

SOFTVERSKI ALGORITMI U SISTEMIMA AUTOMATSKOG UPRAVLJANJA

Skripta, 2022./2023.

A Algoritmi i strukture podataka	4
Šta su algoritmi? Šta se očekuje od računarskog algoritma?	4
Analiza algoritama i asimptotske notacije.	4
Linearna i binarna pretraga.	7
Problem sortiranja (opis, nabrojati algoritme i njihove osobine).	9
Selection sort algoritam.	9
Insertion sort algoritam.	10
Merge sort algoritam.	11
Quicksort algoritam.	14
Elementarne strukture podataka. Red i Stek.	16
Stek	17
Red	18
Liste. Jednostruko i dvostruko spregnute liste.	19
Heap struktura. Osobine i operacije.	22
Reorganizuj-Heap metoda	23
Izgradi-Heap metoda	24
Heapsort algoritam.	25
Red sa prioritetima (Priority Queue).	26
Poređenje kao model računanja (stablo odlučivanja).	28
Algoritmi sortiranja složenosti (n).	28
Counting sort algoritam	28
Radix Sort algoritam	29
Redosledna statistika (opis, min, max, jednovremeni min i max, medijana).	30
Minimum i maksimum	31
Jednovremeno traženje minimuma i maksimuma	31
Rečnik podataka i heširanje (primene, priheš i heš, kolizije, amortizovano vreme izvršavanja).	31
Priheš	32
Hešing	32
Problem kolizije	33
Amortizovano vreme izvršavanja	33
Heširanje ulančavanjem.	34
Dinamičko programiranje. Primer sečenja cevi.	35
Problem sečenja cevi.	36
Dinamičko programiranje. Primer zagrada.	37
Problem matričnog množenja i zagrada.	38
Složenost računanja i klase problema (P, NP, EXP, R, problem odlučivanja, primeri).	40
Problem odlučivanja	41
„Nalik“ slični problemi	41
NP problemi (definicija P i NP problema, vrste NP problema, redukcija).	41

Svođenje ili redukcija problema	42
Algoritam za nalaženje najduže zajedničke podsekvence (definicije sekvene, stringa,..., LCS algoritam).	45
Algoritmi podudaranja stringova (naivni, Rabin-Karp, konačni automat, KMP).	46
Algoritam grube sile	47
Rabin-Karp algoritam	47
Konačni automat	48
KMP algoritam	49
B Grafovi	50
Grafovi (definicija, tipovi, primena, vrste algoritama).	50
Topološko sortiranje grafova. Pretrage grafova.	52
Topološko sortiranje grafova	52
Pretrage grafova	53
Breadth First Search	53
Depth First Search	55
Najkraći put u grafu. Dijkstra algoritam.	57
Najkraći put u grafu. Bellman-Ford algoritam.	59
Minimalno razapinjuće stablo. Kruskalov algoritam.	60
Minimalno razapinjuće stablo. Primov algoritam.	62
Algoritam svih najkraćih putanja.	63
Floyd-Warshall algoritam.	64
C Mašinsko učenje	65
Mašinsko učenje. Definicija i osnovna podela.	65
Merenje rastojanja između podataka.	66
Skaliranje podataka.	68
Skaliranje svojstava - normalizacija	68
K-means klasterizacija. Izbor optimalnog k.	69
Linearna regresija.	72
Vrste gradijentnog algoritma.	73
Lokalno ponderisana linearna regresija.	74
Logistička regresija. Postupak određivanja parametara.	75
K najbližih suseda. Izbor optimalnog k.	77
Izbor modela. Unakrsna validacija. Vrste.	79
Unakrsna validacija (Cross Validation)	81
Jednostavna unakrsna validacija	82
K-tostruka validacija	82
Validacija jednostrukе eliminacije	83
Izbor svojstava. Algoritmi.	83
D Lanci blokova i kriptografija	84
Kriptografija (primena, procesi, šifrovanje i bezbednosne pretnje).	84
Osnovni algoritmi kriptografije (podela, primitivni i napredni algoritmi).	85

Primitivni algoritmi šifrovanja	85
Simetrična kriptografija	86
Asimetrična kriptografija	86
RSA algoritam (principi, uključeni algoritmi).	87
Algoritam za digitalne potpise. DSA algoritam.	88
Lanac blokova (blockchain).	90
Problem vizantijskih generala.	92

A Algoritmi i strukture podataka

Šta su algoritmi? Šta se očekuje od računarskog algoritma?

Algoritmi su višekoračni postupci za izvršavanje nekog zadatka. Definišu se još kao matematička apstrakcija računskog programa, računarska procedura za rješavanje nekog problema. Algoritmi se dosta oslanjaju na **strukture podataka**, te dobro osmišljen algoritam može značajno smanjiti potrebnu računarsku snagu za rješavanje problema.

Opis algoritma mora biti **veoma precizan**, na osnovu zadatih ulaza da proizvede **korektno rješenje**, da bude **efikasan** (u kontekstu vremena izvršavanja), **funkcionalan, robustan, jednostavan**.

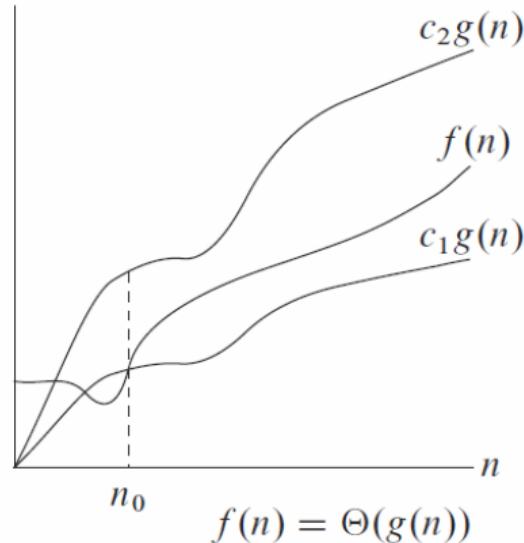
Analiza algoritama i asimptotske notacije.

Vremenska analiza algoritama predstavlja približno predviđanje vremena izvršavanja algoritma u zavisnosti od veličine ulaza n (i to za veliko n). Takođe, pored ulaza, zavisna je od arhitekture na kojoj se izvršava i od jezika u kojem je implementirana, ali se generalno ti uticaji ne uzimaju u obzir.

Koristi se nekoliko asimptotskih notacija

1. $\Theta(g(n))$ **notacija**, ograničava sa obje strane, označava skup funkcija za koje važi

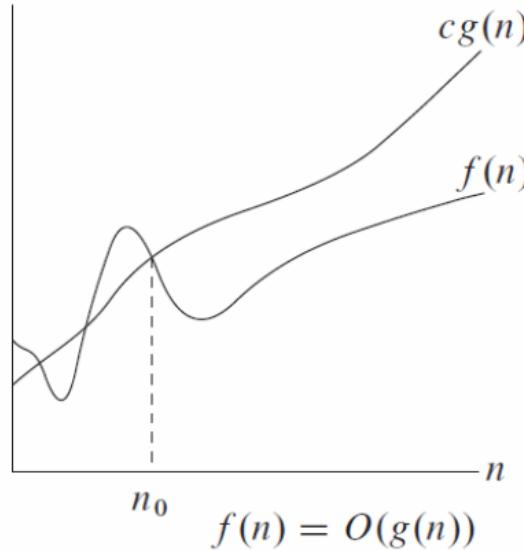
$$\Theta(g(n)) = \{f(n): \forall n \geq n_0, \exists c_1, c_2 > 0 \Rightarrow 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$



Slika 1. $\Theta(f(n))$ složenost

2. **$O(g(n))$ notacija, “najgori slučaj”, ograničava sa gornje strane, označava skup funkcija za koje važi**

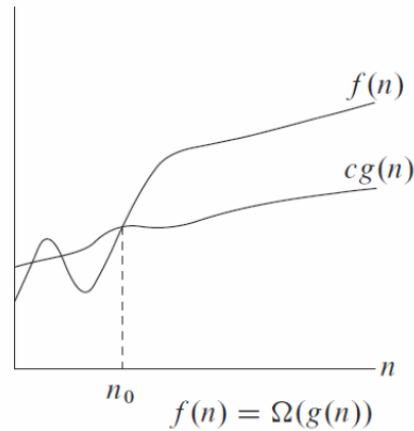
$$O(g(n)) = \{f(n) : \forall n \geq n_0, \exists c > 0 \Rightarrow 0 \leq f(n) \leq cg(n)\}$$



Slika 2. $O(f(n))$ složenost

3. **$\Omega(g(n))$ notacija, “najbolji slučaj”, ograničava sa donje strane, označava skup funkcija za koje važi**

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



Slika 3. $\Omega(f(n))$ složenost

Kompleksnost algoritma se određuje na sledeće načine:

1. Za svaku naredbu odrediti koliko će se puta izvršiti i sabrati dobijene vrijednosti.
2. Zadržati samo najveći sabirak u dobijenom izrazu (jer pričamo o jako velikom n).
3. Ignorišemo konstante jer ne utiču na formiranje složenosti.

Neke klase algoritama sa primjerima:

1. **$O(1)$ algoritmi**
 - a. Vrijeme algoritma ne zavisi od veličine ulaza.
 - b. Ne mora nužno značiti izvršavanje u jako kratkom vremenu.
 - c. *Jednostavne operacije - print, if, return,...*
2. **$O(\log_2 n)$ algoritmi**
 - a. *Algoritam binarne pretrage* - skraćuje niz za polovinu u svakoj iteraciji dok ne pronađe elemenat.
3. **$O(n)$ algoritmi**
 - a. *Algoritam linearne pretrage* - prolazak kroz niz i postavljanje pitanja da li je taj elemenat koji tražimo elemenat na kojem smo trenutno.
 - b. *Bucket sort* algoritam.
4. **$O(n \log_2 n)$ algoritmi**
 - a. *Divide and conquer* algoritmi - *Quicksort, Merge sort*
 - b. *Heapsort*.
5. **$O(n^2)$ algoritmi**
 - a. Ugnježdene petlje.
 - b. *Bubble sort, Selection sort, Insertion sort,...*
6. **$O(2^n)$ algoritmi**
 - a. Pronalaženje Fibonačijevih brojeva.
 - b. Algoritmi rekurzivnih poziva - eksponencijalna složenost.
7. **$O(n!)$ algoritmi**
 - a. *Brute force* algoritmi, generisanje permutacije niza,...

Linearna i binarna pretraga.

Linearna pretraga - naziva se i **sekvencijalna pretraga** jer iterira kroz niz redom dok ne pronađe traženi elemenat. Složenost algoritma je $O(n)$.

Postoje dvije verzije:

1. *Bez stražara.*

```
Linearno-Pretraži(A, key):
    for k = 1 to A.length:
        if A[k] == key:
            return k
    return nije-nađen
```

2. *Sa stražarem* - ideja ubacivanja stražara je da se dio koda koji se ponavlja oslobođi porebe za provjerom kraja niza

```
Linearno-Pretraži-Sa-Stražarem(A, key):
    n = A.length
    last = A[n]
    A[n] = key
    k = 1
    while A[k] != key:
        k += 1
    A[n] = last
    if k < n or A[n] == key:
        return k
    else:
        return nije-nađen
```

Binarna pretraga - ulaz u navedeni algoritam je **predsortiran** niz, dok je princip rada:

- Ako je vrijednost traženog ključa jednaka ključu na sredini niza, pretraga je gotova.
- Ako je vrijednost traženog ključa manja od ključa na sredini niza, pretražuje se lijeva polovina niza.
- Ako je vrijednost traženog ključa veća od ključa na sredini niza, pretražuje se desna polovina niza.

Niz se cijepa na polovine dok se ne dođe do odgovarajućeg elementa. Pogodan za **rekurzivnu izvedbu**. Složenost algoritma je $O(\log n)$.

1. Nerekurzivna binarna pretraga

```
Binarna-Pretraga(A, key):
    p = 1
    r = A.length

    while p <= r:
        q = floor((p+r)/2)
        if A[q] == key:
            return q
        else if A[q] > key:
            r = q - 1
        else:
            p = q + 1
    return nije-nađen
```

2. Rekursivna binarna pretraga

```
Binarna-Pretraga(A, key):
    Rekursivna-Binarna-Pretraga(A, 1, A.length, key)

Rekursivna-Binarna-Pretraga(A, p, r, key):
    if p > r:
        return nije-nađen
    else:
        q = floor((p+r)/2)
        if A[q] == key:
            return q
        else if A[q] > key:
            Rekursivna-Binarna-Pretraga(A, p, q-1, key)
        else:
            Rekursivna-Binarna-Pretraga(A, q+1, r, key)
```

Problem sortiranja (opis, nabrojati algoritme i njihove osobine).

Problem sortiranja - kao ulaz posmatra se **sekvenca brojeva**, dok bi izlaz trebalo da bude **permutacija** tih brojeva tako da je svaki naredni veći (ili manji, zavisno od vrste sortiranja) od prethodnih brojeva. Pitanje (problem) je **kako naći takvu permutaciju**. Osobine sortirajućih algoritama su

1. **Da li rade u mjestu ili ne** - da li zahtjevaju dodatni memorijski prostor (odnosno, da li postoje operacije stvaranja nekog novog pomoćnog niza).
2. **Da li su stabilni** - da li isti ključevi zadržavaju isti poredak prije i poslije sortiranja (recimo, ključevi 3_a i 3_b su isti, ali nose različite **satelitske podatke** (a i b) - ako je algoritam stabilan, oni će ostati u istom poretku kao na ulazu).

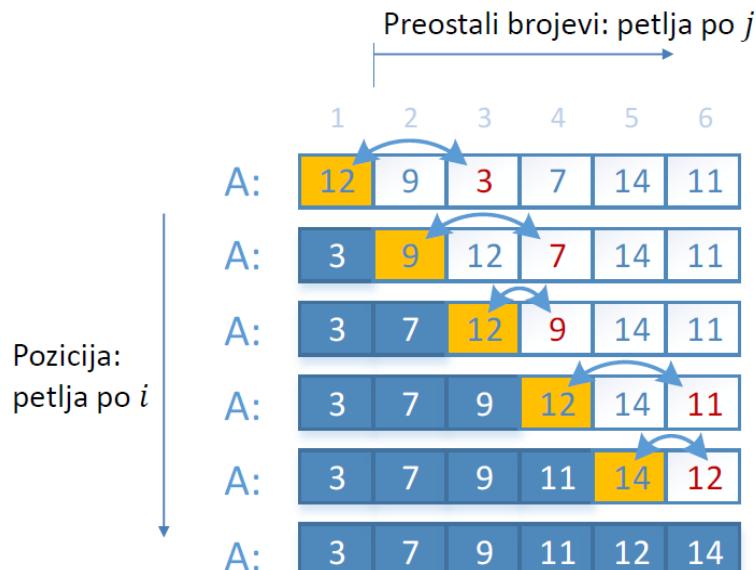
Neki od **algoritama sortiranja** su *Selection sort*, *Insertion sort*, *Merge sort*, *Quicksort*, *Bucket sort*,...

Selection sort algoritam.

Naziva se **sortiranje izborom** (**Selection sort**) jer se za svaku poziciju u nizu bira najmanji elemenat od te pozicije pa do ostatka niza. Kada se pronađe takav elemenat, vrši se zamjena.

Lijevo od pozicije biće sortirani brojevi. Za poslednji elemenat u nizu algoritam nema šta da radi jer je on najveći (najmanji).

Složenost algoritma je $O(n^2)$. **Stabilan algoritam koji radi u mjestu**.



Slika 4. Selection sort algoritam

```

Sortiraj-Izborom(A):
    for i = 1 to A.length-1:
        indmin = i
        for j = i + 1 to A.length:
            if A[j] < A[indmin]:
                indmin = j
            swap(A[i], A[indmin])

```

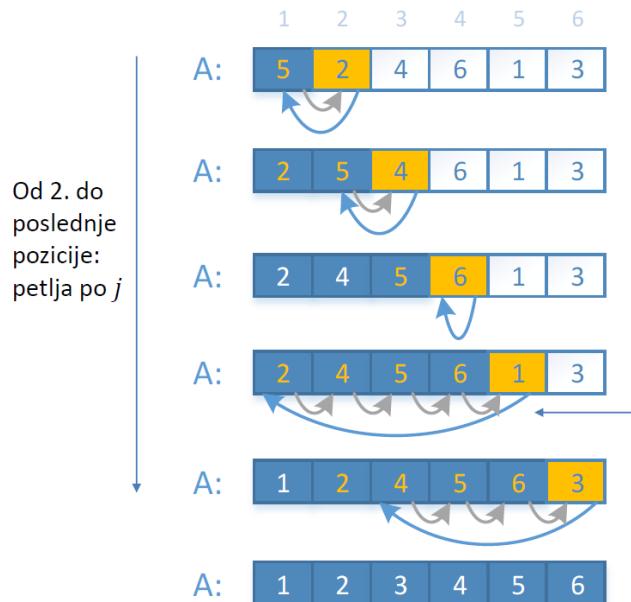
Insertion sort algoritam.

Ideja algoritma slična je postupku ređanja karata u ruci:

- Kako nove karte "pristižu", iterativno se prolazi kroz sve karte unazad dok novoj ne nađemo "mjesto".
- Nova karta predstavlja svaki sledeći elemenat od naše trenutne pozicije.

U najboljem slučaju, niz je već sortiran pa ćemo imati 0 pomjeranja te složenost $\Omega(n)$.

Najgori slučaj će biti niz sortiran u obrnutom redoslijedu, te ćemo imati složenost $O(n^2)$. Karakteriše ga **stabilnost i rad u mjestu**.



Slika 5. Insertion sort algoritam

```

Sortiraj-Umetanjem(A):
    for j = 2 to A.length:
        key = A[j]
        i = j - 1
        while i > 0 and A[i] > key:
            A[i+1] = A[i]
            i -= 1
        A[i+1] = key

```

Merge sort algoritam.

Merge sort algoritam koristi algoritamsku paradigmu “*podijeli i osvoji*”.

Paradigma koja dijeli zadatak na manje koji su slični originalnom zadatku. Manji zadaci se tako rješavaju rekurzivom i njihova rješenja se sklapaju da bi se dobilo finalno rješenje.

Koraci su:

- **Podijeli** - podijeliti zadatak na manje.
- **Osvoji** - manji zadaci se rješavaju rekurzivno. Kada je zadatak dovoljno mali, rješavanje je trivijalno i naziva se **base case**.
- **Kombinuj** - rješenja se kombinuju da bi se dobilo rješenje originalnog zadatka.

```

Sortiraj-Objedinjavanjem(A):
    Sortiraj-Objedinjavanjem-Korak(A, 1, A.length)

Sortiraj-Objedinjavanjem-Korak(A, p, r):
    if p < r:
        q = floor((p+r)/2)
        Sortiraj-Objedinjavanjem-Korak(A, p, q)
        Sortiraj-Objedinjavanjem-Korak(A, q+1, r)
        Objedini(A, p, q, r)

```

Niz se cijepa na polovine dok se ne dođe do baznog slučaja (niz dužine 1). Nakon baznog slučaja, niz se objedinjuje (metoda koja ga i sortira).

```

Objedini(A, p, q, r):
    br_el_lijevo = q - p + 1
    br_el_desno = r - q

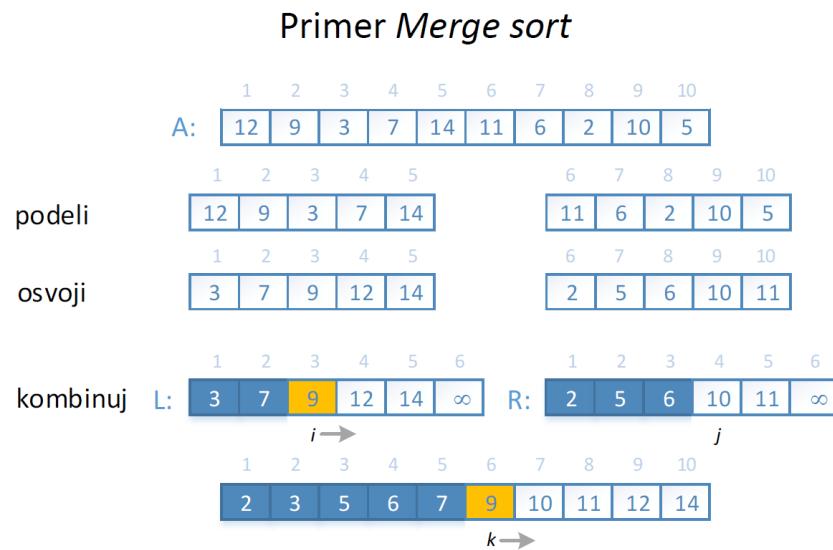
    for i = 1 to br_el_lijevo:
        Lijevi_Podniz = A[p+i-1]
    for j = 1 to br_el_desno:
        Desni_Podniz = A[q+j]

    Lijevi_Podniz[br_el_lijevo + 1] = Inf
    Desni_Podniz[br_el_desno + 1] = Inf

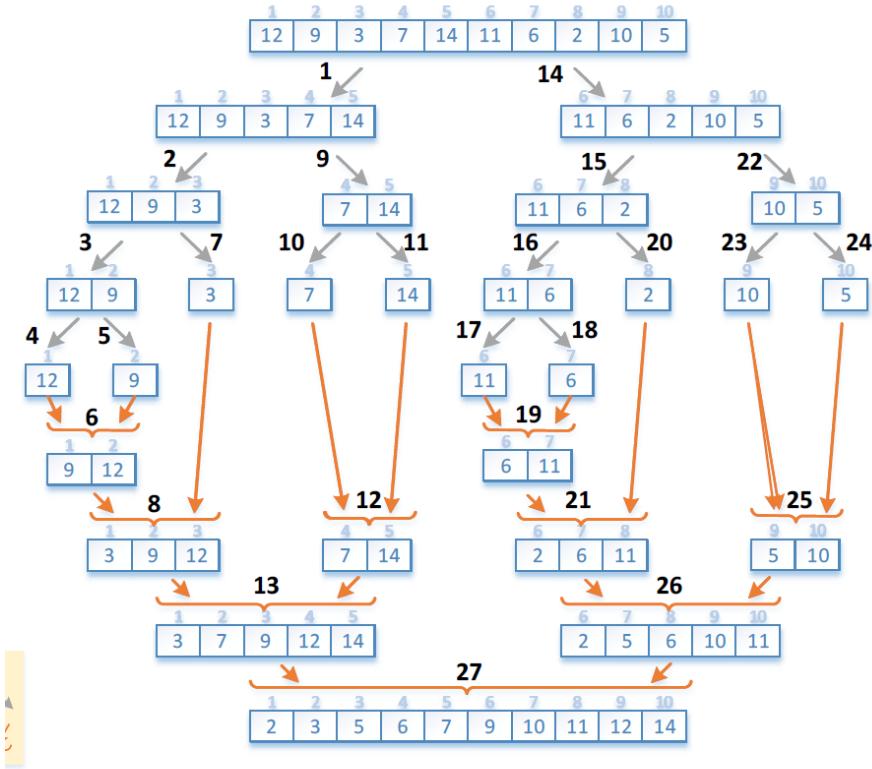
    i = 1
    j = 1

    for k = p to r:
        if Lijevi_Podniz[i] <= Desni_Podniz[j]:
            A[k] = Lijevi_Podniz[i]
            i += 1
        else:
            A[k] = Desni_Podniz[j]
            j += 1

```



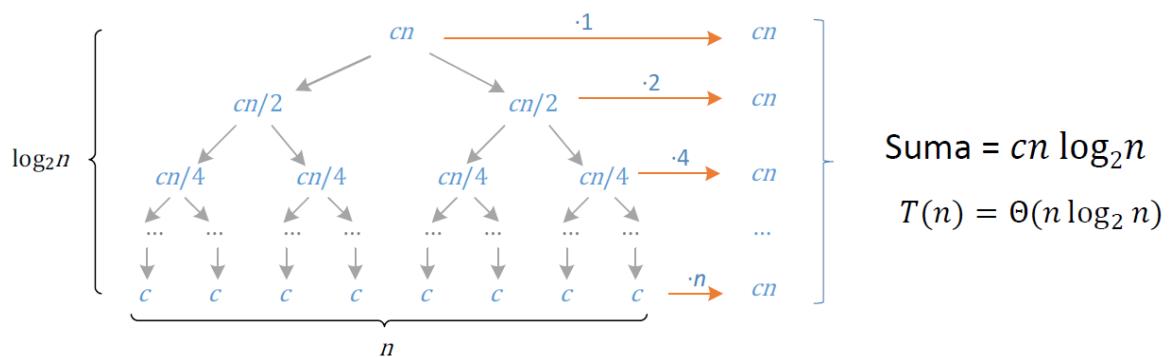
Slika 6. Primjer Merge sort algoritma



Slika 7. Vizualizacija Merge sort algoritma

Vrijeme izvršavanja algoritma je $O(n \log n)$ što je znatno brže od $O(n^2)$ složenosti. Međutim, sporiji je za malo n zbog konstantnog faktora koji je veći nego kod drugih algoritama.

