarne matrične operacije
5) Stablo razapinjanja u grafu
6) Većina NP – problema se rešava "pametno osmišljenom" grubom silom
7) etc.

Poteškoće implementacije paralelnih rešenja

- Paralelni algoritmi se oslanjaju na paralelno programiranje i/ili distribuirano programiranje.
- Pored vremena potrebnog za izvršavanje (korisnog dela) algoritma potrebno je dodatno vreme za:
  -Inicijalizaciju i/ili sinhronizaciju izvršavanja niti,
  -Sinhronizaciju pristupa deljenim promenljivim,
  -Komunikaciju između niti (kod distribuiranih rešenja).
- Nekada je trajanje "dodatnih aktivnosti" značajno u odnosu na vreme izvršavanja algoritma. Treba znati još da je broj raspoloživih procesora ograničen, kao i da je angažovanje dodatnih procesora "trošak".

***40* Kriptografija (primena*, procesi, šifrovanje i bezbednosne pretnje)**

- **Kriptografija** je nauka koja se bavi metodama očuvanja tajnosti informacija šifrovanjem. U bitnim programskim sistemima iservisi se moraju očuvati od bezbedonosnih pretnji.

Osnovni pojmovi:

- poruka (P) – podatak ili deo podatka koji strana A šalje strani B.
- šifrovana poruka (C) – poruka koja je promenjena sa ciljem da bude nerazumljiva za potencijalne napadače.
- ključ (K) – tajni podatak koji omogućava stranama da poruke šifriraju.
- šifrovanje(enkripcija) (E) – proces u kojem predajna strana modifikuje poruku da ona bude nerazumljiva za potencijalne napadače. Rezultat ovog procesa je šifrovana poruka.
- dešifrovanje (dekripcija) (D) – proces u kojem se od šifrovane poruke pravi izvorna poruka.

- Bezbednost u računarskim sistemima je usko je povezana sa pojmom zavisnosti (*dependability*). Između ostalog zavisnost uključuje:
  -**Tajnost** (*confidentiality*) – informacija se daje samo autorizovanim(ovlašćenim) licima/servisima
  -**Integritet** (*integrity*) – modifikacija podateka zahteva autorizovan pristup.

BEZBEDONOSNE PRETNJE

1. **Presretanje** (interception) – neautorizovana strana dobije pristup podacima ili servisima.
2. **Prekidanje** (interruption) – servis ili podatak postane nedostupan, neupotrebljiv ili uništen. npr. denial of service (DoS).
3. **Modifikacija** (modification) – neautorizovana izmena podataka ili izmena servisa tako da više nije u skladu sa specifikacijom.
4. **Fabrikacija** (fabrication) – generisanje dodatnih podataka ili aktivnosti koji ne bi postojali u normalnim situacijama npr. ilegalno ponavljanje ranijeg zahteva za prenos novca, dodavanje zapisa u datoteku sa lozinkama, itd.

- Prekidanje, modifikacija i fabrikacija se mogu posmatrati kao *falsifikovanje podataka*.

MEHANIZMI SPROVOĐENJA BEZBEDNOSTI

- Za bezbedan sistem potrebno je definisati bezbedonosne zahteve. **Bezbedonosna politika (*security policy*)** sistema tačno opisuje šta je korisnicima, servisima i računarima u sistemu dozvoljeno, a šta nije.

Mehanizmi sprovođenja bezbednosti su:

1. **Šifrovanje** (encryption) transformiše podatke u format koji nije razumljiv za napadače.
2. **Autentifikacija** (authentification) – verifikacija identiteta korisnika, klijenata, servisa, itd.
3. **Autorizacija** (authorization) – provera da li korisnik ili servis ima pravo na izvršavanje akcije
4. **Beleženje istorijata aktivnosti** (auditing) – upisivanje u dnevnike događaja (uglavnom tekstualne) koji je entitet čemu pristupio i na koji način.
   - Ovaj mehanizam ne obezbeđuje direktnu zaštitu od napada, ali omogućava naknadnu analizu bezbedonosnih problema.

ŠIFROVANJE

- Šifrovanje je transformacija informacije tako da je njeno pravo značenje sakriveno. Potrebno je "posebno znanje" da se preuzme informacija. Šifrovanje koristi ključeve kao "posebno znanje". Savremeni algoritmi sprečavaju pokušaje otkrivanje kluča gubom silom.

Primer:

- Ako strana A želi da pošalje poruku m strani B, onda da bi zaštitila poruku od bezbednosnih pretnji:
  a) A će izvršiti šifrovanje (enkripciju) poruke m (dobija se m')
  b) A će poslati šifrovanu poruku (m')
  c) B će izvršiti dešifrovanje (dekripciju) poruke m' i dobiće m

*41*Osnovni algoritmi kriptografije (podela, namene osnovnih algoritama)*

- Primitivan aloritam šifrovanja je način kodiranja teksta gde se neko slovo zamenjuje drugim, a dekodiranje ponovi zamenu.

PODELA

a) **Simetrična kriptografija** – kriptografski sistem sa deljenim ključem. Koristi isti ključ i za šifrovanje i dešifrovanje. Da bi komunikacija bila bezbedna ključevi moraju biti tajni(kao i kod svih kriptografskih sistema). Oznaka za daljeni ključ je: $\mathbf{K_{A,B}}$

$$C = E_{K_{A,B}}(P) \leftarrow \text{Na predajnoj strani}$$

$$P = D_{K_{A,B}}(C) \leftarrow \text{Na prijemnoj strani}$$

b) **Asimetrična kriprtografija** – Odvojeni ključevi za šifrovanje i dešifrovanje. Svaki od učesnika ima dva ključa: javni i tajni ključ. Tajni ključ K_D (zove se i privatni ključ) nikome se ne daje. Javni ključ K_E može posedovati svako. Ovakvi sistemi se zovu i "sistemi sa javnim ključem"

Princip rada:

-Proces šifrovanja koristi javni ključ K_E da napravi šifrovanu poruku

-Takvu poruku može da pročita samo proces dešifrovanja koji koristi tajni ključ.

$$P = D_{K_D}(E_{K_E}(P))$$

OSNOVNI ALGORITAMI (NAMENE)

- Tekstualne poruke nisu jedine u računarskoj komunikaciji i kodiranje zamenom slova nije dovoljno sigurno. Ako se koristi ShiftChiper dovoljno je probati samo 26 slučajeva alfabeta. Kriptografija Koristi binarnu predstavu poruka i često se u algoritmima kodiranja koristi ekskluzivno ILI, tj binarna operacija (XOR).

Tablica istinitosti za XOR:	Osobine (x, y su biti):
$0 \oplus 0 = 0$	$x \oplus 0 = x$
$0 \oplus 1 = 1$	$x \oplus 1 = \sim x$
$1 \oplus 0 = 1$	$(x \oplus y) \oplus y = x$
$1 \oplus 1 = 0$	

1. **"Binarna podloga" – One- Time Pad algoritam**

- Posatramo niz bita koji čine poruku **p**.
- "Binarna podloga" **k** je ključ – slučajno izabran niz bita iste dužine kao poruka.
- Poruku "postvimo na podlogu" i dobijamo kodiranu poruku c = p XOR k
  -Operacija XOR se primenjuje na svaki par bita k_i i p_i.
  -Kodirana poruka je u obliku gde se teško može rekonstruisati neka dodatna informacija u originalnoj poruci.
- Prijemna strana takođe poseduje isti ključ i poruku postavlja na podlogu p_2 = c XOR k = (p XOR k)XOR k = p
- Osobine algoritma su da čini kodiranu poruku bezbednom, koristi ključ dužine poruke, i ponovna upotreba ključa je rizična.
- ***Edicija na One-Time Pad*** jeste takva da se poruke seckaju, prave se kraći ključevi. Ključ se primenjuje na svaki od blokova po principu da se za obradu narednog bloka za ključ koristi rezultat prethodnog bloka tzv. "ulančavanje blokova".

2. **AES (Advanced Encryption Standard)** – je javno dostupan i besplatan algoritam šifrovanja simetričnim ključem. Koristi ključ od 128, 192, ili 256, a blokove dužine 128 bita. Nastao je 2001. godine nakon četvorogodišnjeg takmičenja i saradnje američke vlade, privatne industrije i naučnika za izbor najboljeg algoritma koji je:

- *Bezbedan* – otporan na kritoanalizu, ispravnosti matematike, slučajnost izlaza, itd.
- *Jeftin* – brz i ima male memorijske zahteve.
- *Dobrih karakteristika* – jednostavan, fleksibilan, pogodan za hardversku implementaciju.

- Zamenio 3DES algoritam.

42* RSA algoritam (principi, uključeni algoritmi)

- RSA je algoritam šifrovanja asimetričnim ključem. Algoritam se oslanja na broj koji je proizvod 2 velika prosta broja(dovoljno velik da niko ne može otkriti činioce u razumno dugom vremenskom periodu. npr. broj sa 2048 bita je dovoljno velik). Naziv je dobio po tvorcima Ronald Rivest, Adi Shamir i Leonard Adelman.