Karakteristika algoritma jeste da **ne radi u mjestu**.



Slika 8. Složenost Merge sort algoritma

Quicksort algoritam.

Kao i *Merge sort*, primjenjuje programsku paradigmu “*podijeli i osvoji*”.

Radi na principu biranja **pivot** elementa, gdje ređa brojeve lijevo i desno od **pivota** ako su veći ili manji od njega, respektivno. Kada poređa, primjenjuje isti princip na dobijene podnizove brojeva.

Karakteristika je da **radi u mjestu**.

Praktično je često u upotrebi - konstantan faktor u asymptotskoj notaciji manji od faktora *Merge sort* algoritma.

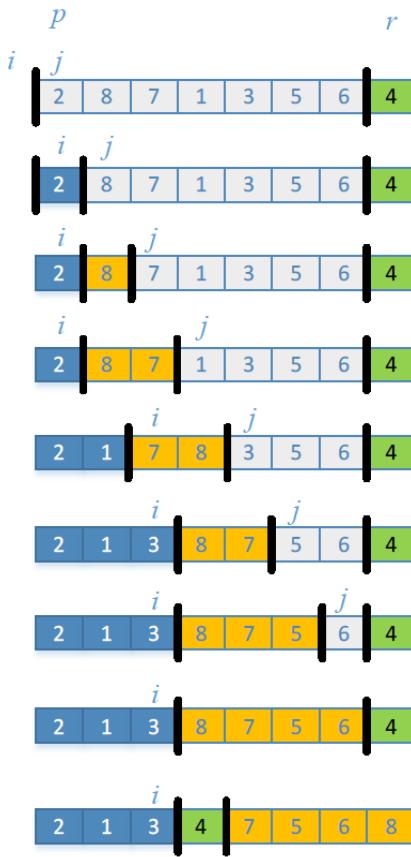
```
Sortiraj-Razdvajanjem(A):
    Sortiraj-Razdvajanjem-Korak(A, 1, A.length)

Sortiraj-Razdvajanjem-Korak(A, p, r):
    if p < r:
        q = Podijeli(A, p, r)
        Sortiraj-Razdvajanjem-Korak(A, p, q-1)
        Sortiraj-Razdvajanjem-Korak(A, q+1, r)
```

Podijeli() je metoda koja radi na principu biranja **pivot** elementa. **Pivot** se stavlja na poslednje mjesto u (pod)nizu i elementi se ređaju ili lijevo ili desno, u zavisnosti od toga da li su veći ili manji od **pivota**. Kada se poredaju svi elementi, **pivot** se postavlja između ta dva podniza.

```
Podijeli(A, p, r):
    pivot = A[r]
    i = p - 1
    for j = p to r-1:
        if A[j] <= pivot:
            i += 1
            swap(A[i], A[j])
    swap(A[i+1], A[r])
    return i+1
```

Ako je podjela balansirana (odnosno, ako postoji približno jednako manjih i većih elemenata od pivota), **Quicksort** ima složenost $\Theta(n \log n)$. Ako to nije slučaj, algoritam ima složenost $\Theta(n^2)$.



Slika 9. Quicksort algoritam

Elementarne strukture podataka. Red i Stek.

Postoji nekoliko vrsta elementarnih struktura podataka:

- **Stek (Stack)**
- **Red (Queue)**
- **Povezane liste (Lists)**
- **Stabla (Trees)**

Stek i red su dinamički skupovi gdje se elementi uklanjuju unaprijed određenim redoslijedom:

- Kod **steka** se briše poslednji dodat elemenat (poznato kao **LIFO - Last In First Out** princip).
- Kod **reda** se briše prvi dodat elemenat (poznato kao **FIFO - First In First Out** princip).

Stek

Struktura koja podsjeća na slaganje tanjira - *slažemo tanjur na tanjur i prvo "skidamo" onaj sa vrha, nikad one između.*

Operacije vezane za **stek**:

- Dodavanje elemenata na **stek** - **push**.
- Skidanje elemenata sa **steka** - **pop**.
- Provjera da li ima elemenata?

Stek se može implementirati (predstaviti) **listom** ili **nizom** $S[1, \dots, S.vrh]$ koji sadrži elemente:

- $S[1]$ je elemenat na dnu, $S[S.vrh]$ je elemenat na vrhu **steka**.
- $S.vrh + 1$ je indeks **prvog praznog mesta** na **steku**. Ako je $S.vrh == 0$, **stek je prazan**.
- Ako se pozove **pop** operacija na prazan **stek**, dolazi do **underflow** greške.
- Ako se pozove **push** operacija na **stek** na kojem više nema mesta (zbog memorijskog ograničenja), dolazi do **overflow** greške.

Stek primer

Primer par operacija sa stekom:

(početno stanje a))

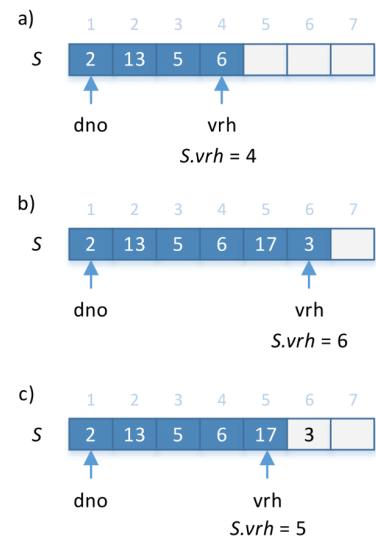
GURNI(S, 17)

GURNI(S, 3)

(nastaje stanje b))

a = POVUCI(S)

(nastaje stanje c))



Slika 10. Primjer steka

Operacije **steka** su **brze** - složenosti $O(1)$.

Red

Struktura koja podsjeća na red u čekaonici - *onaj ko je prvi došao, njegov zahtjev se prvo "obrađuje"*.

Operacije vezane za **red**:

- Dodavanje elemenata - **enqueue**.
- Preuzimanje ili uklanjanje elemenata - **dequeue**.
- Provjera da li ima elemenata?

Redovi koje posmatramo su **single ended** - elemenat se može dodavati samo sa jednog kraja. Postoje i **double ended queue (deque)** strukture, gdje se elementi na **red** mogu dodavati sa oba kraja.

Podaci (**indeksi**) vezani za **red**:

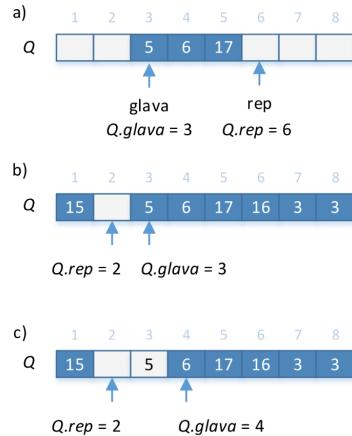
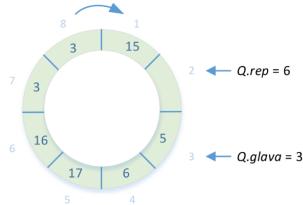
- **Glava (head)** - pokazuje na prvi elemenat **reda**. Preuzima se elemenat sa početka - **glave reda**.
- **Rep (tail)** - pokazuje na poslednji elemenat reda. Dodavanje elemenata se vrši na kraj reda - **rep reda**.

Red se može implementirati **nizom**, **listom** ili kao **kružni buffer**. Bitno je da red ima fiksani broj elemenata - **kapacitet reda**.

Primer reda

- Primer par operacija sa redom:

```
DODAJ(Q,16)
DODAJ(Q,3)
DODAJ(Q,3)
DODAJ(Q,15)
a = PREUZMI(Q)      // a = 5
```



Slika 11. Primjer reda

Operacije **reda** su konstantne složenosti - $O(1)$.

Liste. Jednostruko i dvostruko spregnute liste.

Struktura podataka **ulančavanja** - jedan elemenat uvijek pokazuje na sledećeg (susjednog). Koriste se u slučajevima kada pored **ključa** postoje **satelitski podaci** (dodatni podaci o elementu). Za razliku od nizova gdje je poređak uređen preko indeksa, kod lista imamo **pokazivače** na susjedne elemente (što znači da logička uređenost (uređenost u **listi**) ne mora nužno značiti fizičku uređenost (uređenost u memoriji)).

Pogodna struktura za česta posjećivanja, dodavanja i brisanja elemenata.

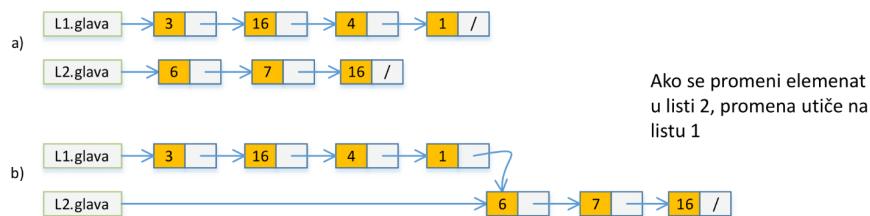
Razlikuju se:

- **Jednostruko spregnute liste** - elemenat pokazuje na naredni elemenat.
Omogućeno je kretanje samo u **jednu stranu**.

Jednostruko spregnuta lista

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

Primer nadovezivanja listi:



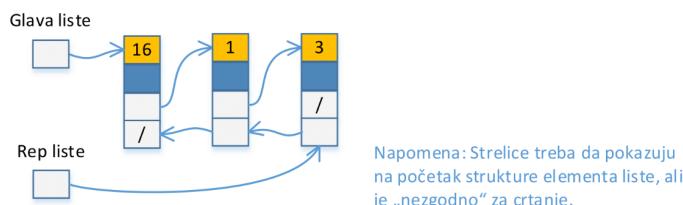
Slika 12. Jednostruko spregnuta lista

- **Dvostruko spregnute liste** - dva pokazivača - elemenat pokazuje i na naredni i na prethodni elemenat. Omogućeno je kretanje u **obje strane**.

Dvostruko spregnuta lista

- Svaki elemenat sadrži
 - *naredni* – pokazivač na naredni elemenat (*next*) (== Nil kada je poslednji)
 - *prethodni* – pokazivač na prethodni elemenat (*prev*) (== Nil kada je prvi)
 - *ključ*, prisutan kod sortiranih lista (*key*)
 - drugi – satelitski podaci
- Pokazivač na prvi elemenat – *L.glava (.head)*
 - Ako je *L. glava == NIL* lista je prazna
- Pokazivač na poslednji elemenat – *L.rep (.tail)* – nije uvek prisutan

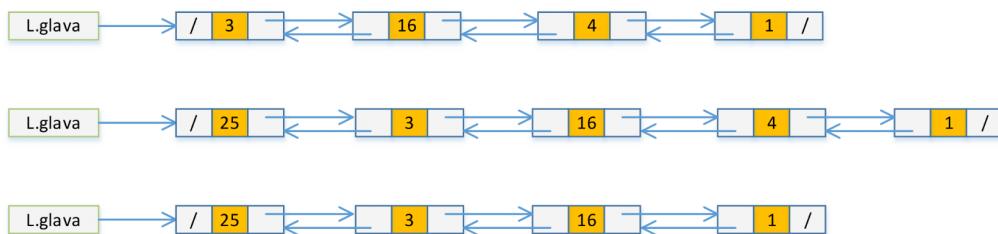
Element liste:



Slika 13. Dvostruko spregnuta lista

Primer rada sa listom

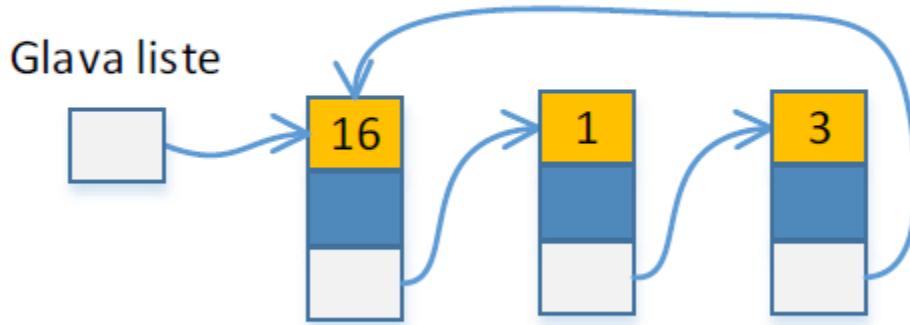
- a) Lista sadrži 4 elementa sa ključevima {3, 16, 4, 1}
- b) Dodat je novi elemenat sa ključem 25
Primetiti: dodan je na početak liste
- c) Pretraživanjem liste je dobijen pokazivač na elemenat sa ključem 4 i zatim je obrisan.



Slika 14. Dvostruko spregnuta lista

- **Cirkularne liste** - poslednji element pokazuje na prvi. Pogodne za implementaciju **kružnog buffera**.

Primer cirkularne (kružne) liste



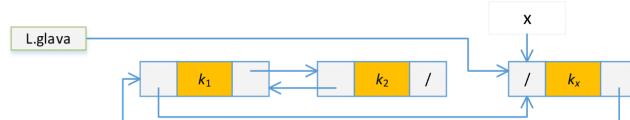
Slika 15. Cirkularne liste

Operacije sa listama:

- **Pretraži listu** - u najgorem slučaju traženi elemenat je na kraju liste (ili ga uopšte nema u listi), te tako dobijamo složenost operacije od $O(n)$.
- **Dodavanje elementa u listu**

Dodavanje elementa u listu

```
DODAJ-U-LISTU(L,x)
1 x.naredni = L.glava
2 if L.glava ≠ Nil
3   L.glava.prethodni = x
4   L.glava = x
5 x.pretodni = Nil
```

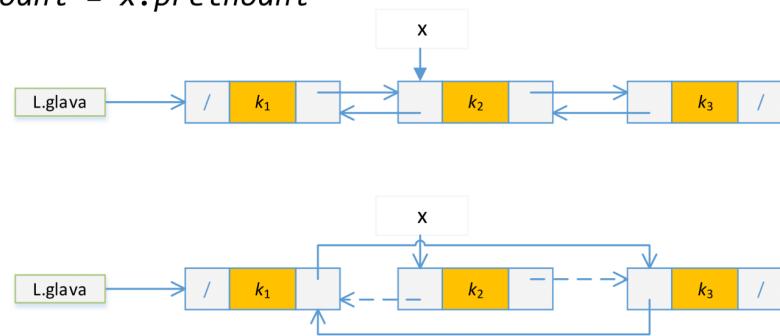


Slika 16. Dodavanje elementa u listu

- **Brisanje elementa iz liste**

Brisanje elementa iz liste

```
BRIŠI-IZ-LISTE(L,x)
1  if x.prethodni ≠ Nil
2      x.prethodni.naredni = x.naredni
3  else L.glava = x.naredni
4  if x.naredni ≠ Nil
5      x.naredni.prethodni = x.prethodni
```



Slika 17. Brisanje elementa iz liste

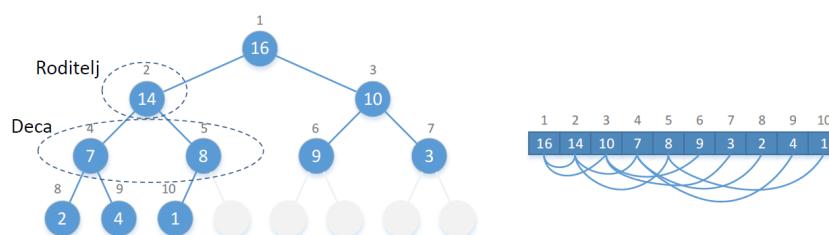
Heap struktura. Osobine i operacije.

Heap je struktura koja se logički predstavlja kao binarno stablo (skoro kompletno, što znači da je moguće da je poslednji nivo nepotpunjen do kraja).

Pogodna je za realizaciju **reda sa prioritetima**.

Postoje dvije vrste **heap struktura**:

1. **Max-heap**, koji ima osobinu da je svaki **roditelj** veći od svoje **djece**.
2. **Min-heap**, koji ima osobinu da je svaki **roditelj** manji od svoje **djece**.



Slika 18. Prikaz dobijanja heap strukture od niza

Postoji nekoliko osobina **heap strukture** i operacija koje se mogu izvesti nad **heap**-om:

- **Roditelj** određenog elementa u nizu dobijamo kao $i/2$, gdje je i indeks tog elementa u nizu.
- **Dubina (visina) heap strukture** je $\log n$.
- **Lijevo dijete** elementa dobijamo kao $2i$, dok **desno dijete** kao $2i+1$.
- **Reorganizuj-Heap (Max-Heapify / Min-Heapify)** - održava **min/max heap** osobinu (složenosti $O(\log n)$).
- **Izgradi-Heap (Build-Max-Heap / Build-Min-Heap)** - pravi (izgrađuje) **min/max heap** na osnovu nesortiranog niza (složenost $O(n)$).
- **Sortiraj-Heap-om (Heapsort)** - sortira niz u mjestu (složenost $O(n \log n)$).

U narednim primjerima koristi se max-heap struktura.

Reorganizuj-Heap metoda

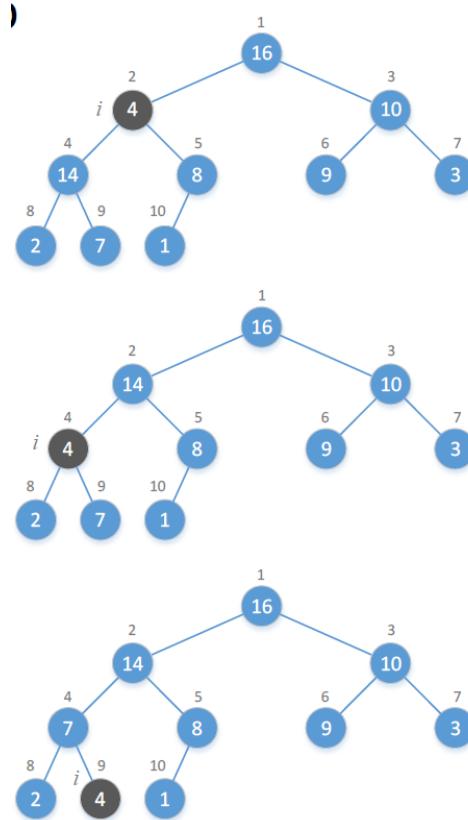
Metoda koja se primjenjuje radi izrade **heap** strukture.

Radi na principu da ako u **korjenu (roditelju)** podstabla postoji vrijednost manja nego u **djeci**, prosljeđujemo tu vrijednost ka "dole" kako bi se održala **max-heap** osobina.

```
Reorganizuj-Heap(A, i):
    L = Lijevo-Dijete(i)
    D = Desno-Dijete(i)

    if L <= A.veličina_heap and A[L] > A[i]:
        najveći = L
    else:
        najveći = i

    if D <= A.veličina_heap and A[D] > A[najveći]:
        najveći = D
    if najveći != i:
        swap(A[najveći], A[i])
        Reorganizuj-Heap(A, najveći)
```



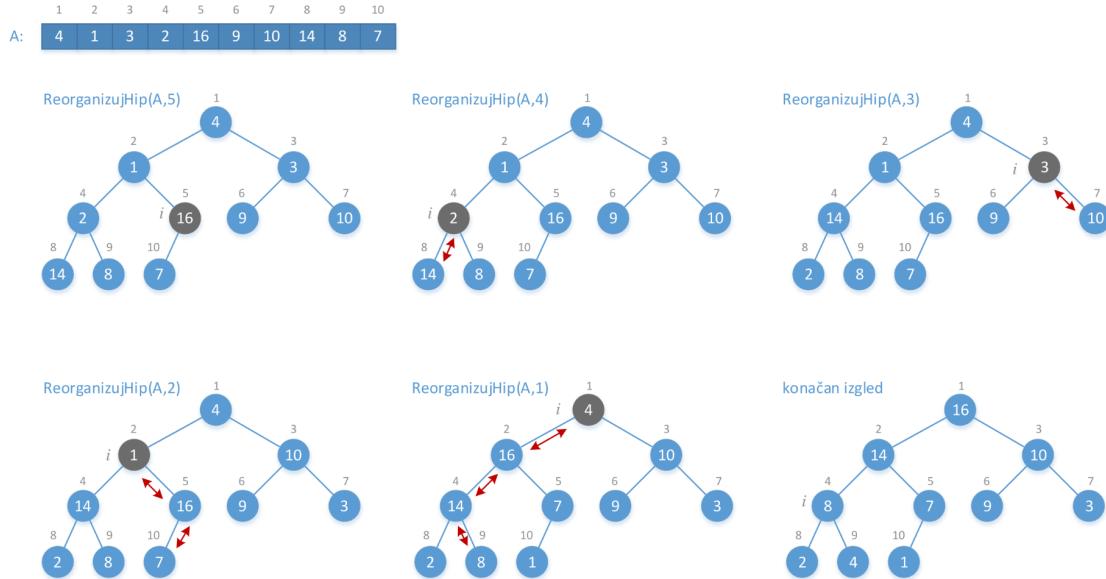
Slika 19. Reorganizuj-Heap metoda

Izgradi-Heap metoda

Metoda koja koristi *Reorganizuj-Heap* metodu tako što je primjenjuje (unazad) na svim elementima koji su **roditelj**, gdje se poslednji roditelj (odnosno roditelj od kojeg se počinje) nalazi na indeksu polovine niza.

```
Izgradi-Heap(A):
    A.veličina_heap = A.length
    for i = floor(A.length / 2) downto 1:
        Reorganizuj-Heap(A, i)
```

Primer izgradnje hipa



Slika 20. Primjer izgradnje heap strukture

Heapsort algoritam.

Nakon izgradnje **heap-a**, najveći elemenat niza će biti u korjenu. On se “izbacuje” (mijenja mjesto sa posljednjim elementom u nizu), te se poziva metoda

Reorganizuj-Heap na korjen koja će opet izbaciti najveći elemenat niza u korjen.

Brzina algoritma je odlična (složenosti $O(n \log n)$), ali od njega je bolji dobro implementiran *Quicksort* algoritam.

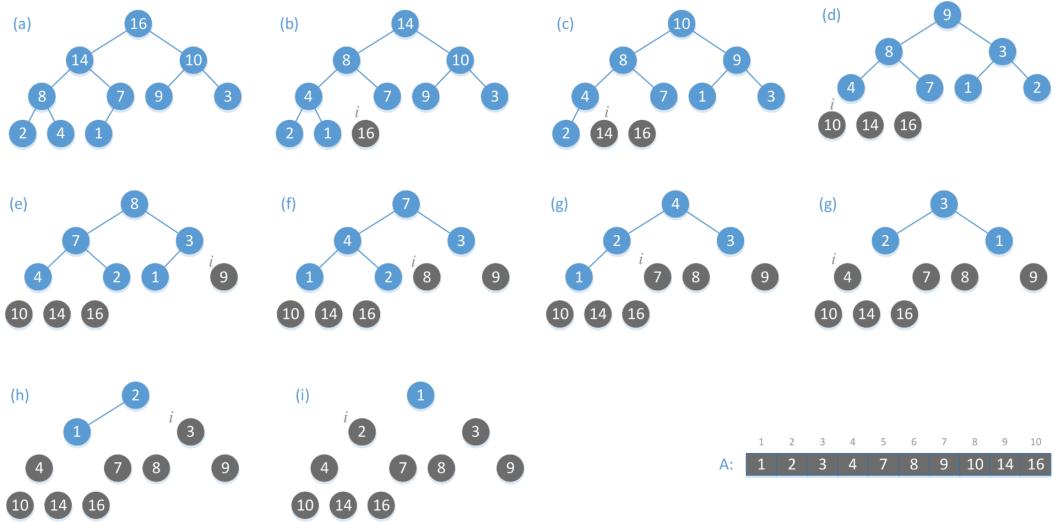
Karakteristika algoritma je **što radi u mjestu**, jer se promjene mesta elementima odvijaju u istom fizičkom nizu.

```

Heapsort(A):
    Izgradi-Heap(A)
    for i = A.length downto 2:
        swap(A[1], A[i])
        A.veličina_heap = A.veličina_heap - 1
        Reorganizuj-Heap(A, 1)

```

Primer sortiranja hipom



Slika 21. Heapsort algoritam

Red sa prioritetima (Priority Queue).

Struktura podataka koja organizuje skup podataka S gdje svaki elemenat ima udružen prioritet - "ključ" naziva se **red sa prioritetima**.

Redovi sa prioritetima podržavaju operacije:

- *Maksimum (Maximum)* - vraća elemenat iz S sa najvećim ključem.

```
Maksimum(A):
```

```
    return A[1]
```

- *Izdvoj-Maksimum (Extract-Max)* - uklanja i vraća elemenat iz S sa najvećim ključem.

```
Izdvoj-Maksimum(A):
```

```
    if A.veličina_heap < 1:
        throw error
    max = A[1]
    A[1] = A[A.veličina_heap]
    A.veličina_heap = A.veličina_heap - 1
    Reorganizuj-Heap(A, 1)
    return max
```

- *Povećaj-Ključ (Increase-Key)* - povećava vrijednost (ključa) elementa x na k (pod uslovom da je k veće od tekućeg ključa).

```
Povećaj-Ključ(A, i, ključ):
```

```
    if ključ < A[i]:
        throw error
    A[i] = ključ
    while i > 1 and A[Roditelj(i)] < A[i]:
        swap(A[Roditelj(i)], A[i])
        i = Roditelj(i)
```

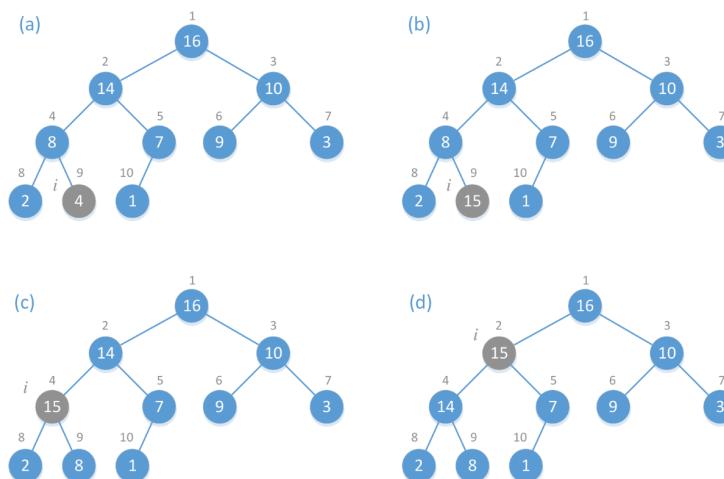
- *Dodaj (Insert)* - dodaje elemenat x u skup S

```
Dodaj(A, ključ):
```

```
    A.veličina_heap = A.veličina_heap + 1
    A[A.veličina_heap] = -Inf
    Povećaj-Ključ(A, A.veličina_heap, ključ)
```

Primer Priority Queue

- Prioritet zadatka #9 ($i=9$) je promjenjen sa 4 na 15 ...



Slika 22. Primjer promjene prioriteta

Poređenje kao model računanja (stablo odlučivanja).

Poređenje kao model računanja jeste model koji se oslanja na poređenje i koji se može primjeniti na proizvoljan (apstraktan) tip podataka, gdje se trajanje algoritma iskazuje **brojem operacija poređenja**.

Svi gorepomenuti sortirajući algoritmi su **algoritmi poređenja**, jer porede vrijednosti u nizu kako bi izvršavali dalje korake.

Svi algoritmi poređenja se mogu prikazati **stablom odlučivanja - potpuno binarno stablo** (svaki čvor u stablu ili je dijete ili ima oba djeteta). Njime se prikazuju svi mogući ishodi koji se mogu desiti pri sortiranju.

Stablo odluke, za n elemenata u nizu, posjeduje bar $n!$ listova (jer toliko postoji permutacija niza, a listovi označavaju pojedinačne permutacije koje nisu unikatne, odnosno može se jedna permutacija pojaviti više puta, zato "bar $n!$ ").

Kako binarno stablo ima **najviše** 2^h listova (gdje je h visina stabla), važiće da je

$$n! \leq 2^h, \text{ odnosno } h \geq \log(n!) = \Omega(n \log n).$$

Zaključak je da je **sortiranje u modelu poređenja složenosti** $\Omega(n \log n)$. Slično se dobija da je **problem pretrage u modelu poređenja složenosti** $\Omega(\log n)$.

Algoritmi sortiranja složenosti $\Theta(n)$.

Algoritmi koji imaju **linearno vrijeme trajanja** - razlog zašto su brži od prethodnih algoritama (i teorijske granice $\Omega(n \log n)$) jeste što uzimaju u obzir **dodatne informacije**, a ne **samo oslanjanje na poređenje vrijednosti**.

Counting sort algoritam

Algoritam sortiranja prebrojavanjem radi nad prirodnim brojevima.

Radi na principu brojanja koliko se puta pojavio elemenat u nizu i tako ga smješta u niz (recimo da, ako postoji 17 manjih brojeva od broja m , tada se broj m nalazi na 18. mjestu).

Karakteristika je da **ne radi u mjestu**, ali je **stabilan**.

Sastoji se od tri koraka:

- **Prebrojavanje pojava** iste vrijednosti u nizu i vođenje "evidencije" o prebrojavanju u nizu C .
- **Pravljenje kumulativne sume pojava** da bi se dobile pozicije gdje će se naći poslednje pojave iste vrijednosti.
- **Raspoređivanje** vrijednosti u novi niz B .

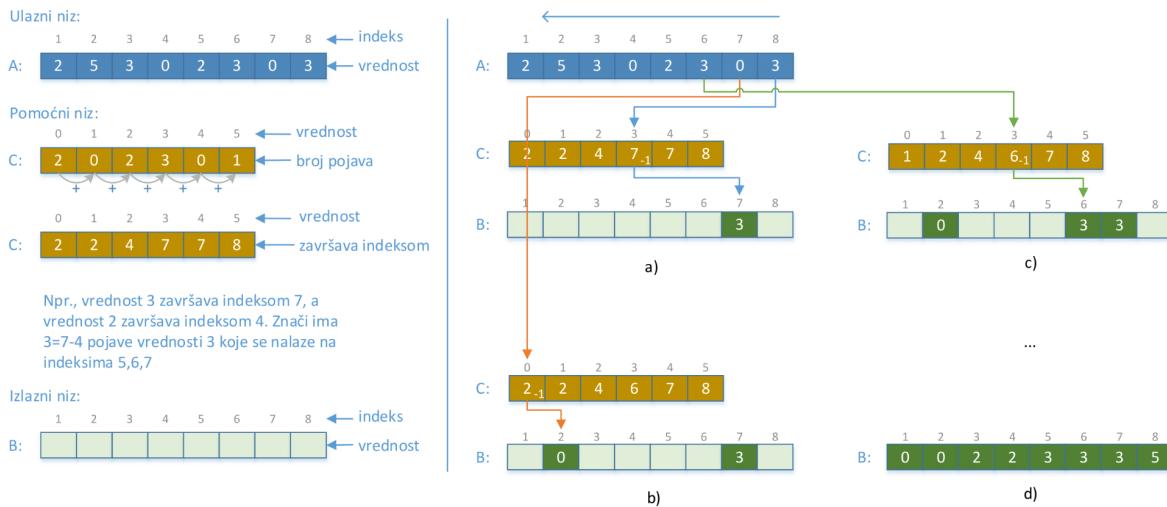
Napomena: **podrazumijeva se da se u nizu nalaze vrijednosti u opsegu od 0 do k** .

```

Sortiraj-Prebrojavanjem(A, B, k):
    for i = 0 to k:
        C[i] = 0
    for j = 1 to A.length:
        C[A[j]] = C[A[j]] + 1
    for i = 1 to k:
        C[i] = C[i] + C[i-1]
    for j = A.length downto 1:
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] - 1

```

Counting sort - Primer



Slika 23. Counting sort algoritam

Radix Sort algoritam

Koristi Counting sort algoritam kako bi sortirao brojeve **cifru-po-cifru**, polazeći od najniže cifre. Ovo je moguće jer je Counting sort **stabilan algoritam**.

```

Sortiranje-Radix(A, d):
    for cifra = 1 to d:
        Sortiraj cifru stabilnim sort algoritmom.

```

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Slika 24. Radix sort

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

Posmatra se i -ti redosledni elemenat iz skupa od n elemenata je i -ti najmanji elemenat

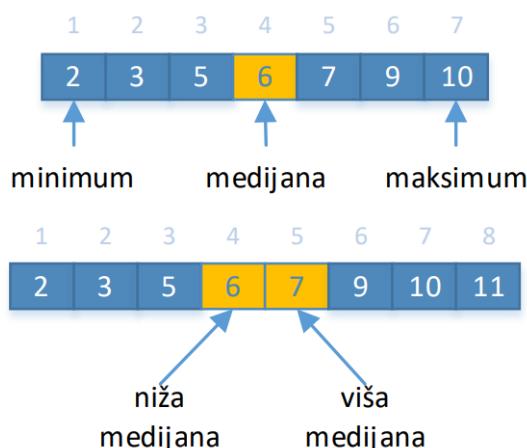
- **Minimum** - redosledno 1. elemenat, $i = 1$.
- **Maksimum** - redosledno n -ti elemenat, $i = n$.
- **Medijana** - redosledno na sredini minimuma i maksimuma.

- Za neparan broj elemenata $i = \frac{n}{2}$

- Za paran broj elemenata

- **Viša medijana:** $i = \frac{n+1}{2}$

- **Niža medijana:** $i = \frac{n-1}{2}$



Slika 25. Primjer minimuma, medijana i maksimuma

Problem - Odrediti i -ti redosledni elemenat na skupu od n različitih brojeva.

Ovo je **problem selekcije** koji se može rešiti u dva koraka:

- **Sortiranjem niza A .**
- **Izborom i -tog elementa.**

Prethodno rešenje ima složenost $O(n \log n)$ i **asimptotski je neefikasno**, traže se bolja rešenja **složenosti $O(n)$** .

Minimum i maksimum

*Koliko je najmanje operacija poređenja neophodno za pronalaženje **mimimuma ili maksimuma?** Odgovor je $n - 1$.*

MINIMUM(A)	MAKSIMUM(A)
1 $m = A(1)$	1 $m = A(1)$
2 for $i=2$ to $A.Length$	2 for $i=2$ to $A.Length$
3 if $A[i] < m$	3 if $A[i] > m$
4 $m = A[i]$	4 $m = A[i]$
5 return m	5 return m

Slika 26. Pseudokod za pronalaženje minimuma i maksimuma

Jednovremeno traženje minimuma i maksimuma

Umesto poređenja (jednog) i -tog elementa sa tekućom najmanjom i najvećom vrednosti, **trebalo bi posmatrati po 2 elementa (parove elemenata)** iz niza A :

- međusobno se uporede elementi u paru
- manji se poredi sa tekućim minimumom
- veći se poredi sa tekućim maksimumom.

Rečnik podataka i heširanje (primene, priheš i heš, kolizije, amortizovano vreme izvršavanja).

Rečnik podataka je apstraktan tip podataka koji sadrži (i održava) skup elemenata gde svaki ima **ključ** (**key**).

- Elemenat (**item**) se može posmatrati kao uređeni par **{ključ, vrednost}**.
- Bitno je da ključ bude tipa koji se može porebiti (poredak jednakost, prvenstveno).

Posmatramo ga kao dinamičku strukturu podataka - moguće dodavanje, brisanje i vrlo brza pretraga. Vrlo često se koristi i naziv **mapa**.

Osnovne operacije (složenosti $O(1)$ - koriste **heširanje**):

- Ubaci elemenat u skup (**Dodaj**).
- Ukloni elemenat iz skupa: (**Obriši**).
- Pronađi elemenat – ako postoji: (**Pretraži** - po ključu!).

Vrlo široka primena:

- **Baze podataka**
- Prevodioci: **imena → promenljive**
- Rutiranje mrežnog saobraćaja: **IP adresa → žica**
- Virtuelna memorija: **virtuelna memorija → fizička adresa**
- **Kriptografija**
- Vjerovatno **najčešće upotrebljavana** struktura podataka - zbog **brzine**.
Implementiran u savremenim programskim jezicima.

Problemi koji se javljaju pri ovakvoj implementaciji jesu da ključ mora biti cijelobrojna vrijednost, kao i da širok opseg ključeva zahtjeva ogromnu tabelu. Ovi problemi se rješavaju **priheširanjem i heširanjem**, respektivno.

Priheš

Priheš je funkcija koja **vrednost ključa prevodi u celobrojnu vrednost** (nenegativnu)

$$h_p : k \rightarrow i$$

U teoriji bi trebalo da važi:

$$\begin{aligned} x = y &\Leftrightarrow h_p(x) = h_p(y) \\ x \neq y &\Leftrightarrow h_p(x) \neq h_p(y) \end{aligned}$$

Dok je elemenat u tabeli, njegova priheš funkcija h_p se **ne sme menjati**, jer ga onda ne možemo pronaći.

Hešing

Hešing redukuje potencijalno velike vrijednosti $i = h_p(k)$ na veličinu tabele m . Formalno matematički,

$$h : U \rightarrow \{0, 1, 2, \dots, m - 1\}$$

gdje je U domen $h_p(k)$.

Primer heš funkcije - ključevi se često predstavljaju kao prirodni brojevi $k \in N$.

Npr. tekst predstavljen ASCII znacima se može predstaviti kao broj:

- Primer: "danas"

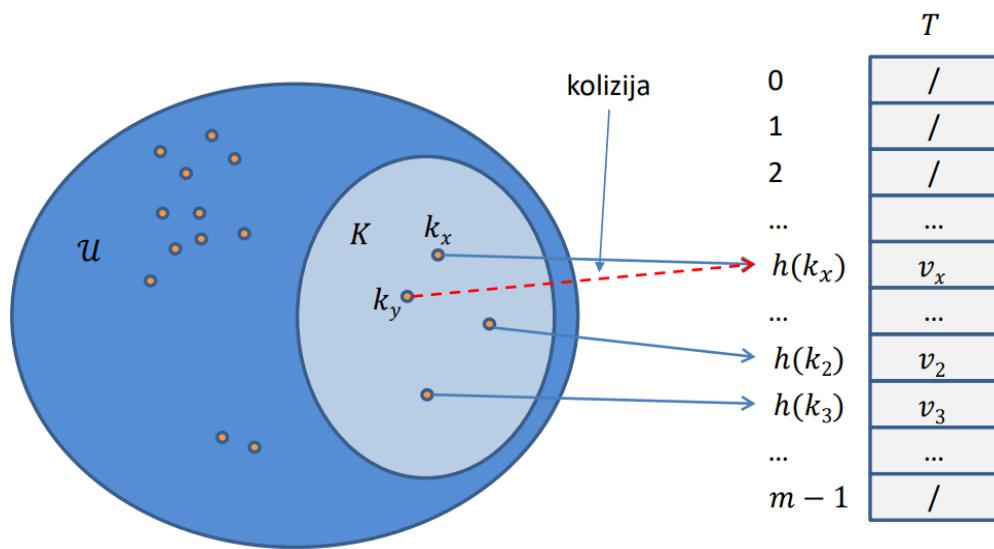
$$100 \cdot 128^4 + 97 \cdot 128^3 + 110 \cdot 128^2 + 97 \cdot 128^1 + 115 \cdot 128^0 = 27048784115$$

Stoga, heš funkcije tipično imaju kao parametar veliki prirodan broj.

Problem kolizije

Problem kolizije se zasniva na pojavi da se za dva ključa $k_x, k_y \in K$ dobijaju **iste vrijednosti** heširanja, odnosno $h(k_x) = h(k_y)$.

To znači da će se nakon smeštanja u tabelu para (k_x, v_x) (u red $h(k_x)$) se upiše vrednost v_x , par (k_y, v_y) **ne može smestiti** (nema gde) jer je red $h(k_y) = h(k_x)$ zauzet.



Slika 27. Problem kolizije

Kolizije rješavamo dvema tehnikama:

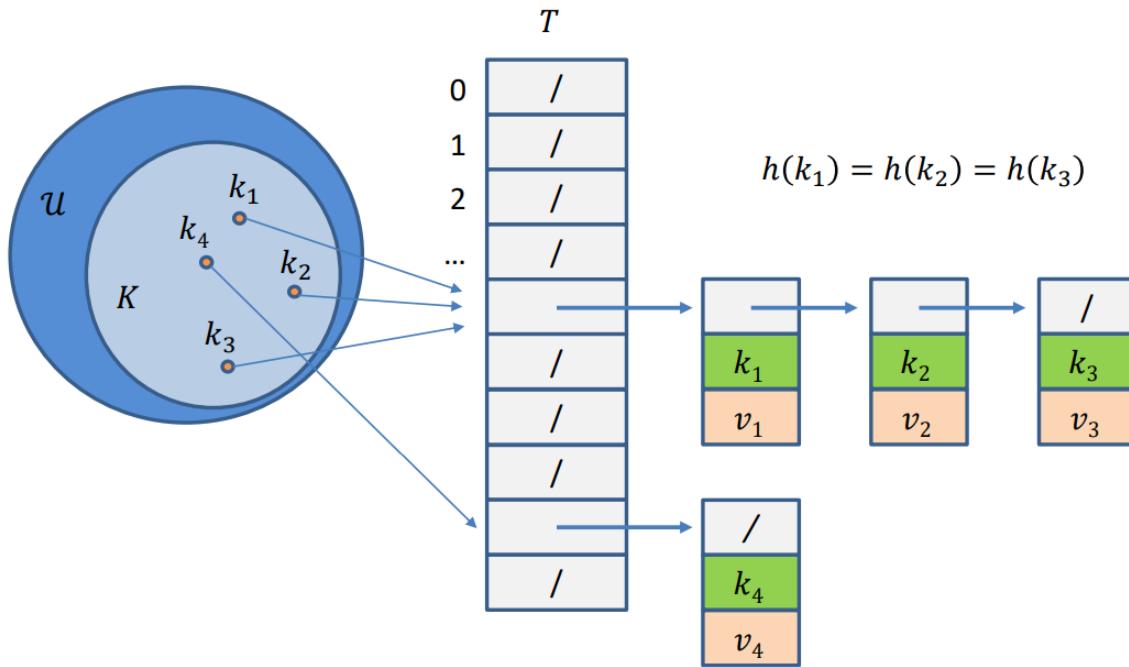
- **Heširanje ulančavanjem**
- **Otvoreno adresiranje**

Amortizovano vreme izvršavanja

Operacija ima amortizovano vreme izvršavanja $T(n)$ ako je za k operacija trajanje $\leq k \cdot T(n)$.

Grubo gledano: amortizovano vreme je prosečno vreme za ponovljene operacije.

Heširanje ulančavanjem.



Slika 28. Heširanje ulančavanjem

Elementi sa istom heš vrednosti se postavljaju u ulančanu listu, a tabela T se sastoji od pokazivača na liste.

Posledica:

- Dodavanje postavlja elemenat na početak liste. Složenost $O(1)$.
- Pretraga prolazi kroz celu listu $T[h(k_i)]$
Složenost $\Theta(n)$, kada se svih n elemenata nalazi u istom redu u T
- Brisanje elementa je veoma slično pretrazi.

Optimalno bi bilo kada bi tabela bila uniformno popunjena - svaki red u tabeli sadrži liste podjednakih dužina - **jednostavno uniformno heširanje**.

- Svaki ključ ima istu šansu da se mapira (hešira) na bilo koji red u tabeli, nezavisno od mesta gde se heširaju ostali ključevi.

- Definiše se **faktor popunjenošti** tabele $\alpha = \frac{n}{m}$, gdje je n broj elemenata koje smeštamo u tabelu T a m broj mogućih mesta (redova) u tabeli T .
- Očekivano trajanje pretrage je $\Theta(1 + \alpha)$ - ako je $m = \Omega(n)$, tada je vrijeme pretrage $\Theta(1)$.

Problem: Kako izabrati heš funkciju da se smanje kolizije?

Rješenje: Primjeri dobrih heš funkcija:

- $h(k) = \text{floor}(km)$ - ako se radi o ključevima koji su slučajno generisani realni brojevi sa uniformnom raspodjelom u opsegu $0 \leq k < 1$.
- $h(k) = k \bmod m$.
- $h(k) = \text{floor}(m(kA \bmod 1))$, gdje $0 \leq A < 1$.

Dinamičko programiranje. Primer sečenja cevi.

Dinamičko programiranje je moćna i često korištena tehnika za dizajn algoritama.

Pomaže kod tipova problema **gde naivna pretraga dovodi do eksponencijalne složenosti - dinamičko programiranje redukuje složenost na polinomsku**.

DP problemi prepoznaju se na nekoliko načina:

- **Problem ima više (mnogo) izbora** i svaki izbor ima dodeljenu vrednost tako da se traži jedan izbor sa najboljom vrednošću.
- **Broj izbora je isuviše velik** tako da upotreba *brute force* algoritma i isprobavanje svih nema smisla.
- **Do rešenja se dolazi traženjem optimalnih rešenja za jednostavnije potprobleme.**
- **Potproblemi se preklapaju** - potproblemi imaju svoje potprobleme i isti potproblem se ponavlja kod njihovog rešavanja.

Načini rešavanja **DP** problema su obično **rekurzivni**. Direktna implementacija **rekurzije** dovodi do eksponencijalne složenosti, ali se koristi **pamćenja rešenja** manjih problema, upotrebljavajući se kasnije. Takva pamćenja zavise od pristupa:

- **Pristup "od dole ka gore"** - prvo se odrede rešenja manjih problema pa se rešavaju veći problemi koristeći rešenja manjih - tako redom do glavnog problema.
- **Pristup "od gore ka dole"** - kreće se od glavnog problema čije se rešenje može dobiti rastavljanjem na manje potprobleme (koji se takođe sami mogu rastaviti), te se rešavanje takvih manjih **pamti**. Pri svakom rešavanju potproblema proverava se da li već prethodno rešen - **memoizacija**.

Koraci pri rešavanju **DP** problema su:

- **Definisati potprobleme.**
- **Pokazati da se rešenje zadatog problema može dobiti rastavljanjem na manje potprobleme - upotreba rekurzije.**
- **Prepoznati i rešiti osnovne slučajeve.**

Problem sečenja cevi.

Problem - Parčadi cevi se prave sečenjem dugačke cevi (dužine n). Kako iseći tu cev da bi se postigla najveća zarada?

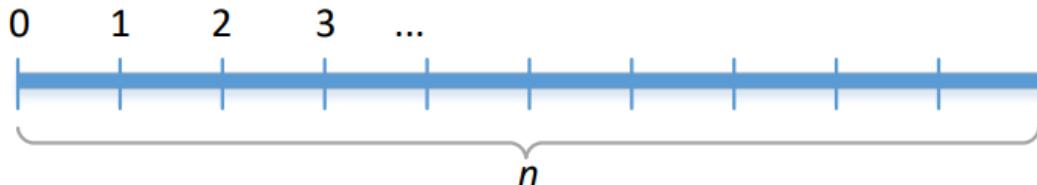
Opis problema: Firma se bavi prodajom parčadi cevi za šta je utvrđen cenovnik

Dužina L_i	1	2	3	4	5	6	7	8	9	10
Cena c_i	1	5	8	9	10	17	17	20	24	30

Slika 29. Cenovnik cevi

Cev i mesta gde se može seći:

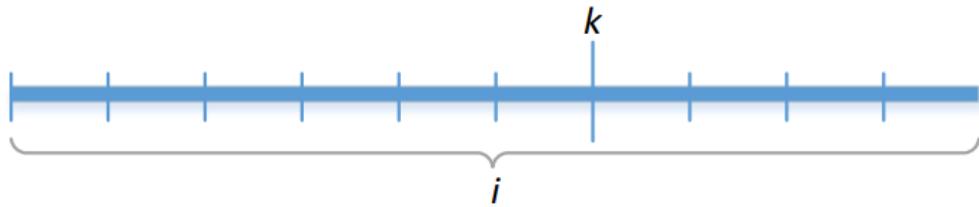
- Ako se svaka pozicija sečenja predstavi jednim bitom (1=seći, 0=nemoj) onda je rešenje problema reč od n bita.
- Broj mogućih rešenja je 2^{n-1} , što je eksponencijalna složenost



Slika 30. Cev podeljena na n delova

Sečenje će napraviti dve kraće cevi. Kada se sečenje desi na izabranoj poziciji, onda se optimalno rešenje problema svodi na rešavanje dva manja problema (jer su parčadi cevi kraći od početne), što je isti polazni problem ali je dužina drugačija. Zarada za cev dužine i je:

$$z_i = \max_{k \in 1..i} (z_k + z_{i-k})$$



Slika 31. Presečena cev na dužini k

Nadalje, ovo se može pojednostaviti ako razmišljamo da mesto reza treba usaglasiti sa tabelom cenovnika tako da se može pojaviti samo na zadatim rastojanjima L_i .

$$z_i = \max_{k \in 1..i} (c_k + z_{i-k})$$

$$z_1 = c_1 + z_0 = 1$$

$$z_2 = \max(c_1 + z_1, c_2 + z_0) = \max(2, 5) = 5$$

Korak	Sečenje cevi
Potproblemi	Maksimalna zarada za i -tu dužinu cevi
Broj potproblema	$\Theta(n)$, n je ukupan dužina
Probati	Seći na svaku jediničnu dužinu?
Broj izbora	$1..n = \Theta(n)$
Odnos potproblema	$z_i = \max_{k \in 1..i} (c_k + z_{i-k})$ $s[i] = k \text{ za koje je } \max \text{ (tj. gde je sečeno)}$
Vreme potproblema	$\Theta(i)$, $i = 1..n$
Algoritam	Izračunati $z[i]$, $i = 1..n$
Ukupno vreme	$\Theta(n^2)$
Originalan problem	$z[n]$

Slika 32. Sumiranje karakteristika algoritma

Dinamičko programiranje. Primer zagrada.