PRINCIPI RADA

1. Izabrati 2 velika (najmanje 1024 bita svaki) različita prosta broja p i q.
2. Izračunati n = p*q
3. Izračunati r = (p-1)*(q-1)
4. Izabrati mali broj e tako da su e i r uzajamno prosti
5. Izračunati d kao inverzan elemenat za množenje, tj. da je e*d mod r = 1
6. RSA javni ključ je K_E = (e,n)
7. RSA privatni ključ je K_D = (d,n)
8. Funkcije šifrovanja i dešifrovanja su : $E_{K_E}(x) = X^e \bmod n$, : $D_{K_D}(x) = X^d \bmod n$

Primer sa malim brojevima:

1. Izabrati p = 17 i q = 29
2. Izračunati n = p*q = 493
3. Izračunati r = (p – 1)*(q – 1) = 448
4. Izabrati mali broj e = 5 jer su 5 i 448 uzajamno prosti.
5. Izračunati d = 269
   provera: e*d mod r = (5 * 29 mod 448) = 1345 mod 448 = (3 * 448 + 1) mod 448 = 1
6. RSA javni ključ je je K_E = (5,493)
7. RSA privatni ključ je je K_D = (269, 493)
8. Primer šifrovanja i dešifrovanja:

$$E_{K_E}(327) = 327^5 \bmod 493 = 3.738.856.210.407 \bmod 493 = 259$$
$$D_{K_D}(259) = 259^{269} \bmod 493 = \cdots = 327$$

UKLJUČENI ALGORITMI

1. **Računanje po modulu:**
   Primer: (23 + 7) mod 11 = 30 mod 11 = (2*11 + 8) mod 11 = 8

- Koristi se u nekim algoritmima šifrovanja, a osobine su mu:
  - $(a + b) \bmod n = ((a \bmod n) + (b \bmod n)) \bmod n$
  - $ab \bmod n = ((a \bmod n)(b \bmod n)) \bmod n$
  - $a^b \bmod n = (a \bmod n)^b \bmod n$
  - $xn \bmod n = 0$

2. **Generisanje velikih prostih brojeva**

- Prosti brojevi su prirodni brojevi deljivi samo sa 1 i sa samim sobom.
  Primer: 1, 2, 3, 5, 7, 11, 13,.... recimo 6 nije prost broj jer 6 = 2 * 3
- *Generšemanje velikog prostog broaj treba da proveri da li je slučajno izabran broj prost*.
  Generišemo slučajan, neparan, veliki broj i proverimo da li je prost (npr. Miler-Rabin test se koristi za proveru da li je broj prost). Veliki prosti brojevi nisu retki, teorema prostih brojeva kaže da se za veliko m šansa da je to prost broj iznosi 1 u ln(m). Za 1024 bitni broj treba probati oko 710 brojeva ($\ln(2)^{1024} = 710$)

3. **Uzajamno prosti brojevi**
   - Dva prirodna broja su uzajamno prosti ako nemaju zajednički delilac veći od 1.
   - Euklidov algoritam nalazi najveći zajednički delilac:
     - U osnovi algoritam za brojeve **e** i **r** nalazi najveći zajednički delilac **g** tako što računa **g = ei + rj,** gde su i & j celi brojevi i jedan od njih može biti negativan. Složenost algoritma je $O(\log_2 e)$

     Primer: 6 = 30i + 18j, i = -1, j= 2
   - Algoritam se primenjuje za nalaženje **e**, tako što proba redom sa prostim brojevima koji su manji od **r**. Ima oko **r/ln(r)** prostih brojeva manjih od **r**, ali r ima najviš $\log_2(r)$ faktora tako da su solidne šanse da se **e** brzo pronađe.
   - Takođe, rezultat euklidovog algoritma se koristi za računanje **d**.

4. **Brzo računanje celobrojnog stepena nekog broja**

- Brzi algoritam računa izraz x^d mod n u $\Theta(\log 2d)$ vremenu upotrebom tehnike ponavljanja stepenovanja:
  - ***za d parno*** $x^d = (x^{d/2})^2$, a
  - ***za d neparno*** $x^d = (x^{d-1/2})^2 * x$

43* Algoritmi za rad sa stringovima (definicije, namene osnovnih algoritama)