Dinamičko programiranje je moćna i često korištena tehnika za dizajn algoritama. Pomaže kod tipova problema **gde naivna pretraga dovodi do eksponencijalne složenosti - dinamičko programiranje redukuje složenost na polinomsku**.

DP problemi prepoznaju se na nekoliko načina:

- **Problem ima više (mnogo) izbora** i svaki izbor ima dodeljenu vrednost tako da se traži jedan izbor sa najboljom vrednošću.
- **Broj izbora je isuviše velik** tako da upotreba *brute force* algoritma i isprobavanje svih nema smisla.
- **Do rešenja se dolazi traženjem optimalnih rešenja za jednostavnije potprobleme.**
- **Potproblemi se preklapaju** - potproblemi imaju svoje potprobleme i isti potproblem se ponavlja kod njihovog rešavanja.

Načini rešavanja **DP** problema su obično **rekurzivni**. Direktna implementacija **rekurzije** dovodi do eksponencijalne složenosti, ali se koristi **pamćenja rešenja** manjih problema, upotrebljavajući se kasnije. Takva pamćenja zavise od pristupa:

- **Pristup "od dole ka gore"** - prvo se odrede rešenja manjih problema pa se rešavaju veći problemi koristeći rešenja manjih - tako redom do glavnog problema.
- **Pristup "od gore ka dole"** - kreće se od glavnog problema čije se rešenje može dobiti rastavljanjem na manje potprobleme (koji se takođe sami mogu rastaviti), te se rešavanje takvih manjih **pamti**. Pri svakom rešavanju potproblema proverava se da li već prethodno rešen - **memoizacija**.

Koraci pri rešavanju **DP** problema su:

- **Definisati potprobleme.**
- **Pokazati da se rešenje zadatog problema može dobiti rastavljanjem na manje potprobleme - upotreba rekurzije.**
- **Prepoznati i rešiti osnovne slučajeve.**

Problem matričnog množenja i zagrada.

Problem: odrediti optimalno izračunavanje asocijativnih izraza upotrebom zagrada. Na primjer, **množenje matrica** $A_0 A_1 A_2 \dots A_{n-1}$ ishodi sledeće rezultate:

- Izraz $(A_{5 \times 1} B_{1 \times 5}) C_{5 \times 1}$ ima 50 množenja.
- Izraz $A_{5 \times 1} (B_{1 \times 5} C_{5 \times 1})$ ima 10 množenja.
- Izraz $B_{m \times r} C_{r \times k}$ ima $m \cdot r \cdot k$ operacija množenja.

Broj mogućih rešenja je $\Omega(2^n)$, te je *brute force* rešavanje loša strategija!

Ovde se ne rešava množenje matrica nego se traži rešenje za optimalan način (redosled) izračunavanja

Ideja: razmatramo poslednje množenje

$$(A_1 \dots A_{i-1}) \underbrace{(A_i \dots A_n)}$$

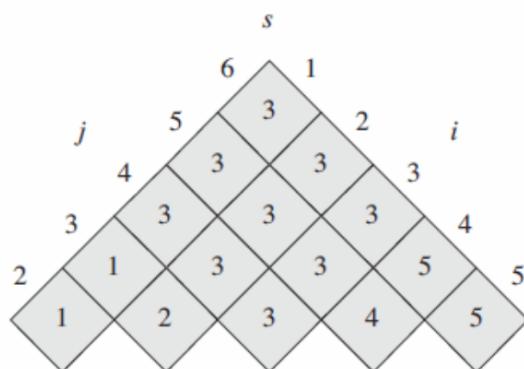
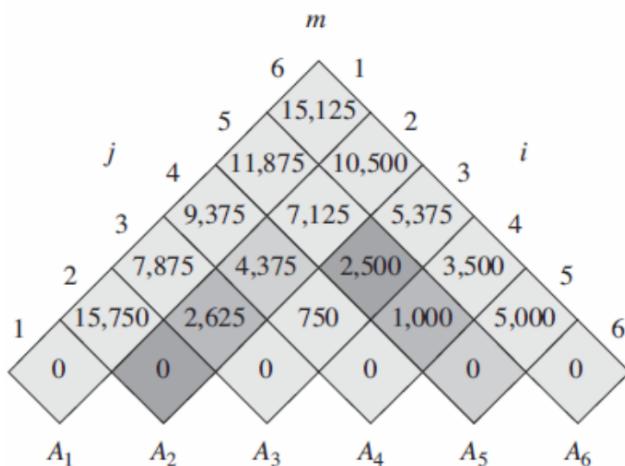
i jedno množenje pre njega

$$(A_1 \dots A_{i-1}) \underbrace{(A_i \dots A_j)} \underbrace{(A_{j+1} \dots A_n)}$$

$$\begin{array}{ccccccc} A_1 & A_2 & A_3 & A_4 & A_5 & A_6 \\ 30 \times 35 & 35 \times 15 & 15 \times 5 & 5 \times 10 & 10 \times 20 & 20 \times 25 \end{array}$$

$$m[2, 5] = \min \begin{cases} m[2, 2] + m[3, 5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13,000, \\ m[2, 3] + m[4, 5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \\ m[2, 4] + m[5, 5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11,375 \end{cases} = 7125.$$

$$q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$$



Korak	Množenje matrica
Potproblemi	$A_i \dots A_j, \quad 1 \leq i \leq j \leq n$
Broj potproblema	$\Theta(n^2)$
Probati	Gde postaviti zagradu u potproblem? $(A_i \dots A_k) \cdot (A_{k+1} \dots A_j)$
Broj izbora	$j - i = \Theta(n)$
Odnos potproblema	$m(i,j) = \min(m(i,k) + m(k+1,j) + bom)$ for $k = i, i+1, \dots, j-1$ $m(i,i) = 0$
Vreme potproblema	$\Theta(n)$
Algoritam	Izračunati $m(i,j)$ for $i = 1, \dots, n-1$ for $j = 2, \dots, n$
Ukupno vreme	$\Theta(n)\Theta(n^2) = \Theta(n^3)$
Originalan problem	$m(1,n)$

Slika 33. Sumiranje karakteristika algoritma

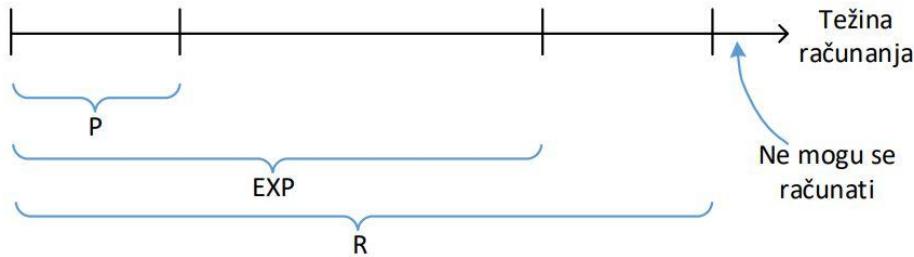
Složenost računanja i klase problema (P , NP , EXP , R , problem odlučivanja, primeri).

Za **jednostavne probleme** (pretrage, sortiranje, obilazak grafa,...) postoje efikasni algoritmi čija je vremenska složenost $O(P(n))$ ograničena nekim polinomom $P(n)$. Takođe, rješenje **P problema** se može jednostavno proveriti.

Nažalost, nemaju svi problemi rešenja u obliku jednostavnih i efikasnih algoritama, odnosno **ne mogu se svi problemi rešiti u polinomskom vremenu!**

Podjelom po složenosti problema se dobijaju osnovne klase problema:

- **P problemi** - rešivi u **polinomskom vremenu** ($\sim nc$).
- **EXP problemi** - rešivi u **eksponencijalnom vremenu** ($\sim 2^{nc}$).
- **R problemi** - rešivi u **konačnom vremenu**
- **Nerezivi problemi**



$$P \subset EXP \subset R$$

$$P \neq EXP \neq R$$

Slika 34. Podela problema

Jedan od najčešćih primera složenog problema jeste **primer 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 vрати u garažu.

Koliko ima mogućih različitih putanja?

- Broj putanja je $n \cdot (n - 1) \cdot (n - 2) \dots \cdot 3 \cdot 2 \cdot 1 = n!$
- Neka se radi o svega 20 različitih adresa, što daje
 $20! = 2.432.902.008.176.640.000$ putanja.

Rešavanje problema analizom svih mogućih putanja nema smisla i uzimanje one koja ima najmanje troškove – nije praktičan algoritam!

Onda biramo inženjerski pristup i tragamo za rešenjem koje možda nije najbolje, ali je „dovoljno dobro“.

Problem odlučivanja

Problem odlučivanja je mapiranje određenog ulaza (**problema**) na **binarnu odluku** $\{NE, DA\} = \{0, 1\}$.

Jedan primjer ovakvih problema javlja se pri pitanju “*Da li je broj paran?*” (i recimo, skup prirodnih brojeva N preslikao bi se redom u 01010101...). Drugi primjer bi bio “*Da li je broj prost?*” (i recimo, skup prirodnih brojeva N sada se slika na 11101010...) i sl.

Dakle, isti prirodan broj (program) mapira se različito u zavisnosti od problema, te ispada da postoji više problema nego programa.

$$|\mathbb{N}| < |\mathcal{P}(\mathbb{N})| \text{ ili } |\mathbb{N}| < |\mathbb{R}|$$

Svaki od ovih problema preslikava skup N na neki njegov partitivan skup $P(N)$, čiji se broj članova može smatrati istom kardinalnošću kao i skup realnih brojeva, R .

„Nalik“ slični problemi

Problemi čije postavke imaju međusobne sličnosti ali su različitih klasa.

Primjeri:

- Traženje najkraćeg/najdužeg puta u grafu
 - Traženje **najkraćeg puta** u grafu $G(V, E)$ od datog čvora je algoritam složenosti $O(|E| + |V| \log |V|)$.
 - Traženje **najdužeg puta** u grafu između dva čvora je **NP-kompletan** problem.
- 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 odrediti u $O(|E|)$ vremenu.
 - Hamiltonov ciklus je zatvorena putanja koja sadrži svaki čvor grafa. To je **NP-kompletan** problem.

NP problemi (definicija P i NP problema, vrste NP problema, redukcija).

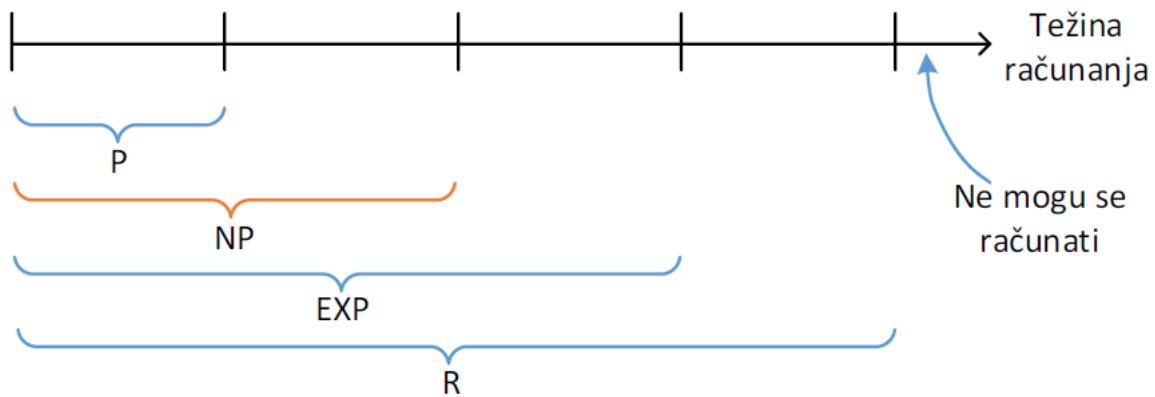
P -problem su problemi čiji algoritmi rešavanja imaju vremensku složenu predstavljenu kao **polinomsku funkciju zavisnu od broja ulaza**.

NP -problem su rešivi u polinomskom vremenu uz upotrebu **nedeterminističkog modela računanja** (svako odlučivanje (grananje) može se istovremeno sračunati sa ostalim). Nedeterministički model računanja se može sprovoditi na determinističkoj mašini (~klasičan računar), ali sporo (rekurzivno, bez mogućnosti jednovremenog računanja).

Suštinski, *možda ne postoji algoritam koji bi rešio problem u polinomskom vremenu, ali postoji način da se neko dobijeno rešenje zaista proveri u polinomskom vremenu*.

NP sadrži **P probleme** (prethodna rečenica, uz razliku da za **P probleme** postoji algoritam koji bi rešio problem u polinomskom vremenu) - svaki **P problem** je ujedno i **NP problem** jer ne samo da se za dato rešenje **P problema** može proveriti da je ono korektno, nego se za **P** vreme može i pronaći takvo rešenje.

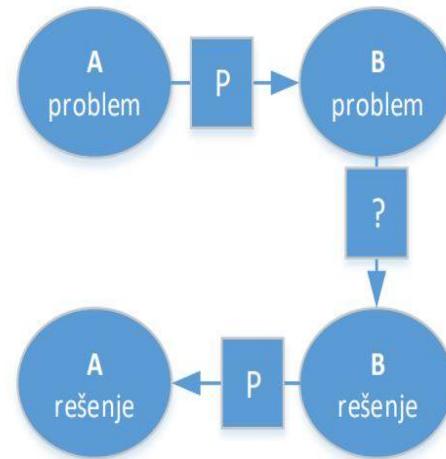
Vlada opšte mišljenje da **NP problemi** nemaju algoritam rešavanja u polinomskom vremenu kao **P problemi** (da za svaki problem za koji neki algoritam može brzo proveriti dato rešenje takođe postoji algoritam koji brzo pronalazi takvo rešenje), te da važi da **$P \neq NP$** .



Slika 35. NP problemi

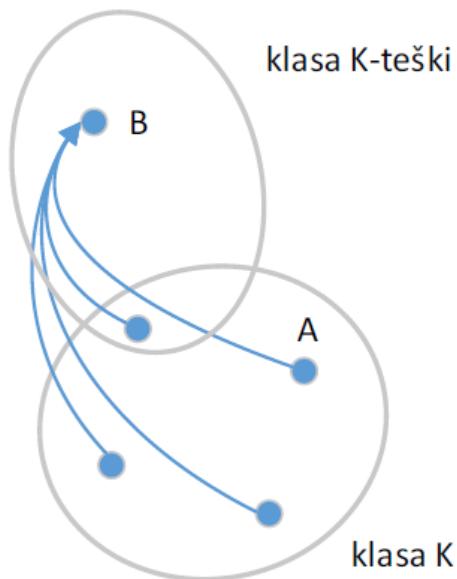
Svođenje ili redukcija problema

Redukcija problema A je transformacija tog problema u drugi problem B čijim se rešavanjem dobija rešenje osnovnog problema A .



Slika 36. Redukcija (transformacija) problema

Ako se svaki problem iz klase K (A problem) može redukovati na problem B onda B pripada klasi **K -teških** problema.



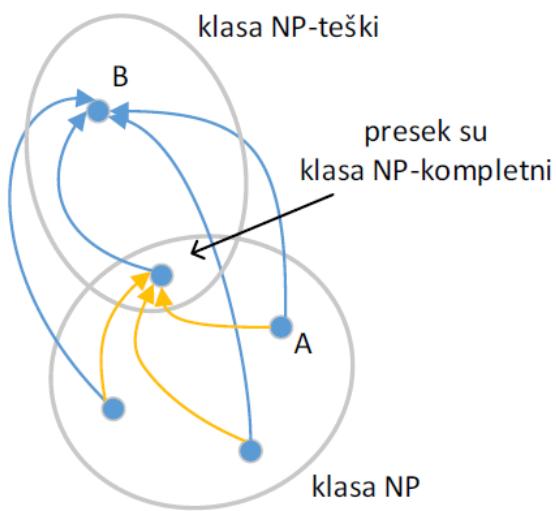
Slika 37. K-teški problemi

Vrste NP problema:

- “Običan” **NP problem** je problem čije se rešenje može proveriti u polinomskom vremenu.
- **NP-težak problem** je problem težak barem kao svaki **NP problem** (ali se njegovo rešenje ne može proveriti u polinomskom vremenu) - *skoro teški kao najteži problemi u NP*.
- **NP-kompletan problem** je presjek skupova **NP** i **NP-teških problema**.

Problem je tipa NP-kompletan ako zadovoljava 2 uslova:

- On je podklasa **NP problema** (rešenje se može proveriti u polinomskom vremenu).
- Ako za problem postoji algoritam koji se izvršava u polinomskom vremenu, onda postoji način da se svaki problem u **NP** konvertuje u taj problem na način da se svi oni izvršavaju u polinomskom vremenu (postoji redukcija nekog problema u taj problem).



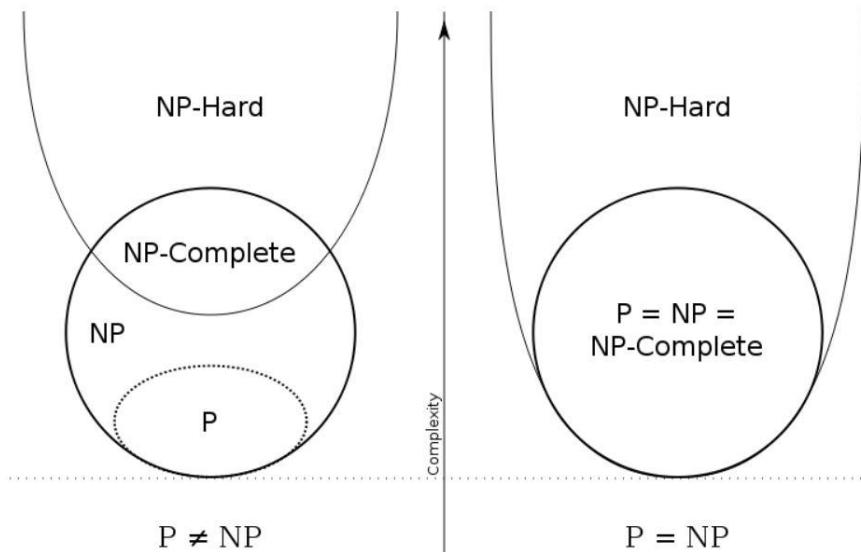
Slika 38. NP vs NP-hard vs NP-complete problemi

NP-kompletni problemi postoje u brojnim oblastima - Bulova logika, grafovi, aritmetika, dizajn mreža, skupovi i particionisanje, skladištenje i očitavanje podataka... Rešavaju se tehnikama koje daju približno rešenje – **rešenja nisu optimalna**.

Tri koncepta se koriste da se pokaže da je problem NP-kompletan:

- **Problem** se svodi na **problem odluke** (tako da se rešenje problema svodi na odgovor „da“/„ne“).
- **Problem se redukuje** na drugi problem za koji se prepostavlja da je klase **NP** i tako redom dok se transformacijama problem ne svede na **prvi NP-kompletan problem**.
- **Prvi NP-kompletan problem** je problem za koji je dokazano da je **NP-kompletan**.

Odnos: P, NP, NP-kompletan, NP-težak



Slika 39. P vs NP vs $NP\text{-hard}$ vs $NP\text{-complete}$

Algoritam za nalaženje najduže zajedničke podsekvence (definicije sekvence, stringa,..., LCS algoritam).

String je niz (**sekvencia** - *niz elemenata gde je bitan redosled*) karaktera (iz datog skupa karaktera). Skup karaktera čine **slova** (mala i velika), **cifre**, **znaci interpunkcija**, **matematički** i **neki drugi simboli**.

Podstring (substring) je deo stringa koji sadrži **uzastopne karaktere**.

Problem najduže zajedničke podsekvence - za dva stringa pronalači najdužu podsekvenku koja se nalazi u oba stringa.

Algoritam koristi **dinamičko programiranje** i ima dve faze:

- Izgrađuje pomoćnu tabelu dimenzije $(n + 1)(m + 1)$ gde su n i m dužine datih stringova – složenost faze je $\Theta(nm)$.
- Pomoću tabele nalazi **LCS** – složenost je $O(n + m)$.

- $X = \text{"ABCBDAB"}, Y = \text{"BDCABA"}$
- | | | | | | | | | |
|-----|-------|---|---|---|---|---|---|---|
| | j | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| i | y_j | B | D | C | A | B | A | |
| 0 | x_i | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 2 | B | 0 | 1 | 1 | 1 | 2 | 2 | 2 |
| 3 | C | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| 4 | B | 0 | 1 | 1 | 2 | 2 | 3 | 3 |
| 5 | D | 0 | 1 | 2 | 2 | 2 | 3 | 3 |
| 6 | A | 0 | 1 | 2 | 2 | 3 | 3 | 4 |
| 7 | B | 0 | 1 | 2 | 2 | 3 | 4 | 4 |
- Rešenje = "BCBA" dužine 4
- ```

LCS_DUŽINA(X, Y)
1 $n=X.Length, m=Y.Length$
2 $L[0..n,0..m], b[1..n,1..m]$ // tabele
3 for $i=1$ to n
4 $L[i,0] = 0$
5 for $j=1$ to m
6 $L[0,j] = 0$
7 for $i=1$ to n
8 for $j=1$ to m
9 if $X[i] == Y[j]$
10 $L[i,j] = L[i-1,j-1] + 1, b[i,j] = \nwarrow$
11 elseif $L[i-1,j] \geq L[i,j-1]$
12 $L[i,j] = L[i-1,j], b[i,j] = \uparrow$
13 else
14 $L[i,j] = L[i,j-1], b[i,j] = \leftarrow$
15 return L, b

```

Slika 40. LCS algoritam

### Algoritmi podudaranja stringova (*naivni, Rabin-Karp, konačni automat, KMP*).

Imamo dva stringa: string teksta  $T$  (dužine  $n$ ) i string šablona  $P$  (dužine  $m$ ,  $m \leq n$ ), i želimo da pronađeno sve pojave  $P$  u  $T$ .

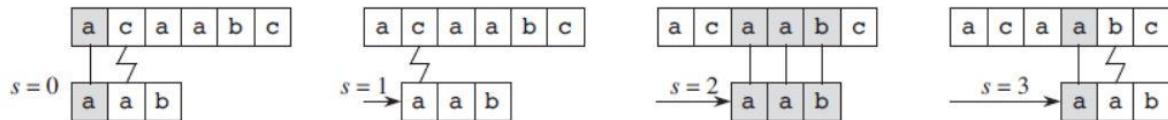
- **Algoritam grube sile** (brute force) je **naivan** algoritam gde se vrši poređenje šablona sa podstringom koji počinje na poziciji  $k$ , slovo po slovo.
- **Rabin-Karp** algoritam poredi heš vrednost šablona i heš vrednosti podstringova (iz teksta) dužine šablona. Za računanje heš vrednosti podstringova koristi se rolling heš.
- **Konačni automat** tokom poređenja sa šablonom prelazi iz stanja u stanje.
- **KMP (Knuth-Morris-Pratt)** algoritam – pogodan kada šablon ima podstringove koji se ponavljaju, i tako da tokom poređenja šablon „pomera“ iza podstringa koji se ponavlja.

| Algoritam                          | Pretprocesiranje | Poređenje                    | Zauzeće memorije |
|------------------------------------|------------------|------------------------------|------------------|
| Algoritam grube sile               | -                | $\Theta(nm)$                 | -                |
| Rabin-Karp algoritam               | $\Theta(m)$      | $\Omega(n + m), O((n - m)m)$ | $O(1)$           |
| Konačni automat                    | $O(m \Sigma )$   | $O(n)$                       | $O(m \Sigma )$   |
| KMP (Knuth-Morris-Pratt) algoritam | $\Theta(m)$      | $\Theta(n)$                  | $\Theta(m)$      |

Slika 41. Karakteristike algoritama

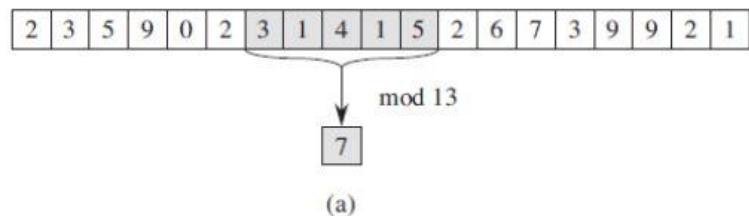
### Algoritam grube sile

Naivan algoritam - poređenje ne koristi informacije iz prethodnih poređenja.