- **String** je niz (sekvenca) karaktera – skup karaktera čine slova(velika i mala), cifre, interpkcija, matematički i neki drugi simboli.
- **Niz** (sequence) je lista elemenata gde u kojoj je bitan njihov redosled.
  - Isti elemenat se može ponavljati u nizu.
  - String se može posmatrati kao niz slova.
- **Podniz** (subsequence) nekog niza je novi niz koji se dobija od početnog brisanjem nekih elemenata niza, bez promene relativnog redosleda preostalih elemenata.
  Primer 1:
  {B, C, D, G} - je podniz od niza
  {A, C, B, D, E, G, C, E, D, B, G }
  sa odgovaraju'im niyom indeksa {3, 7, 9, 11}

  Primer 2:
  {A, B, C, D, A, A, D, D, C, B, A, A} ima podnizove
  {A, B, C, D, A, A, D, D, C, B, A, A},{A. A. A}, {A. B. C .D}, {A. C. A. D. B. A} itd., ali {B. B. B} i {A. B. C. C. D} nisu podnizovi.

- **Podrstring** (substring) je deo stringa koji sadrži uzastopne karaktere. Svaki string je ujedno i podniz, ali obrnuto ne važi.
  Primer: {ABCDAADDCBAA} ima podstringove
  - A,BCDA, CBAA, ABCDAA, itd., ali
  - AAA,ACADBA nisu podstringovi

ALGORITMI (primerima je prikazana primena na genetiku)

1. **Najduži zajednički podniz (longest common subsequence LCS)** za data dva stringa pronalazi majduži subsequence koji se nalazi u oba stringa.
   Primer:
   - stringovi CATCGA i GTACC imaju dva LCS: CTCA i TCGA.
   - Algoritam koristi dinamičko programiranje (ideja je da je subsequence podstringa ujedno i subsequence originalnog stringa)
   - Algoritam ima dve faze:
     - Izgrađuje pomoćnu tabelu dimenzije (n+1)(m+1) gde su n i m dužine datih stringova – složenosti Θ(nm)
     - Pomoću tabele nalazi LCS – složenosti O(n+m)
2. **Transformacija jednog stringa u drugi** - potrebno je transformisati string X u string Z upotrebom minimalnog broja operacija.
   - X čine karakteri x_i,i = 1,2,...n
   - Z čine karakteri z_j,j = 1,2,...m

   - Operacije:
   - Copy – kopiranje karaktera : $z_j = x_i$, i = i + 1, j = j + 1
   - Replace – zamena karaktera $z_j = x_i = a$, i++, j++
   - Delete – brisanje karaktera i++ (preskoči karakter u X
   - Insert – ubacivanje karaktera z_j = a, j++ (X se ne dira).

   Primer: ATGATCGGCAT postaje CAATGTGAATC ...

Operation	*X*	*Z*
initial strings	ATGATCGGCAT	
delete A	ATGATCGGCAT	
replace T by C	ATGATCGGCAT	C
replace G by A	ATGATCGGCAT	CA
copy A	ATGATCGGCAT	CAA
copy T	ATGATCGGCAT	CAAT
replace C by G	ATGATCGGCAT	CAATG
replace G by T	ATGATCGGCAT	CAATGT
copy G	ATGATCGGCAT	CAATGTG
replace C by A	ATGATCGGCAT	CAATGTGA
copy A	ATGATCGGCAT	CAATGTGAA
copy T	ATGATCGGCAT	CAATGTGAAT
insert C	ATGATCGGCAT	CAATGTGAATC

- Cena ove transformacije je $5c_k + 5c_z + c_b + c_u$ gde su c_k cena kopiranja, c_z cena zamene, c_b cena brisanja i c_u cena ubacivanja. Problem je za date c_k, c_z, c_b, c_u naći najjeftiniju transformaciju.

3. **<u>Podudaranje stringova (string matching)</u>** – pronalaženje (pattern –a) u datom tekstu.
   - Imamo dva stringa: string teksta T(dužine n) i string paterna P (dužinem, m ≤ n), i želimo da pronađemo sve pojave P u T.

Primer:

- tekst je GTAACAGTAAACG, a patern je AAC

Shift amount	Text and pattern	Shift amount	Text and pattern
0	GTAACAGTAAACG AAC	6	GTAACAGTAAACG AAC
1	GTAACAGTAAACG AAC	7	GTAACAGTAAACG AAC
2	GTAACAGTAAACG AAC	8	GTAACAGTAAACG AAC
3	GTAACAGTAAACG AAC	9	GTAACAGTAAACG AAC
4	GTAACAGTAAACG AAC	10	GTAACAGTAAACG AAC
5	GTAACAGTAAACG AAC		

- Poređenje paterna sa podstringovima koji počinje na poziciji k (shift amount + 1) je primitivan algoritam složenosti O((n-m)m)
- Efikasniji algoritam koji koristi ideju konačnog automata (finite automation) i ima složenost $\Theta(n)$.