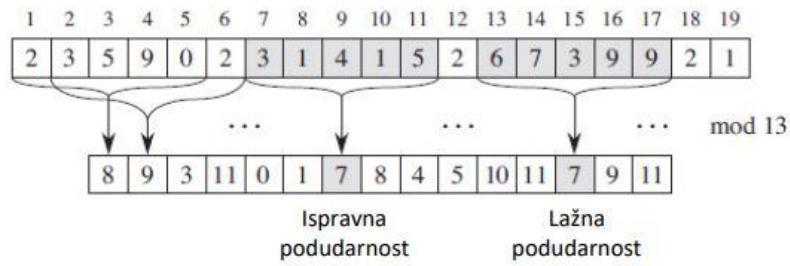


Slika 42. Algoritam grube sile

### Rabin-Karp algoritam



(a)

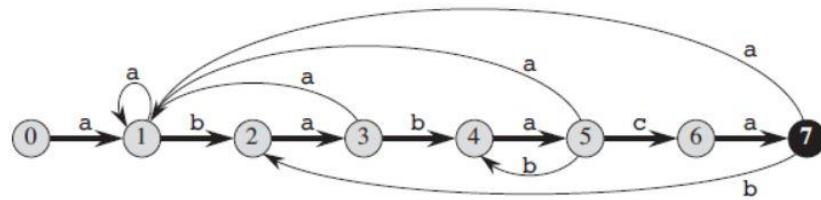


Slika 43. Rabin-Karp algoritam

## Konačni automat

Šablon je: **ababaca**

Dolazak u stanje 7 odgovara podudaranju šablonu



(a)

| state | input |   |   | $P$ |
|-------|-------|---|---|-----|
|       | a     | b | c |     |
| 0     | 1     | 0 | 0 | a   |
| 1     | 1     | 2 | 0 | b   |
| 2     | 3     | 0 | 0 | a   |
| 3     | 1     | 4 | 0 | b   |
| 4     | 5     | 0 | 0 | a   |
| 5     | 1     | 4 | 6 | c   |
| 6     | 7     | 0 | 0 | a   |
| 7     | 1     | 2 | 0 | a   |

$i$  — 1 2 3 4 5 6 7 8 9 10 11  
 $T[i]$  — a b a b a c a b a  
state  $\phi(T_i)$  0 1 2 3 4 5 4 5 6 7 2 3

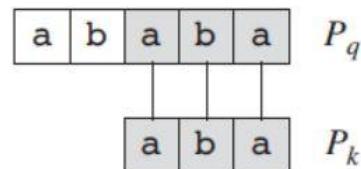
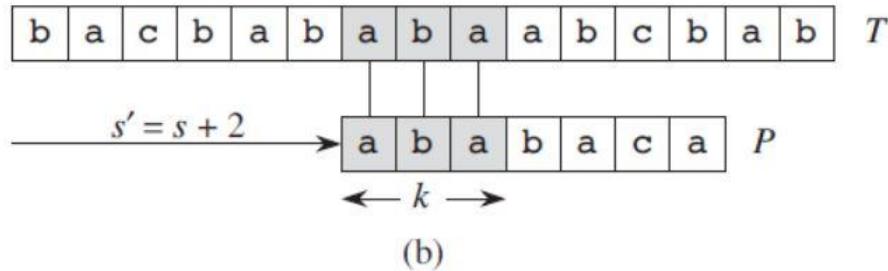
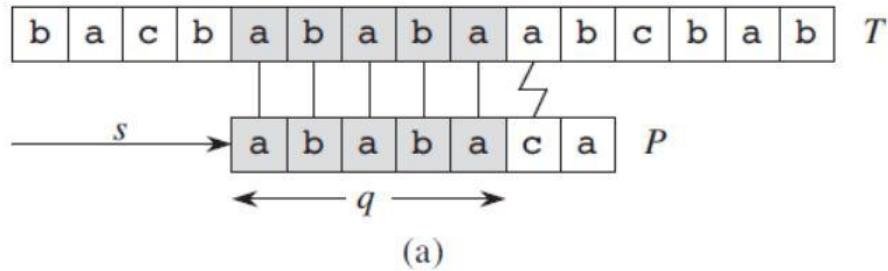
**Slika 44.** Konačni automat

## KMP algoritam

Algoritam izbegava računanja funkcije prelaza stanja  $\delta$  tako što koristi niz  $\pi[1..m]$  koji se popuni u amortizovanom vremenu  $\Theta(m)$ .

Ukupna složenost: priprema  $\Theta(m)$  + pretraga  $\Theta n = \Theta(n + m)$ .

Primer poređenja sa tekstom sa pomerajem  $s$ . Neslaganje karaktera šablona „preskače“ ponovno poređenje početnih  $k$  karaktera šablona.



**Slika 45.** KMP algoritam

# B Grafovi

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

Grafovi su matematičke strukture za **modelovanje odnosa** između parova objekata.

Sastoje se od:

- **Čvorova (objekata)** - *nodes, vertices* (skup čvorova jednog grafa označava se sa  $V$ ).
- **Grana (koje povezuju parove objekata)** - *edges* (skup grana jednog grafa označava se sa  $E$ ).

Matematički zapisano,  $G = (V, E)$ .

Primjena:

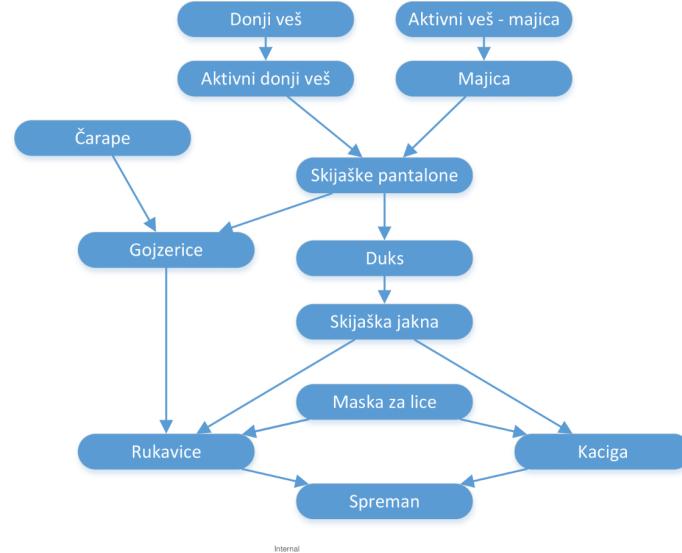
- Sistemi datoteka u operativnom sistemu.
- Struktura *website-a* - čvorovi su *stranice*, dok su usmjerenе grane *hiperlinkovi*.
- Mreža puteva - čvorovi su *raskrsnice*, a grane *putevi*.
- Elektrodistributivna mreža - čvorovi su transformatorske stranice, a grane su vodovi.

**Tipovi grafova:**

- **Sa stanovišta usmjerenosti grana:**
  1. *Neusmjereni (neorjentisani)*
  2. *Usmjereni (orjentisani)*
  3. *Mješoviti (dio grana orjentisan)*
- **Sa stanovišta postojanja petlji:**
  1. *Ciklični*
  2. *Aciklični*
- **Sa stanovišta parametara grana:**
  1. *Direktni graf*
  2. *Težinski graf (grane nose određenu težinu)*

## Primer usmerenog acikličnog grafa

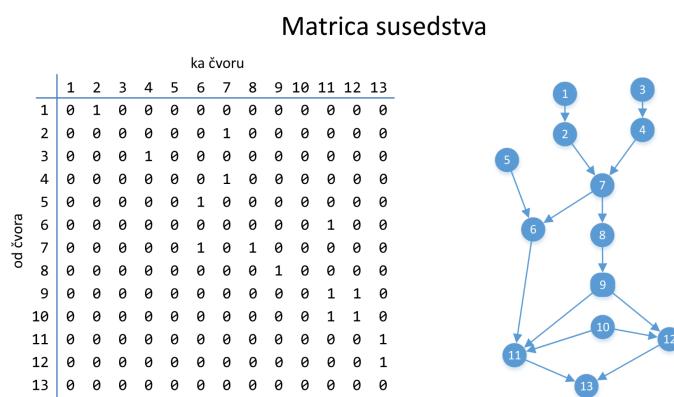
- Oblačenje skijaša ...



**Slika 46.** Primjer usmjerenog acikličnog grafa

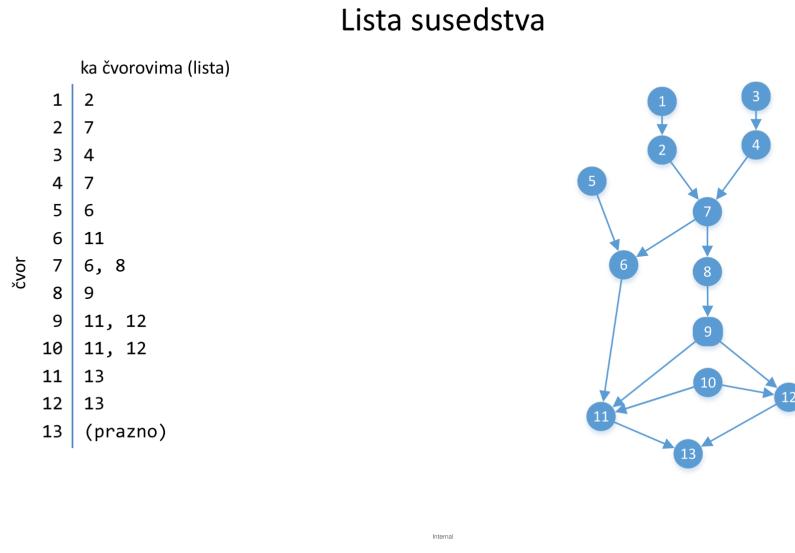
Programski, grafovi se mogu **predstaviti** na više načina:

- **Matricom susjedstva** - matrica u kojem su i redovi i kolone numerisani kao čvorovi, a grana se predstavlja postavljanjem jedinice na red i kolonu datih čvorova koji čine granu. Memorijsko zauzeće ovakve predstave jeste  $O(V^2)$ , što je dosta neefikasno za grafove sa mnogo čvorova i malo grana (sadrži dosta nula).



**Slika 47.** Reprezentacija grafa matricom susjedstva

- **Lista susjedstva** - za svaki čvor definiše se lista čvorova sa kojima je on povezan. Memorijsko zauzeće ovakvog pristupa jeste  $O(|V| + |E|)$ , što je efikasnije od matrice susjedstva u prosječnom slučaju.



**Slika 48.** Reprezentacija grafa listom susjedstva

## Topološko sortiranje grafova. Pretrage grafova.

### Topološko sortiranje grafova

**Topološko sortiranje** jestе algoritam primjenjiv nad **usmjerenim acikličnim grafovima** koji izrađuje poredak čvorova gdje je čvor  $u$  ispred čvora  $v$ , kada ih spaja grana  $(u, v)$ .

Ovakav poredak **ne mora biti jedinstven**.

Algoritam radi na sledeći način:

- Svakom čvoru pridružimo broj ulaza.
- Biramo čvor bez ulaza, dodamo ga u izlazni niz i uklonimo iz grafa.
- Ponavljamo prethodni korak sve dok postoje neposjećeni čvorovi.

**Složenost** ovakvog algoritma jestе  $O(|V| + |E|)$ .

```
Topološko-Sortiranje(G):
```

```
 obraditi = []
 rezultat = []
```

```
 for each u in G.V:
 u.br_ul = 0
```

```

for each u in G.V:
 for each v in G.Susjedi(u):
 v.br_ul += 1
for each u in G.V:
 if u.br_ul == 0:
 obraditi.dodaj(u)

while obraditi:
 u = obraditi.izbaci()
 rezultat.dodaj(u)

 for each v in G.Susjedi(u):
 v.br_ul -= 1
 if v.br_ul == 0:
 obraditi.dodaj(v)

return rezultat

```

## Pretrage grafova

**Pretrage grafova** jesu metode posjećivanja svakog čvora i svake grane u grafu.

Postoje nekoliko vrsta algoritama pretrage grafova:

- **Pretraga u širinu (*Breadth First Search*)**
- **Pretraga u dubinu (*Depth First Search*)**

### Breadth First Search

Jedan od najjednostavnijih algoritama pretrage grafova jeste **Breadth First Search algoritam (BFS)** i predstavlja osnovu za neke druge algoritme (*Dijkstra, Primovo minimalno stablo razapinjanja,...*).

Algoritam definiše **najkraći put (najmanji broj grana koji je potreban)** od izvornog čvora do svih čvorova u grafu.

Radi sa **orijentisanim** i **neorijentisanim** grafovima.

Pretraga polazi od izvornog čvora i pretražuje sve njegove susjede, pa nakon toga njihove susjede,... (“**frontalna**” **pretraga**). Pri tome se pazi da se ne posjećuju već posjećeni čvorovi.

Tokom rada algoritam “boji” čvorove u različite boje:

- **bijela** - čvor još nije posjećen
- **siva** - čvor čeka na obradu
- **crna** - čvor je posjećen
- Bojenje se uvodi da bi se pazilo koji je čvor **posjećen**, odnosno **neposjećen**.

U algoritmu se dodaju **značke** (atributi svakom čvoru):

- $d$  - duljina od izvornog čvora
- $pred$  - veza ka prethodnom čvoru (**prethodnik**, koji čvor je **roditelj-predak** datog čvora)

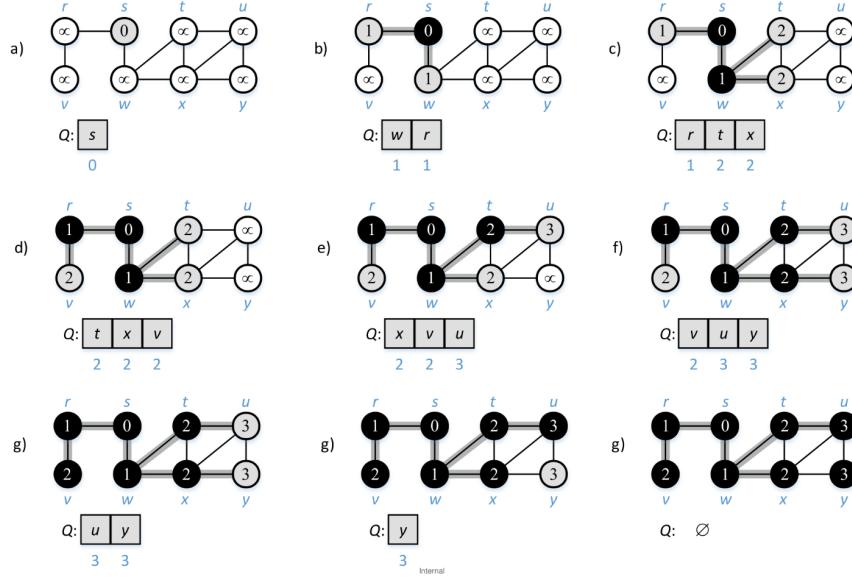
Vrijeme izvršavanja algoritma je  $O(|V| + |E|)$ .

```
BFS(A, s):
 for each u in G.V \ {s}:
 u.boja = BIJELA
 u.d = Inf
 u.pred = NIL

 s.boja = SIVA
 s.d = 0
 s.pred = NIL
 neobrađeni = []
 neobrađeni.dodaj(s)

 while neobrađeni:
 u = neobrađeni.izbaci()
 for each v in G.Susjedi(u):
 if v.boja == BIJELA:
 v.boja = SIVA
 v.d = u.d + 1
 v.pred = u
 neobrađeni.dodaj(v)
 u.boja = CRNA
```

## BFS primer



**Slika 49.** Primjer BFS algoritma

**BFS** algoritam formira **stablo pretrage u širinu** (prikazano debelim sivim linijama). Ako graf nije povezan u potpunosti, BFS algoritam ne može istražiti čitav graf.

## Depth First Search

**Pretraga u dubinu** “prodire” u graf što god može dalje udaljavajući se od izvornog čvora, ostavljajući sa strane neistražene čvorove i grane. Kada dođe do kraja, pretraga se nastavlja od poslednjeg neistraženog čvora.

U algoritmu se dodaju **vremenske značke** (atributi svakom čvoru):

- $d$  - trenutak kada je čvor otkriven
- $f$  - trenutak kada je obrada tog čvora završena

Čvorovi imaju svoje boje koje imaju potpuno isto značenje kao i u **BFS** algoritmu.

Vrijeme izvršavanja algoritma je  $O(|V| + |E|)$ .

Algoritam se može iskoristiti i za topološko sortiranje:

- Izračuna se  $v.f$  za svaki čvor i kada je čvor obrađen, smješta se u listu  $L$ .
- Obrnuta lista  $L$  predstavlja topološko sortiran graf.

```
global vrijeme = 0
```

```
DFS(G):
```

```

for each u in G.V:
 u.boja = BIJELA
 u.pred = NIL
for each u in G.V:
 if u.boja == BIJELA:
 DFS-Posjeti(G, u)

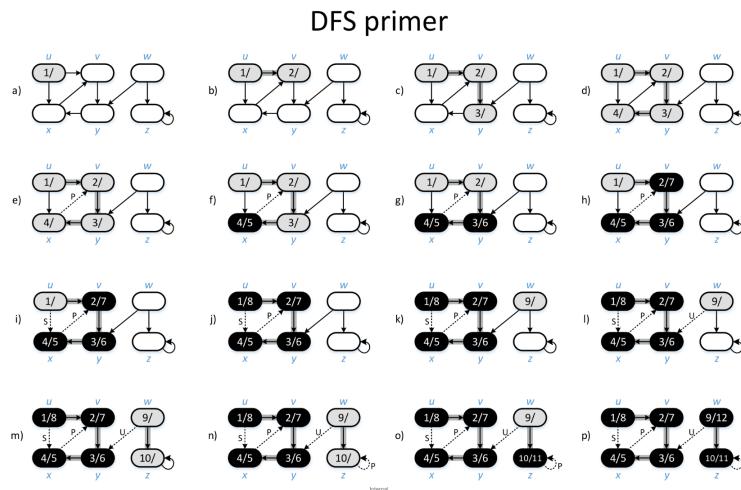
DFS-Posjeti(G, u):
 vrijeme += 1
 u.d = vrijeme
 u.boja = SIVA

 for each v in G.Susjedi(u):
 if v.boja == BIJELA:
 v.pred = u
 DFS-Posjeti(G, v)

 u.boja = CRNA
 vrijeme += 1
 u.f = vrijeme

```

**DFS algoritam** formira **stablo pretrage u dubinu** (prikazano debelim sivim linijama na slici).



**Slika 50. DFS algoritam**

Može se koristiti pri detekciji kružne putanje kada se kretanjem po grafu naiđe na ranije posjećen čvor.

Po **DFS** algoritmu, grane se klasificuju na:

- **Grane stabla (tree edge)** - grane preko kojih se posjećuju čvorovi.
- **Povratne grane (back edge)** - grana ka prethodno posjećenom čvoru u stablu.
- **Preskočne grane (forward edge)** - grana  $(u, v)$  koja potiče iz  $u$  i završava se u  $v$  i važi da je  $u.d < v.d$ .
- **Unakrsne grane (cross edge)** - grane između dva podstabla (koje nemaju zajedničke čvorove - pretke).

## Najkraći put u grafu. *Dijkstra* algoritam.

Dok algoritmi kao što su **BFS** i **DFS** pronalaze **najkraći put u netežinskom grafu**, **najkraći put u težinskom grafu** pronalaze **Bellman-Ford** i **Dijkstra** algoritmi.

Ovakvi algoritmi ne moraju sadržati najmanji broj grana u konačnom rješenju u kontekstu najkraćeg puta.

Algoritmi koriste princip **relaksacije grane** (ako je prethodno zabilježena težina čvora od početnog čvora veća nego **zbir** težine njegovog susjednog čvora i težine grane koja povezuje taj čvor sa tim susjednim čvorom, nova težina postaje taj **zbir**).

```
Relaksacija(u, v, w):
 if v.d > u.d + w(u, v):
 v.d = u.d + w(u, v)
 v.pred = u
```

**Dijkstra** algoritam zahtjeva **nenegativne težine**. Radi slično kao **Primov** algoritam, u smislu da bira najmanji čvor (odnosno, čvor sa *do sada najmanjom udaljenošću od početnog čvora*) ali iz sortiranog niza svih čvorova (odnosno, trenutni **najbliži čvor** koji je povezan sa već posjećenim **čvorom**).

**Složenost algoritma je**  $O(V^2)$ .

```
Dijkstra(G, w, s):
 for each v in G.V:
 v.d = Inf
 v.pred = NIL
 s.d = 0

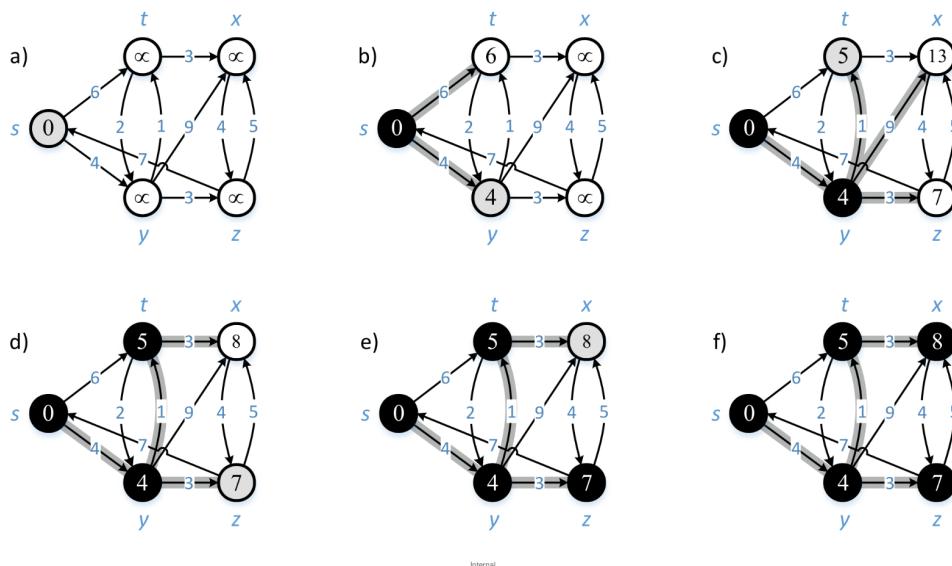
 S = []
 Q = G.V
```

```

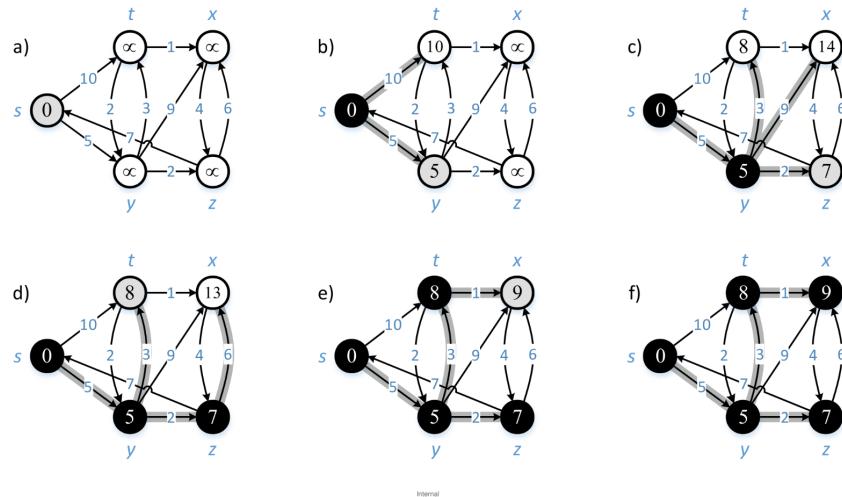
while Q:
 u = Izdvoj-Minimum(Q)
 S.dodaj(u)
 for each v in G.Susjedi(u):
 Relaksacija(u, v, w)

```

### Dajkstra – primer A



### Dajkstra – primer B



**Slika 51.** Primjeri Dijkstra algoritma

## Najkraći put u grafu. *Bellman-Ford* algoritam.

Dok algoritmi kao što su **BFS** i **DFS** pronalaze **najkraći put u netežinskom grafu**, **najkraći put u težinskom grafu** pronalaze **Bellman-Ford** i **Dijkstra** algoritmi.

Ovakvi algoritmi ne moraju sadržati najmanji broj grana u konačnom rješenju u kontekstu najkraćeg puta.

Algoritmi koriste princip **relaksacije grane** (ako je prethodno zabilježena težina čvora od početnog čvora veća nego **zbir** težine njegovog susjednog čvora i težine grane koja povezuje taj čvor sa tim susjednim čvorom, nova težina postaje taj **zbir**).

```
Relaksacija(u, v, w):
 if v.d > u.d + w(u, v):
 v.d = u.d + w(u, v)
 v.pred = u
```

**Bellman-Ford** algoritam koristi principe **dinamičkog programiranja**:

- Najdužu direktnu putanju činiće svi čvorovi (ima ih  $n$ ) i ona će tada imati  $n - 1$  granu.
- Ako se u  $n - 1$  prolaza relaksiraju sve grane grafa, onda će poslije prvog prolaza **sigurno biti relaksirana** grana  $s \rightarrow v_1$  i tako redom ( $s \rightarrow v_2$  jer će biti relaksirana  $v_1 \rightarrow v_2, \dots$ )

Složenost algoritma je  $O(|V||E|)$ .

Algoritam može raditi i sa negativnim težinama te tako može služiti kao **detekcija kružne putanje negativnog pojačanja**.

```
Bellman-Ford(G, w, s):
 for each v in G.V:
 v.d = Inf
 v.pred = NIL
 s.d = 0

 for i = 1 to |V|-1:
 for each (u, v) in E:
 Relaksacija(u, v, w)

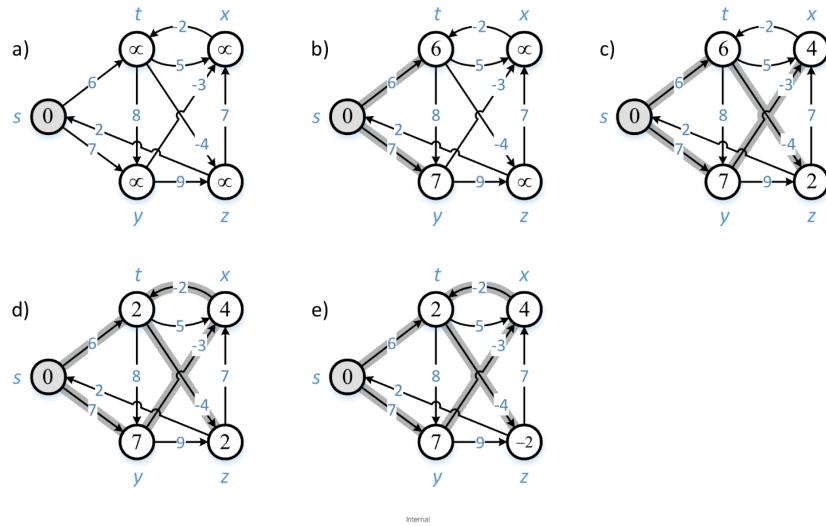
 for each (u, v) in E:
```

```

if v.d > u.d + w(u, v):
 return 'Postoji negativna kružna putanja!'

```

### Belman-Ford – primer



**Slika 52.** Primjer Bellman-Ford algoritma

### Minimalno razapinjuće stablo. Kruskalov algoritam.

Ako se posmatra **neusmijeren težinski povezan graf**  $G = (V, E)$ , **minimalno stablo razapinjanja** je aciklični graf sačinjen od grana  $T \subseteq E$  koje povezuju sve čvorove  $V$  tako da je suma težina u  $T$  minimalna, odnosno

$$\min_T w(T) = \sum_{(u, v) \in T} w(u, v)$$

Do rješenja problema dolazi se postepenom izgradnjom, odnosno postepenim **biranjem sigurnih grana** i dodavanjem jedne po jednu granu u stablo. Stablo se izgrađuje dok se ne povežu svi čvorovi.

**Kruskalov algoritam** na početku ima **šumu stabala** od po jednog čvora (svaki čvor je zaseban "skup") i kao **sigurnu granu** bira granu najmanje težine koje povezuju dva stabla, tako "spajajući" skupove.

**Složenost Kruskalovog algoritma je**  $O(|E| \log |V|)$ .

```

MSR-Kruskal(G, w):
 Stablo = []
 for each v in G.V:
 Napravi-Skup(v)

 Sortiraj-Grane-Po-Težini()

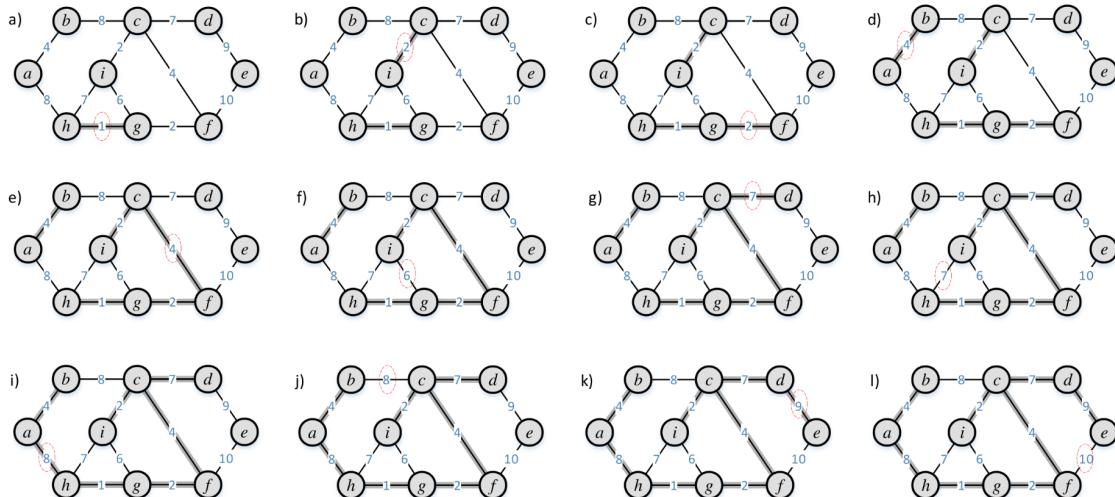
 for each (u, v) in Sortirane-Grane:
 if Pronađi-Skup(u) != Pronađi-Skup(v):
 Stablo.Dodaj((u, v))
 Unija-Skupova(u, v)

 return Stablo

```

Operacije *Unija-Skupova*, *Pronađi-Skup* i *Napravi-Skup* možemo realizovati pomoću **povezanih lista ili stabala**.

### Primer Kruskal



**Slika 53.** Primjer Kruskalovog algoritma

## Minimalno razapinjuće stablo. *Primov* algoritam.

Ako se posmatra **neusmjeren težinski povezan graf**  $G = (V, E)$ , **minimalno stablo razapinjanja** je aciklični graf sačinjen od grana  $T \subseteq E$  koje povezuju sve čvorove  $V$  tako da je suma težina u  $T$  minimalna, odnosno

$$\min_T w(T) = \sum_{(u, v) \in T} w(u, v)$$

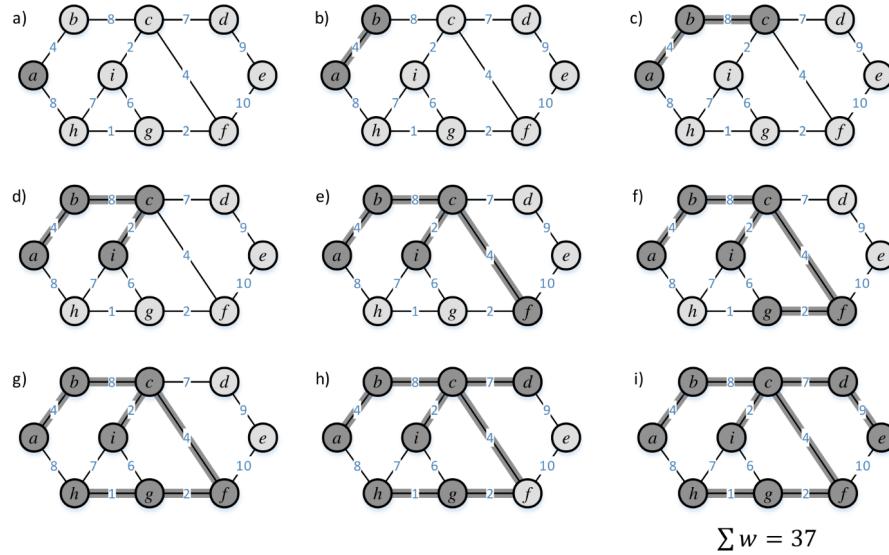
Do rješenja problema dolazi se postepenom izgradnjom, odnosno postepenim **biranjem sigurnih grana** i dodavanjem jedne po jednu granu u stablo. Stablo se izgrađuje dok se ne povežu svi čvorovi.

**Primov algoritam** je **greedy algoritam** - radi na principu biranja sigurne grane. Počinje se od nekog čvora  $r$ , zapišu se sve njegove grane u niz grana i bira se **sigurna** grana (grana koja izvire iz čvora sa najmanjom težinom), te se tako prelazi iz čvora u čvor i ponavlja isti postupak (opet se bira grana najmanje težine od svih zapisanih grana koje povezuju stablo sa neposjećenim čvorom). Složenost Primovog algoritma je  $O(|E| \log |V|)$ , ako se koristi **min-heap** struktura. Ako se koristi **Fibonačijev heap**, dobija se složenost  $O(|E| + |V| \log |V|)$ .

```
MSR-Prim(G, w, r):
 for each u in G.V:
 u.ključ = Inf
 u.pred = NIL
 r.ključ = 0
 Q = G.V

 while Q:
 u = Izdvoj-Minimalnu-Granu(Q)
 for each v in G.Susjedi(u):
 if v in Q and w(u, v) < v.ključ:
 v.pred = u
 v.ključ = w(u, v)
```

### Primer Prim



Internal

**Slika 54.** Primjer Primovog algoritma

### Algoritam svih najkraćih putanja.

Posmatrajmo **usmjeren težinski** graf  $G = (V, E)$ . **Najkraći put između svih čvorova u grafu** može se odrediti ponavljanjem traženja najkraćeg puta od zadatog izvornog čvora do svih ostalih čvorova u grafu (i tako za svaki čvor).

Kako ponavljamo algoritam za svaki čvor, dodajemo mu novi stepen složenosti za  $|V|$ , odnosno:

- Za **Bellman-Ford algoritam**, dobijamo složenost  $O(n^2 m)$  (ako je graf *gust*, odnosno  $|E| \approx |V|^2$ , složenost raste i na  $O(n^4)$ ).
- Za **Dijkstra algoritam**, dobijamo složenost  $O(n^3)$ .

Očigledno se čini da je **Dijkstra algoritam** optimalno rješenje, no ima problem sa **negativnim težinama**. Ovakav problem rješavamo **Floyd-Warshallovim algoritmom**.

## Floyd-Warshall algoritam.

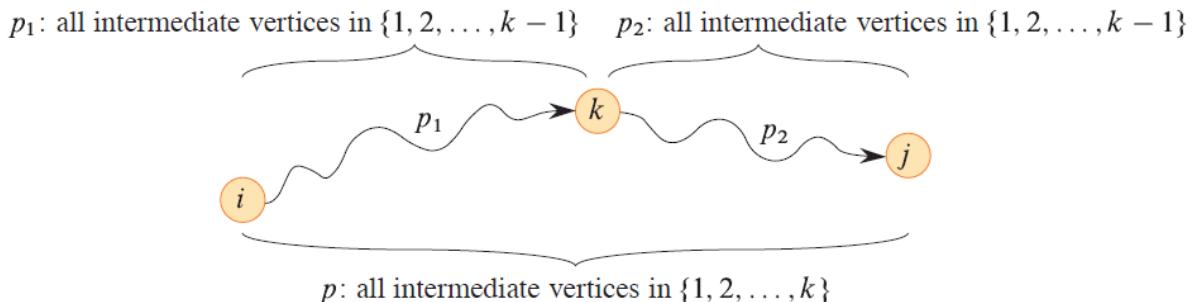
**Floyd-Warshall algoritam** jeste algoritam koji pronalazi najkraći put između svih čvorova u grafu. Koristi dinamičko programiranje i iterativno popravlja rastojanje između svaka dva čvora sve dok procjena ne bude optimalna.

Algoritam se objašnjava na sledeći način - **uzmimo u razmatranje** dva čvora  $i$  i  $j$  i posmatrajmo sve putanje koje vode od  $i$  ka  $j$  čiji se međučvorovi nalaze u skupu  $\{1, 2, \dots, k - 1\}$ . Ako želimo razmotriti dodavanje novog čvora  $k$ , moramo provjeriti sledeće:

- Ako ne postoji direktna putanja između  $i$  i  $j$  koja prolazi kroz  $k$ , onda **najkraća putanja ostaje nepromjenjena**.
- Ako postoji putanja između  $i$  i  $j$  koja prolazi kroz  $k$  i ako je ta putanja **kraća** od **najkraće putanje** iz skupa  $\{1, 2, \dots, k - 1\}$ , možemo dodati čvor u skup i proglašiti **novu najkraću putanju**. Ovdje se ogleda upotreba **dinamičkog programiranja** - najkraća putanja od  $i$  do  $j$  preko  $k$  se traži **rastavljanjem problema na traženje najkraće putanje** od  $i$  do  $k$ , kao i od  $k$  ka  $j$ , razmatrajući prethodnih  $\{1, 2, \dots, k - 1\}$  čvorova.

Na osnovu rečenog, ako krenemo sa dodavanjem čvora po čvor kao međučvorove između svaka dva čvora u grafu, formiramo **rekurzivno računanje rastojanja sa međučvorovima**:

$$d_{i,j}^{(k)} = \min \{d_{i,j'}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\}$$



**Slika 55.** Ilustracija slučaja kada postoji putanja između  $i$  i  $j$  kroz  $k$

Složenost algoritma je  $O(n^3)$ . Pseudokod algoritma:

```
for i in G.V:
 for j in G.V:
 D[i][j] = ∞

for (i, j) in G.E:
 D[i][j] = w(i, j)

for k = 1 to G.|V|:
 for i = 1 to G.|V|:
 for j = 1 to G.|V|:
 D[i][j] = min(D[i][j], D[i][k] + D[k][j])
 # Nove težine jednake minimumu starih i zbiru
```

## C Mašinsko učenje

**Mašinsko učenje.** Definicija i osnovna podela.

**Mašinsko učenje** je podoblast veštačke inteligencije koja se bavi kreiranjem računarskih sistema koji su sposobni da uče na osnovu iskustva.

Komponente **mašinskog učenja** su **model**, **funkcija cilja za procenu dobrog modela** te **metoda optimizacije za učenje modela** koji optimizuje funkciju cilja (recimo, gradijentni algoritam).

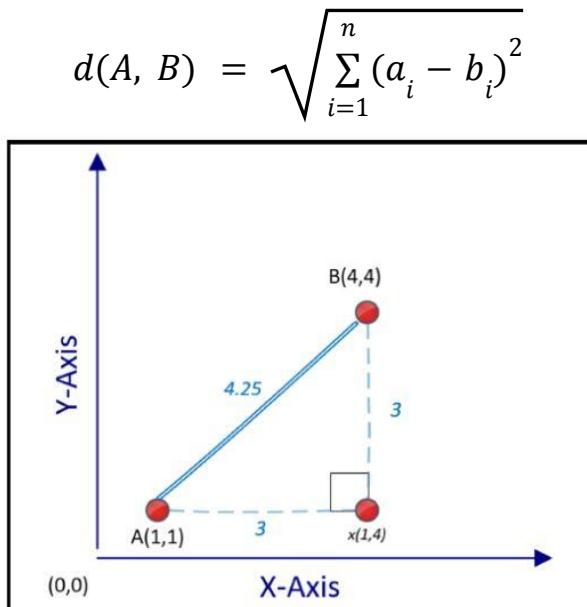
Podela mašinskog učenja:

1. **Nadgledano učenje** - kreiranje modela na osnovu skupa podataka koji se sastoji i od ulaza i od željenih izlaza. Cilj jeste **izvođenje pravila koje predviđa vrednost asociranu sa novim vektorom svojstava**.
  - Problemi **regresije** (cena kuće u zavisnosti od kvadrature) i **klasifikacije** (da li se na slici nalazi mačka, pas,...).
  - Linearna regresija, logistička regresija, stabla odluke, neuronske mreže, *K-nearest neighbors* algoritam, *Support Vector Machines*,...
2. **Nenadgledano učenje** - pronalaženje **struktura, grupa, klastera** i sl. u skupu podataka koji ne sadrži obeležene izlaze.
  - Problemi **klasterizacije** (grupisanje podataka).
  - *K-means* algoritam, *Support Vector Machines*,...

3. **Polunadgledano učenje** - kombinacija prethodna dva. Definiše se očekivani izlaz za mali deo podataka, pa se vrši „**pseudo-labeliranje**“ preostalih podataka.
4. **Učenje sa podsticajem** - obučavanje **softverskih agenata** da izvršavaju određeni zadatak po principu **nagrađivanja i kažnjavanja**.

### Merenje rastojanja između podataka.

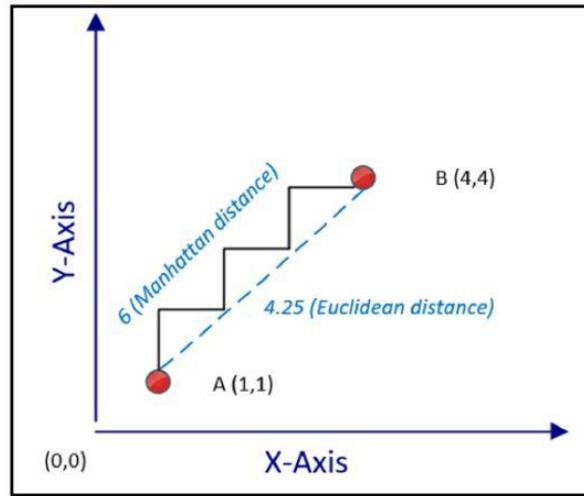
- **Euklidsko rastojanje** – Pitagorina teorema



**Slika 56. Euklidsko rastojanje**

- **Menhetn rastojanje** – Suma apsolutnih vrednosti razlika ulaza u vektorima

$$\text{Manhattan} = \sum_{i=1}^n |x_i - y_i|$$



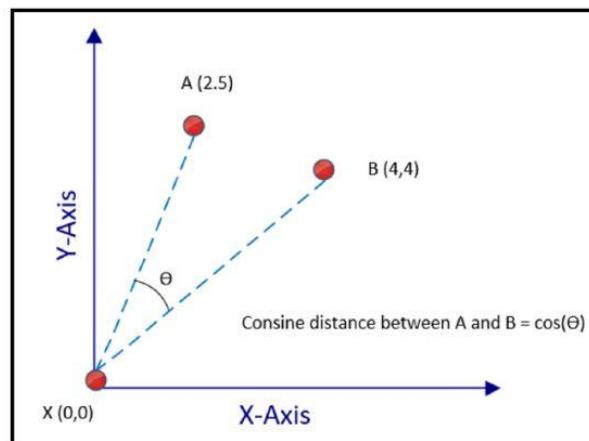
**Slika 57.** Menhetn rastojanje

- **Čebišovo rastojanje** – maksimum apsolutnih vrednosti razlika između svih elemenata vektora

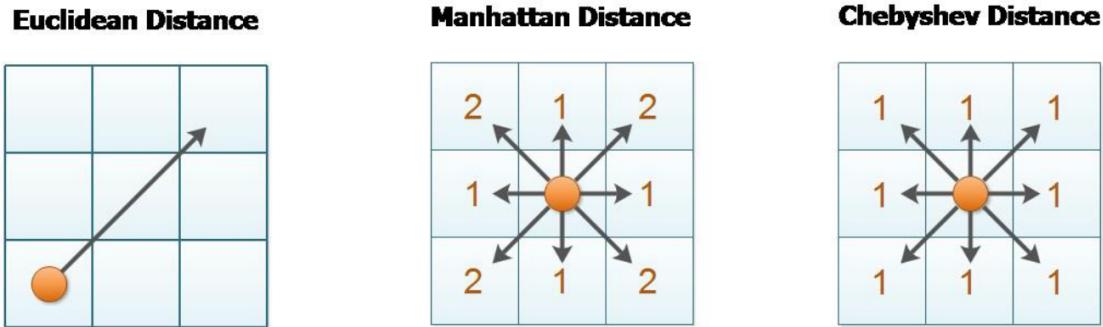
$$\text{Chebyshev} = \max(|x_i - y_i|)$$

- **Sličnost kosinusa** – vektori usmereni u istom smeru su slični (određuje se kosinus između dva vektora)

$$\text{Similarity}(a, b) = \frac{a \cdot b}{\|a\| \cdot \|b\|}$$



**Slika 58.** Sličnost kosinusa



$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \quad |x_1 - x_2| + |y_1 - y_2| \quad \max(|x_1 - x_2|, |y_1 - y_2|)$$

**Slika 59. Metrike**

## Skaliranje podataka.

**Skaliranje** predstavlja prvi korak u cilju pravilnog poređena svojstava kod merenja rastojanja između tačaka (recimo, ako klasifikujemo ljudi prema plati i broju članova njihove porodice, vrijednosti plate mogu ići i preko 30000, dok broj članova porodice teško prelazi 6).

- **Jedan od načina** jeste da se po svakom svojstvu u svim podacima odrede srednja vrednost i standardna devijacija i da se skaliraju vrednosti tog svojstva tako da imaju srednju vrednost 0 i standardnu devijaciju od 1.
- **Drugi način** je da je minimalna vrednost 0, a najviša 1.
- 

Cilj je, dakle, **sva svojstva postaviti na približno istu skalu**.

### Skaliranje svojstava - normalizacija

- Svojstvo  $x_i$  zamjenjuje se sa  $x_i - \mu_i$ .
- **Primer:**

| Površina (m <sup>2</sup> ) | Broj spavačih soba | Spratnost | Starost kuće (god) | Cena (€1000) |
|----------------------------|--------------------|-----------|--------------------|--------------|
| 250                        | 5                  | 1         | 15                 | 460          |
| 145                        | 3                  | 2         | 20                 | 232          |
| 162                        | 4                  | 2         | 10                 | 315          |
| 93                         | 2                  | 1         | 32                 | 178          |

$$x_1(1) = \frac{x_1(1) - \text{mean}}{\text{max-min}} = \frac{x_1(1) - 164}{800 - 50} = 0.1446 \quad x_1 = \frac{x_1}{\text{max}(x_1)} \quad x_1 \in [0,1]$$

## **K-means klasterizacija. Izbor optimalnog $k$ .**

**K-means klasterizacija** predstavlja tehniku **nenadgledanog učenja**.

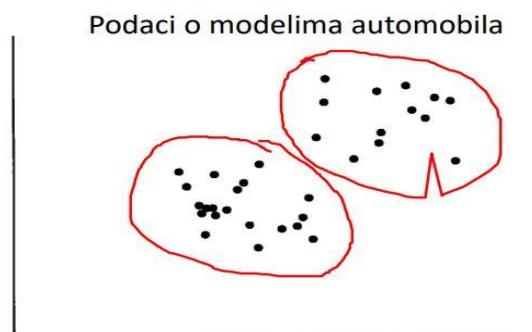
Grupisanje neobeleženih podataka na osnovu sličnosti njihovih osobina. Cilj je "pametno" grupisati tačke.

**Primeri:**

- Klasifikacija kupaca prema istoriji kupovine - svaka karakteristika može biti trošak za različite vrste robe.
- Optimalan raspored parkirališta u gradu.
- Optimizacija veličine vrata i dužine ruku košulja.
- Grupisanje sličnih slika bez prethodnog klasifikovanja.

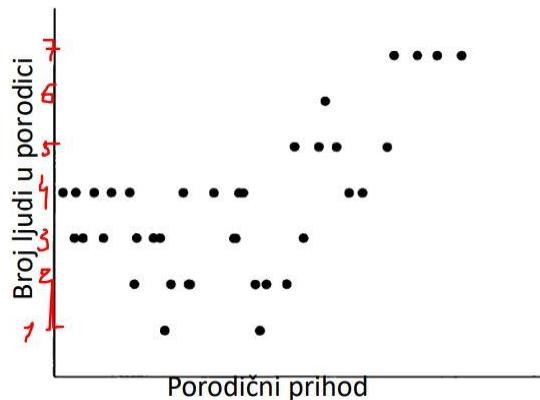
**Vrste problema klasifikacije:**

- **Traženje strukture unutar skupova neklasifikovanih podataka, sa pretpostavkom o kategorijama**
  - Podaci o modelima automobila - cena, efikasnost, cena goriva, veličina točkova,...



**Slika 60. Primjer klasifikacije - automobil**

- **Veštačko deljenje podataka čak i ako ne postoji očigledno grupisanje**
  - Proizvođač escajga želi da pakuje pribor za jelo - kriterijum je broj viljušaka i noževa, "otmenost" (cena) pribora.



**Slika 61.** Primjer klasifikacije - escajg

Postupak je sledeći:

- Izabrati  $K$  – **broj klastera**, kao i **obučavajući skup** -  $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$
- Kao izlaz ćemo dobiti **grupisane (markirane, labelirane) tačke**  $\{c^{(1)}, c^{(2)}, \dots, c^{(m)}\}$

#### **K-means algoritam:**

- **Korak 0:** Skaliranje podataka (veoma važno).
- **Korak 1:** Slučajan izbor  $K$  težišta  $\mu_1, \mu_2, \dots, \mu_k$ .
- **Korak 2:** Odrediti udaljenost  $D(i, k)$  tačaka (vektora  $x^{(i)}$ ) do težišta svakog klastera  $k$ .

$$D^{(i, k)} = \|x^{(i)} - \mu_k\| = \sqrt{\sum_{j=1}^n (x_j^{(i)} - \mu_{j(k)})^2}$$

**Svaka tačka se pridružuje najbližem klasteru.**

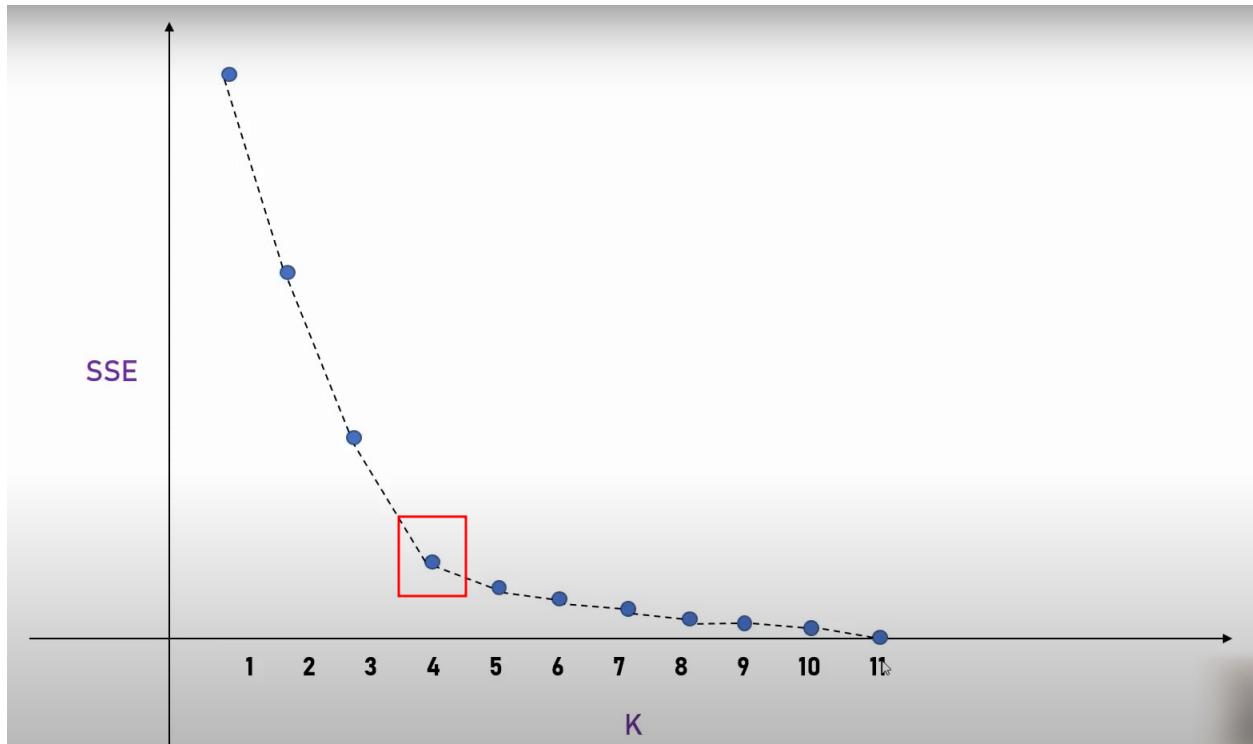
- **Korak 3:** Pronaći novih  $K$  težišta za formirane klastera (traženje centra mase novonastalih klastera).
- **Korak 4:** Povratak na **korak 2** i ponavlja se postupak sve do konvergencije težišta (težišta konvergiraju onda kada ne postoji promjena klastera elemenata obučavajućeg skupa).

#### **Inicijalizacija – slučajan izbor centroida**

- Izbor  $K$  težišta na slučajan način ( $K < m$ , gdje je  $m$  broj elemenata obučavajućeg skupa).
- Težišta su iz skupa  $X$  i slučajno izabrane tačke iz  $R^n$ .

Problem je **kako izabrati optimalno  $k$  - metoda laka**.

- Sprovede se  **$K$ -means algoritam** na nekoliko izabralih  $k$ , te se računa suma srednjih kvadrata greške između elemenata i centroida klastera.
- Bira se ono  $k$  koje se nalazi u **laktu krive**.



*Slika 62. Izbor optimalnog  $k$  metodom laka*

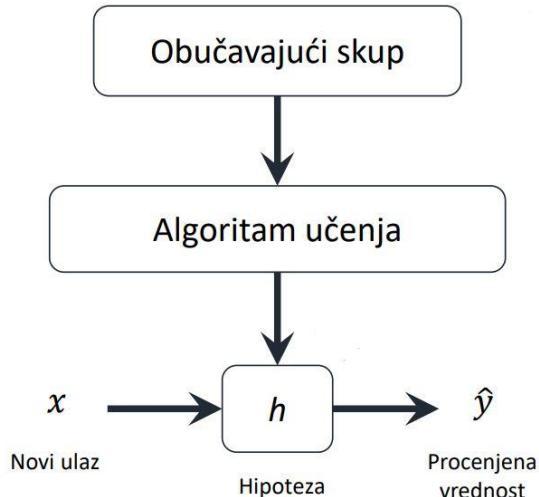
## Linearna regresija.

Model **nadgledanog učenja** - zadati su ulazi i za njih dobijeni izlazi.

Modelovanje relacije između **izlaza** i **ulaza**, na način da izlaz **linearno zavisi** od nepoznatih parametara. Cilj je dobiti pravu (krivu) koja će sa "najmanjom greškom" dobro opisati skup podataka.

Za jednostavniji slučaj sa jednom promenljivom (ulazom):

$$\hat{y} = h_{\theta}(x) = \theta_0 + \theta_1 x$$



**Slika 63.** Algoritam linearne regresije

Definiše se **optimizacioni kriterijum (metoda najmanjih kvadrata)** kako bi se što preciznije ulazi "slikali" na izlaze.

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2 = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

Problem je **kako naći (sub)optimalne parametre  $\theta$  da se dobije minimalna greška  $J$**  - za minimizaciju greške se može koristiti **gradijentni postupak ili normalna jednačina**.

Potrebno je pažljivo birati broj parametara  $\theta$ :

- Premali broj dovodi do problema **underfitting-a** - kada prava (kriva) nije dovoljno dobra da opiše skup podataka.
- Preveliki broj dovodi do problema **overfitting-a** - kada hipoteza odlično opisuje obučavajući skup, dok njene performanse padaju na testnom skupu podataka.

### Prednosti:

- Najjednostavniji mogući model.
- Model jednostavan za interpretaciju.
- Brzo računanje.

### Mane:

- Uglavnom previše jednostavna.
- Podložna overfitting-u.
- Osetljiva na anomalije.

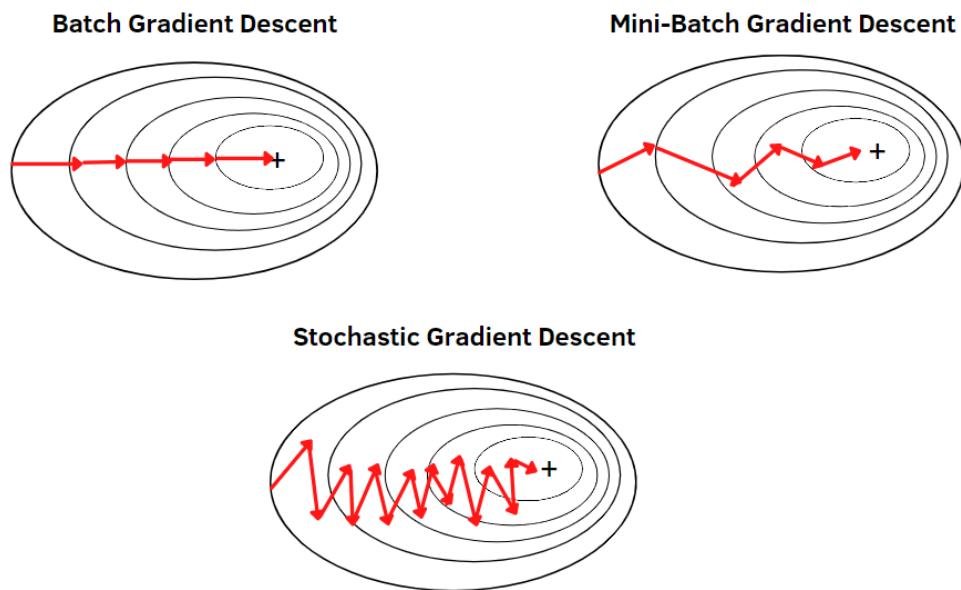
## Vrste gradijentnog algoritma.

### Paketni ili „šaržni“ (Batch Gradient Descent)

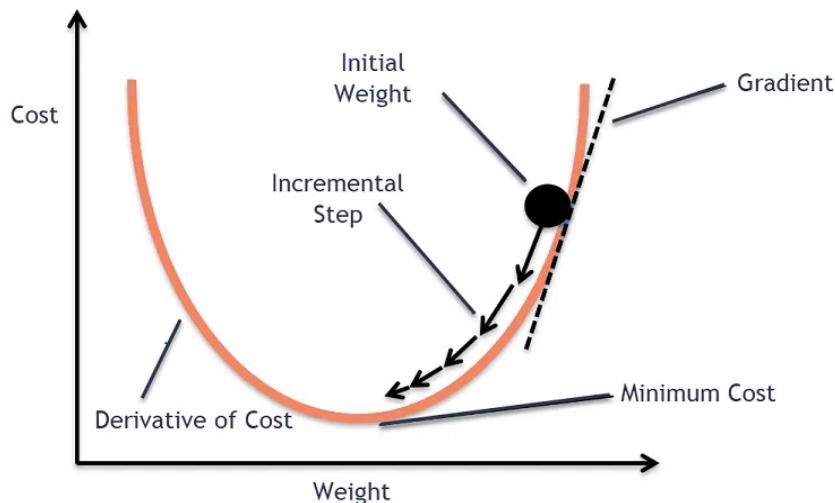
- Svi podaci se uzimaju odjednom u obzir.
- **Nedostatak:** Za veliku količinu podataka svi se moraju učitati iz baze i sporije dolazi do rešenja.

### Stohastički (Stochastic Gradient Descent)

- Izvršava se u svakom koraku za npr. jedan slučajan podatak.
- Za veliko  $m$  je **brži** od paketnog gradijentnog algoritma.



**Slika 64.** Vrste gradijentnih algoritama



**Slika 65.** Korak kod gradijentnih algoritama

Prednosti gradijentnih postukapa jesu što dobro rade i za **veliko n**.

Nedostaci su:

- Izbor  $\alpha$ .
- **Iterativni postupak** (često je potreban veliki broj iteracija).

### Lokalno ponderisana linearna regresija.

Radi na principu toga da su **bliže tačke** oko izabranog mesta *fit*-ovanja **bitnije** u odnosu na **dalje tačke** - prilagođavamo se samo dijelu podataka (*lokalno*) a ne čitavom skupu.

Fituje se  $\theta$  za minimizaciju:

$$\sum_{i=1}^m w^{(i)} (y^{(i)} - \theta^T x^{(i)})^2$$

gde je  $w^{(i)}$  težinska funkcija (**Gausova kriva**)

$$w^{(i)} = \exp\left(-\frac{(x^{(i)} - x)^2}{2}\right)$$

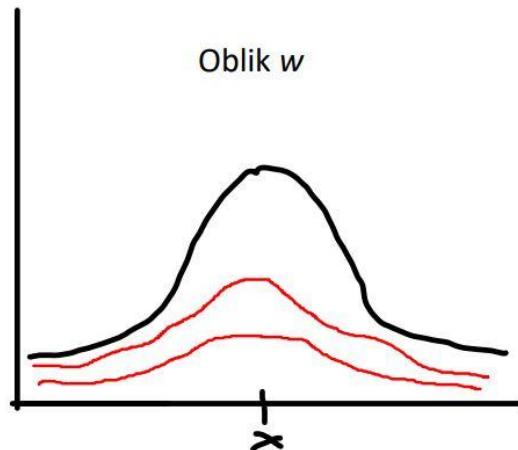
Ako je  $|x^{(i)} - x|$  **veliko**, očigledno je da  $w^{(i)}$  blizu 0, te **dalje tačke nemaju uticaj**.

Suprotno,  $w^{(i)}$  je blizu 1, te **bliže tačke imaju značajniji uticaj**.

Za ponderisanje toga u kolikom području počinjemo smatrati tačke **bitnijim**, koristi se parametar  $\tau$

$$w^{(i)} = \exp\left(-\frac{(x^{(i)} - x)^2}{2\tau^2}\right)$$

Za **veće**  $\tau$ , w je **manje** – spušta se i podiže Gausova kriva.



*Slika 66. Gausove krive za različite parametre  $\tau$*

## Logistička regresija. Postupak određivanja parametara.

Algoritam **nadgledanog učenja**, koristi se za **rešavanje klasifikacionih problema**. Najčešće se koristi za **binarnu klasifikaciju**, tj. za odgovaranje na „da – ne“ pitanja.

### Prednosti:

- Model jednostavan za interpretaciju
- Brzo računanje
- Daje informaciju o značaju svakog atributa

### Mane:

- Nekada previše jednostavna
- Podložna overfitting-u
- Zahteva linearne odnose, zbog čega se retko koristi u stvarnim problemima

**Postupak određivanja parametara** - prepostavlja se probabilistički model

$$\begin{aligned}
P(y = 1 | \mathbf{x}; \boldsymbol{\theta}) &= h_{\boldsymbol{\theta}}(\mathbf{x}) \\
P(y = 0 | \mathbf{x}; \boldsymbol{\theta}) &= 1 - h_{\boldsymbol{\theta}}(\mathbf{x}) \\
p(y | \mathbf{x}; \boldsymbol{\theta}) &= (h_{\boldsymbol{\theta}}(\mathbf{x}))^y (1 - h_{\boldsymbol{\theta}}(\mathbf{x}))^{1-y}, \quad y \in [0,1]
\end{aligned}$$

Gde je  $h$  dato kao:

$$h_{\boldsymbol{\theta}}(\mathbf{x}) = g(\boldsymbol{\theta}^T \mathbf{x}) = \frac{1}{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}}}$$

Primenjuje se **princip maksimalne verodostojnosti** za ceo obučavajući skup

$$\mathcal{L}(\boldsymbol{\theta}) = p(\mathbf{y} | \mathbf{X}; \boldsymbol{\theta}) = \prod_{i=1}^m p(y^{(i)} | \mathbf{x}^{(i)}; \boldsymbol{\theta}) = \prod_{i=1}^m (h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}))^{y^{(i)}} (1 - h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}))^{1-y^{(i)}}$$

Primenom **paketnog gradijentnog postupka** dobije se da je:

$$\boldsymbol{\theta}_j = \boldsymbol{\theta}_j + \alpha \sum_{i=1}^m (y^{(i)} - h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})) \mathbf{x}_j^{(i)} \quad i = 1, \dots, n$$

Jedna od numeričkih metoda za određivanje parametara je i Njutn-Rapsonov postupak  
- u tekućoj tački  $x_k$  se odredi tangenta na krivu  $f$  i nova tačka  $x_{k+1}$  se nalazi na preseku  
tangente i  $x$  ose.

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

Za razmatrani problem:

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \frac{\ell'(\boldsymbol{\theta}^{(k)})}{\ell''(\boldsymbol{\theta}^{(k)})}, \quad k = 0, 1, \dots$$

Generalizacija Njutn-Rapsonovog metoda za višedimenzionalni problem:

$$\frac{\partial \ell(\theta)}{\partial \theta_j} = \sum_{i=1}^m \left( y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)} = 0$$

## $K$ najbližih suseda. Izbor optimalnog $k$ .

Algoritam **nadgledanog učenja**, najčešće se koristi za klasifikaciju.

Zasniva se na pretpostavci da će podaci koji pripadaju istoj klasi biti slični i smešteni jedan blizu drugog u nekom prostoru.

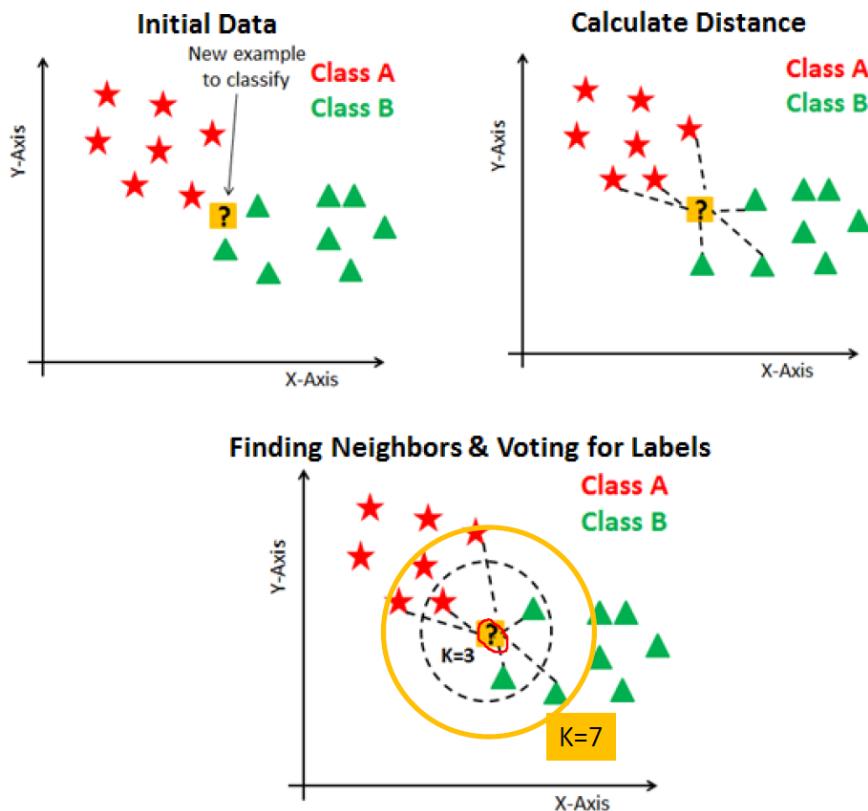
Postupak je sledeći:

- **Korak 1: Skaliranje**

- Prilagođavanje atributa da budu uporedivi. Npr. U *email* poruci skalirati da prosečan broj reči bude 0 sa standardnom devijacijom 1.

- **Korak 2: Nova tačka**

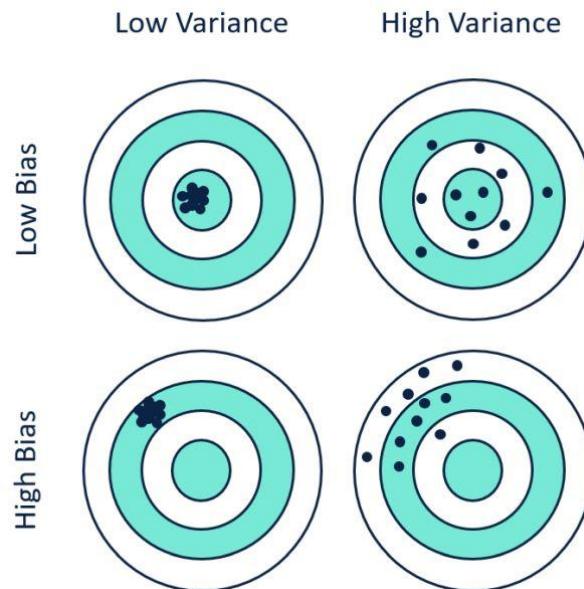
- Meri se udaljenost nove tačke do svih ostalih tačaka. Posmatra se  $K$  tačaka sa najmanjim rastojanjem



*Slika 67. KNN algoritam*

Potrebno je biti pažljiv pri biranju toga koliko susjeda uzimamo u obzir (koliko iznosi parametar  $K$ ).

- **Malo  $K$ :** mala greška, ali velika varijansa.
- **Veliko  $K$ :** velika greška, mala varijansa.

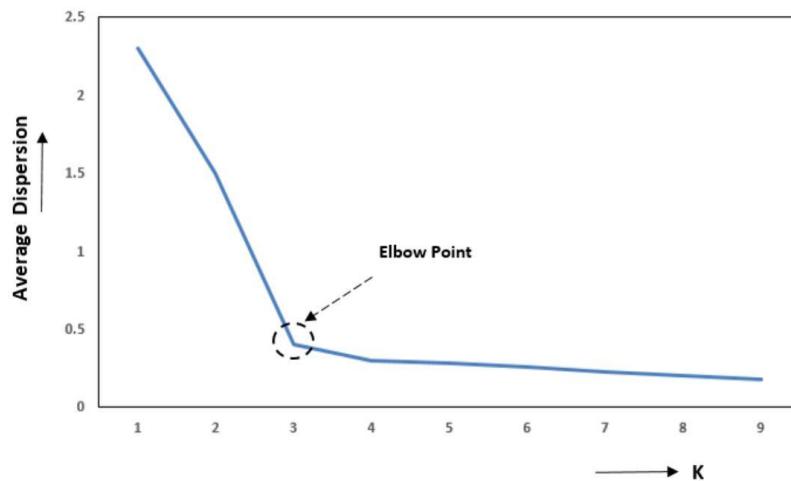


*Slika 68. Vizuelizacija greške i varijanse*

Kod **binarne klasifikacije** ("da-ne" pitanja),  $K$  mora biti neparni broj. Nekada se  $K$  postavlja da bude kvadratni koren broja uzoraka u trening skupu.

**Metoda lakta** jeste najčešća tehnika za odabir  $K$ :

- Sprovede se **KNN algoritam**.
- Bira se ono  $K$  koje se nalazi u **laktu krive**.



*Slika 69. Izbor optimalnog  $K$  metodom lakti*

**Prednosti** algoritma su:

- Lak za razumevanje.
- Lako se prilagođava podacima.
- Lak za kreiranje (malo hiperparametara).

**Mane:**

- Teško odrediti (sub)optimalno  $K$ .
- **Lenji algoritam** - algoritam kod kojih se model generiše samo dodavanjem novih tačaka.
- Neravnomerni podaci (mnogo više podataka jedne klase, npr. trouglova nego krugova) - **nebalansiran skup podataka**.

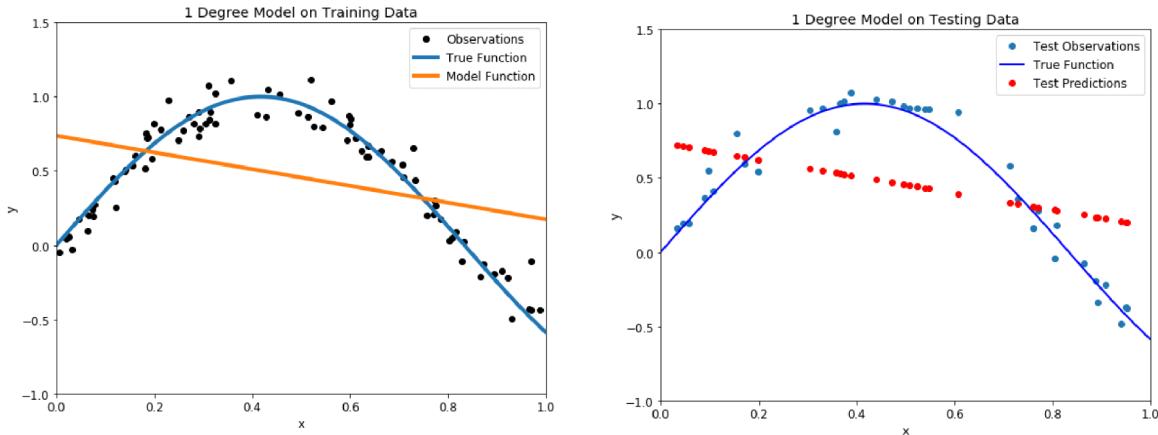
### Izbor modela. Unakrsna validacija. Vrste.

Neka je model predstavljen u obliku:

$$h_{\theta}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_k x^k$$

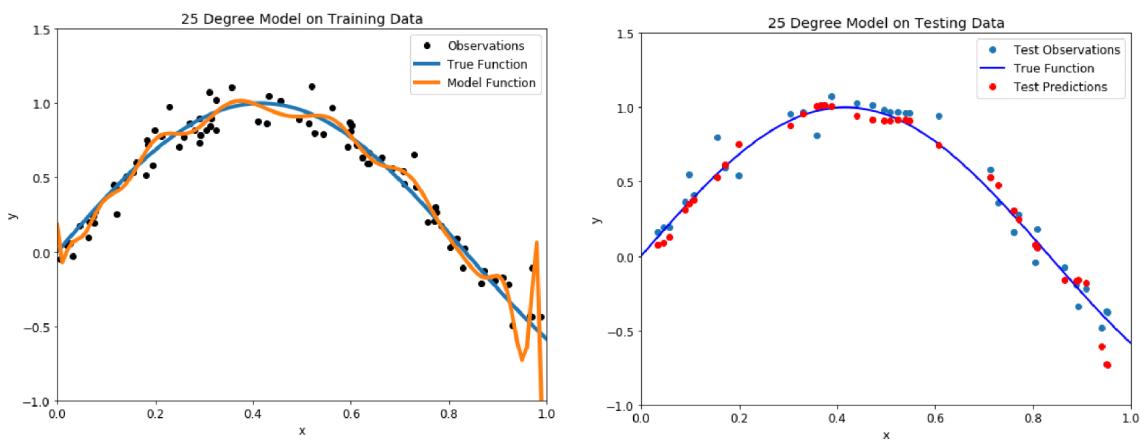
Potrebno je odrediti  $k \in \{1, 2, 3, \dots, d\}$  za koje će model najbolje opisati dati skup podataka, a da ne dođe do problema **underfit**-ovanja, kao ni **overfit**-ovanja. Za svaki broj  $k$  dobijamo više modela te je potrebno ustanoviti način poređenja među njima - **zadatak je odabrati model iz skupa modela**  $M = \{M_1, M_2, M_3, \dots, M_d\}$ .

## Modelovanje polinomom 1. reda



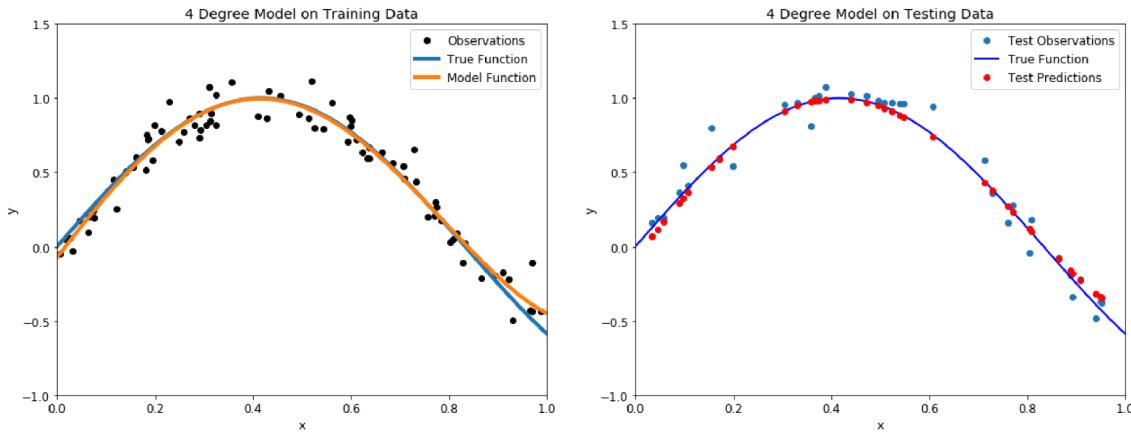
**Slika 70.** Modelovanje hipotezom prvog reda

## Modelovanje polinomom 25. reda

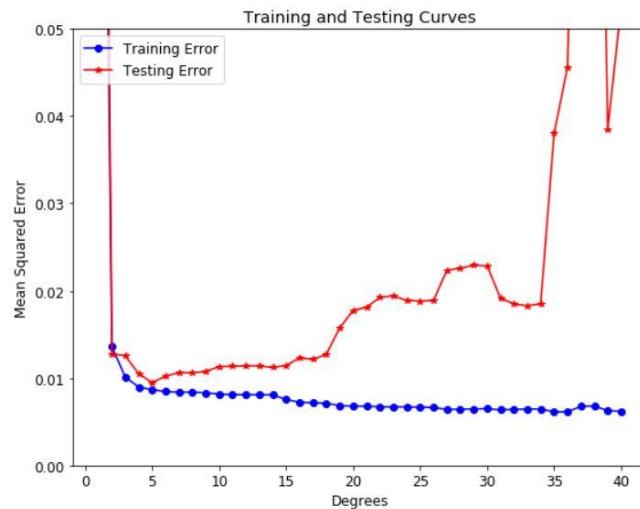


**Slika 71.** Modelovanje hipotezom 25. reda

## Modelovanje polinomom 4 reda



**Slika 72.** Modelovanje hipotezom četvrtog reda



**Slika 73.** Analiza greške za različite modele

### Unakrsna validacija (Cross Validation)

Trenirati svaki model  $M_i$  na obučavajućem skupu  $S$ , dobijajući hipoteze  $h_i$ .

Načini izbora modela:

- Izabrati hipotezu sa najmanjom greškom - ako se izabere polinom velikog reda on će se bolje prilagoditi podacima iz obučavajućeg skupa  $S$  i tako dati

**manju obučavajuću grešku**, dok kod pojave **novih (testnih) podataka** daje veliku varijansu.

- **Algoritmima validacije:**
  - **Jednostavna unakrsna validacija**
  - **K-tostruka validacija**
  - **Validacija jednostrukog eliminacije**

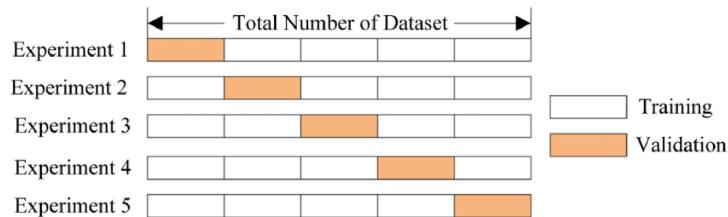
### **Jednostavna unakrsna validacija**

- Na slučajan način se podeli skup  $S$  na  $S_{train}$  (**obučavajući skup**, npr. 70% podataka) i  $S_{CV}$  (**validacioni skup**, npr. 30% podataka).
- Trenira se svaki model na skupu  $S_{train}$  i dobijaju se hipoteze  $h_i$ .
- Bira se hipoteza  $h_i$  sa najmanjom greškom  $\epsilon_{S_{CV}}(h_i)$  na validacionom skupu  $S_{CV}$ .
- **Nedostatak:** Gubitak validacionih podataka koji su mogli koristiti pri treniranju.

### **K-tostruka validacija**

- Slučajno se podeli skup  $S$  na  $k$  disjunktnih podskupova sa  $\frac{m}{k}$  primera u svakom -  $\{S_1, S_2, \dots, S_k\}$ .
- Svaki model  $M_i$  određuje se algoritmom
  - For  $j = 1, \dots, k$ 
    - trenirati model  $M_i$  na  $S_1 \cup \dots \cup S_{j-1} \cup S_{j+1} \cup \dots \cup S_k$  (sve sem  $S_j$ )
    - i dobiti hipotezu  $\hat{h}_{ij}$
    - testirati hipotezu  $\hat{h}_{ij}$  na  $S_j \rightarrow \hat{\epsilon}_{S_j}(\hat{h}_{ij})$
  - Odrediti grešku modela kao  $e_i = \frac{1}{k} \sum_{j=1}^k \hat{\epsilon}_{S_j}(\hat{h}_{ij})$
- Bira se model  $M_i$  **sa najmanjom greškom**.

## 5 –tostruka unakrsna validacija



**Slika 74. K-tostruka unakrsna validacija**

### Validacija jednostrukog eliminacije

- Ako je broj primera jako mali uzima se  $k = m$ .
- Svaki model se obučava na svakom podskupu  $m - 1$ .
- Testira se na jednom (izostavljenom) primeru.
- Uzima se prosek.

### Izbor svojstava. Algoritmi.

Ako je broj svojstava  $d$  veoma velik, često je slučaj da sva svojstva nisu bitna, odnosno da je mali skup svojstava **relevantan**. Zadatak tako postaje **redukcija dimenzionalnosti problema** - izabrati podskup "značajnijih" svojstava.

*Brute force* algoritmom postoji ukupno  $2^m$  poskupova – obimna pretraga.

Pametniji algoritmi:

- **Algoritmi omotača**
  - **Pretraga unapred**
    - Polazi se od praznog skupa svojstava  $X = \emptyset$ .
    - Svojstvo  $x_j \notin X$  sa najmanjom greškom se dodaje u skup  $X$ .
    - Ako postoji još neizabranih odlika, vratiti se na drugi korak. U suprotnom, dobili smo konačni skup  $X$ .
  - **Pretraga unazad**
    - Polazi se od skupa svih svojstava  $X$ .

- Eliminiše se u svakoj iteraciji jedno svojstvo  $x_j$  sa najvećom greškom.
- Ako postoji još neizabranih odlika, vratiti se na drugi korak. U suprotnom, dobili smo konačan skup  $X$ .
- **Rangirajući algoritmi**
  - Uzajamna informacija svojstva i izlaza - **korelacija**.

## D Lunci blokova i kriptografija

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

**Bezbjednost** u računarskim sistemima je usko povezana sa pojmom **oslonjivosti**.

- Očekujemo da se "možemo osloniti na sistem" - da sistem isporuči adekvatan traženi servis.

Pored **bezbjednosti**, oslonjivost uključuje:

- **Tajnost** - informacija se daje samo ovlašćenim (autorizovanim) licima ili servisima.
- **Integritet** - modifikacija podataka zahtjeva autorizovan pristup.

U bitnim programskim sistemima podaci i servisi se moraju **očuvati od bezbjednosnih prijetnji**. Neki od tipova **bezbjednosnih prijetnji** su:

- **Presretanje (interception)** - neautorizovana stranka dobija prisup podacima ili servisima (ilegalno korišćenje podataka).
- **Prekidanje (interruption)** - servis ili podatak postane nedostupan, neupotrebljiv ili uništen.
- **Modifikacija (modification)** - neautorizovana izmjena podataka ili servisa, koji više nije u skladu sa specifikacijom.
- **Fabrikacija (fabrication)** - generisanje dodatnih podataka ili aktivnosti koji ne bi postojali u normalnim situacijama (npr. ilegalno ponavljanje ranijeg zahtjeva za prenos novca ili dodavanje zapisa u datoteku sa lozinkama).

Mehanizmi sprovođenja bezbjednosti su:

- **Šifrovanje (encryption)** - transformiše podatke u format koji nije razumljiv za napadače.
- **Autentikacija (authentication)** - verifikacija identiteta korisnika, klijenata, servera,...
- **Autorizacija (authorization)** - provjera da li korisnik ili servis ima pravo na izvršavanje određene akcije.

- **Bilježenje istorijata aktivnosti (auditing)** - upisivanje u dnevničke događaja koji je entitet čemu pristupio i na koji način. Ovaj mehanizam ne obezbeđuje direktnu zaštitu od napadača ali omogućava naknadnu analizu bezbjednosnih problema.

**Šifrovanje** je transformacija informacije tako da je njen pravo značenje skriveno (što znači da je potrebna naknadna i ne tako lako doseziva informacija da se dekodira šifrovana informacija).

## Osnovni algoritmi kriptografije (podela, primitivni i napredni algoritmi).

Osnovni pojmovi vezani za kriptografiju su:

- **Poruka ( $P$ )** - podatak ili dio podatka koji strana  $A$  šalje strani  $B$ .
- **Šifrovana poruka ( $C$ )** - poruka koja je promijenjena 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 modificira poruku sa ciljem 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.

**Šifrovanje** je transformacija informacije tako da je njen pravo značenje skriveno (što znači da je potrebna naknadna i ne tako lako doseziva informacija da se dekodira šifrovana informacija).

Šifrovana poruka bi trebalo da **sprječi akcije uljeza** (prijetnje bezbjednosti - presretanje, modifikaciju i fabrikaciju). **Dešifrovanje** bez ključa bi trebalo da je **jako otežano**.

## Primitivni algoritmi šifrovanja

Nekoliko primjera:

- Jednostavna zamjena slova je način kodiranja teksta gdje se neko slovo zamjenjuje drugim, a dekodiranje ponovi zamjenu (ako se poruka **DANAS** rastavi na **ASCII** kodove i svakom slovu se doda **ASCII** šifra karaktera 'A', dobija se poruka **HEREW**).
- **Binarna podloga (One Time Pad algoritam):**
  - Posmatrajmo niz bita koji čine **poruku**  $P$ . Na nju primjenimo **masku** iste dužine kao što je i poruka i dobijamo **kodiranu poruku**  $C$ .
  - Maska je obično neki nasumičan niz bitova, dok je algoritam primjene "ekskluzivno ili" (**XOR**) operacija.

- Osobine algoritma su da čini kodiranu poruku bezbjednom, koristi ključ dužine poruke ali je ponovna upotreba ključa rizična jer se primjenom na presretene dvije kodirane poruke otkriva. Ključ može imati ogromne dužine, te zato nije najefikasniji algoritam.
- Primjer: **ONETIMEPAD** poruka na koju je primjenjem ključ **TBFRGFARFM** daje šifrovanu poruku **IPKLPSFHGQ**.
- **Šifrovanje pomoću ključeva** - tehniku sledećih osobina:
  - **Šifrovanje** generše poruku na osnovu originalne poruke i **tajnog ključa**,  $C = E_K(P)$ .
  - **Dešifrovanje** generiše originalnu poruku na osnovu kodirane poruke i **tajnog ključa**,  $P = D_K(C)$ .
  - Nemoguće je naći ključ  $K$  ako su poznate i kodovana i dekodovana poruka, kao i nemoguće je proizvesti ključ  $K'$  takav da  $E_K(P) = E_{K'}(P)$ .
  - Dvije vrste algoritama šifrovanja - **simetrična kriptografija** i **asimetrična kriptografija**.
    - Karakteriše ih upotreba **velikih brojeva** kao **ključeve** - fizički (brute force metodom) gotovo nemoguće pronaći ključ.

## Simetrična kriptografija

Drugo ime - **kriptografski sistemi sa dijeljenim ključem**.

Koristi **istu ključ** za šifrovanje i dešifrovanje (oznaka ključa -  $K_{A,B}$ ). Da bi komunikacija bila bezbjedna, ključ mora biti tajan.

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

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

Kod simetrične kriptografije, koriste se **kraći ključevi** i poruke se "sjeckaju" na **blokove bita**. Ti blokovi se **ulačavaju** i time se sprječava da isti blokovi daju isti šifrovan rezultat (obrada narednog bloka koristi rezultat obrade prethodnog bloka).

**AES (Advanced Encryption Standard)** je javno dostupan besplatan, bezbjedan, jednostavan, fleksibilan algoritam šifrovanja simetričnim ključem (koristi ključ od 128, 192 ili 256 bita, a blokove dužina 128 bita).

## Asimetrična kriptografija

Drugo ime - **kriptografski sistemi sa javnim ključevima**.

Dva odvojena ključa za šifrovanje i dešifrovanje. Svaki od učesnika u komunikaciji ima dva ključa - **javni ( $K_E$ )** i **privatni ( $K_D$ )** ključ.

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.

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

### RSA algoritam (principi, uključeni algoritmi).

**RSA algoritam** je algoritam šifrovanja asimetričnim ključem. Oslanja se na broj koji je proizvod dva velika prosta broja (recimo, broj sa 2048 bita je dovoljno velik).

Princip rada algoritma:

- Izabratи 2 velika različita prosta broja  $p$  i  $q$  (najmanje 1024 bita svaki).
- Izračunati  $n = p \cdot q$ .
- Izračunati  $r = (p - 1) \cdot (q - 1)$ . ( $r$  = koliko je brojeva od 1 do  $n$  koji nemaju zajedničkog delioca sa  $n$ )
- Izabratи mali broj  $e$  tako da su  $e$  i  $r$  uzajamno prosti (nemaju zajedničke djelioce, sem 1). ( $e$  je enkripcija, njime se šifruje poruka)
- Izračunati  $d$  kao inverzan elemenat za množenje, odnosno da važi  $e * d \pmod{r} = 1$ . Nadje se moduo po  $r$  od  $e * d$  ( $d$  je dekripcija, njime se primalac desifruje poruku)
- **RSA** javni ključ tada postaje par  $K_E = (e, n)$ .
- **RSA** privatni ključ tada postaje  $K_D = (d, n)$ .
- Funkcije šifrovanja i dešifrovanja su tada:

$$E_{K_E}(m) = c = m^e \pmod{n}$$

$$D_{K_D}(c) = m^d \pmod{n}$$

### Primer (sa malim brojevima)

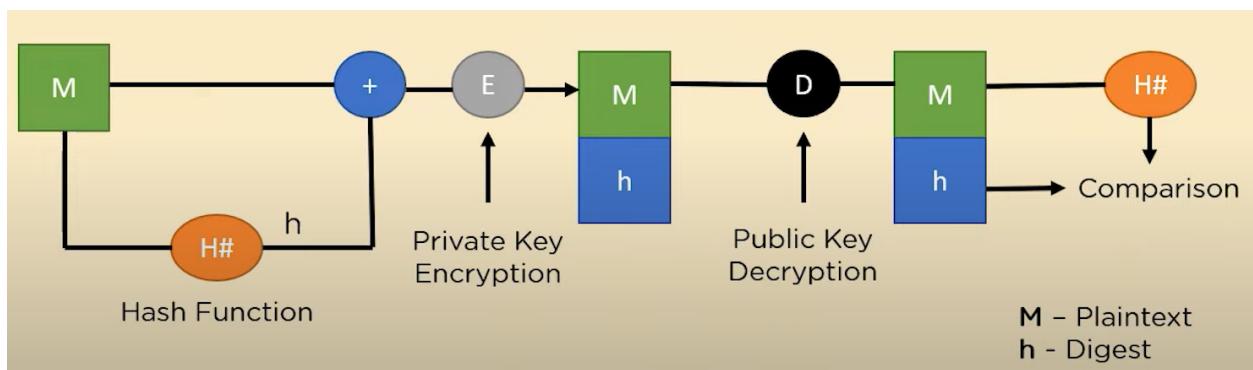
1. Izabratи:  $p = 17$  i  $q = 29$
2. Izračunati  $n = pq = 493$
3. Izračunati  $r = (p - 1)(q - 1) = 448$
4. Izabratи mali broj  $e = 5$  jer su 5 i 448 uzajamno prosti.
5. Izračunati  $d = 269$   
provera:  $ed \bmod r = (5 \cdot 269 \bmod 448) = 1345 \bmod 448 = (3 \cdot 448 + 1) \bmod 448 = 1$
6. RSA javni ključ је  $K_E = (5, 493)$
7. RSA privatni ključ је  $K_D = (269, 493)$
8. Primer šifrovanja i dešifrovanja:  
 $E_{K_E}(327) = 327^5 \bmod 493 = 3.738.856.210.407 \bmod 493 = 259$   
 $D_{K_D}(259) = 259^{269} \bmod 493 = \dots = 327$

**Slika 75.** Primjer RSA algoritma

### Algoritam za digitalne potpise. DSA algoritam.

**Digitalni potpisi** су **mehanizmi određivanja autentikacije** podataka, dokumenata i sl. Rade pomoću **asimetrične kriptografije** uz razliku da oni enkriptuju podatke **tajnim ključem**, a dekriptuju **javnim ključem**.

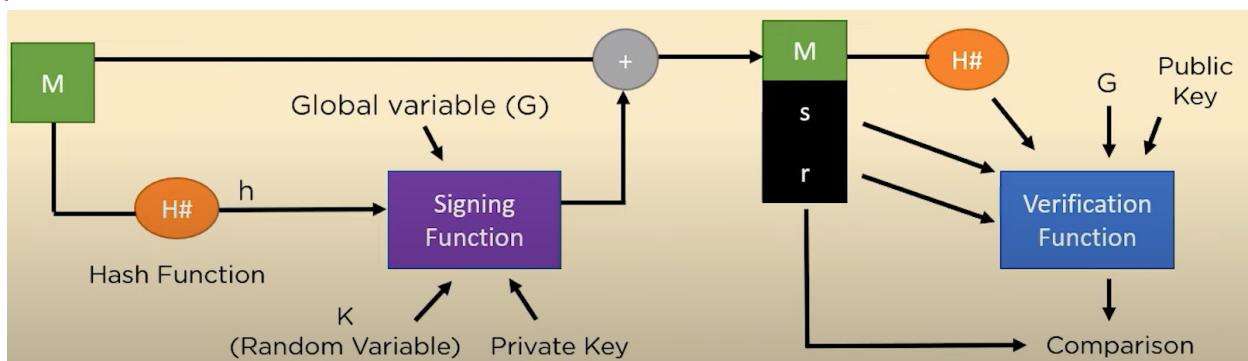
Kod **digitalnih potpisa**, originalna poruka prolazi kroz **heširanje**. Nakon heširanja, dobijeni **heš** se spaja sa originalnom porukom, te se vrši **enkripcija** koristeći **tajni ključ**. Zatim se **javnim ključem pošiljaoca** **dekriptuje** poruka koja prolazi kroz **istu heš funkciju** i upoređuju se **stari i novi heš**, te se tako verifikuje integritet poruke.



**Slika 76.** Mehanizam digitalnih potpisa

**DSA algoritam** je (pored **RSA**) asimetrični kriptografski algoritam koji se koristi za **digitalne potpise**. Dakle, ovaj algoritam ne možemo primjeniti za **enkripciju** niti **razmjenu ključeva**, no isključivo na **digitalne potpise**.

Originalna poruka je propuštena kroz **heš funkciju** (*MD5, SHA-256* funkcije), čiji rezultat je ulaz u **funkciju potpisa**. **Funkcija potpisa**, pored rezultata heširanja, takođe prima parametre u vidu **slučajnog broja** i **privatnog ključa pošiljaoca**, generišući tako **digitalni potpis**. Originalna poruka se, uz digitalni potpis koji se predstavlja kao **dva 16-bitna broja**, šalje **primaocu** koji koristi istu **heš funkciju kao pošiljalac** na dobijenu poruku. Heš ulazi u **verifikacionu funkciju** koja koristi **javni ključ** i provjerava integritet poruke.



**Slika 77. DSA algoritam**

### DSA algoritam - koraci

1. A generiše slučajan broj k manji od q
2. A generiše  
 $r = (g^k \bmod(p)) \bmod(q)$   
 $s = (k^{-1} * (H(m) + x * r)) \bmod(q)$

Parametri r i s su potpis od A i oni se šalju B

3. B verifikuje potpis tako što izračunava  
 $w = s^{-1} \bmod(q)$   
 $u_1 = (H(m) * w) \bmod(q)$   
 $u_2 = (r * w) \bmod(q)$   
 $v = ((g^{u_1} * y^{u_2}) \bmod(p)) \bmod(q)$   
Ako je  $v = r$ , tada je potpis ispravan

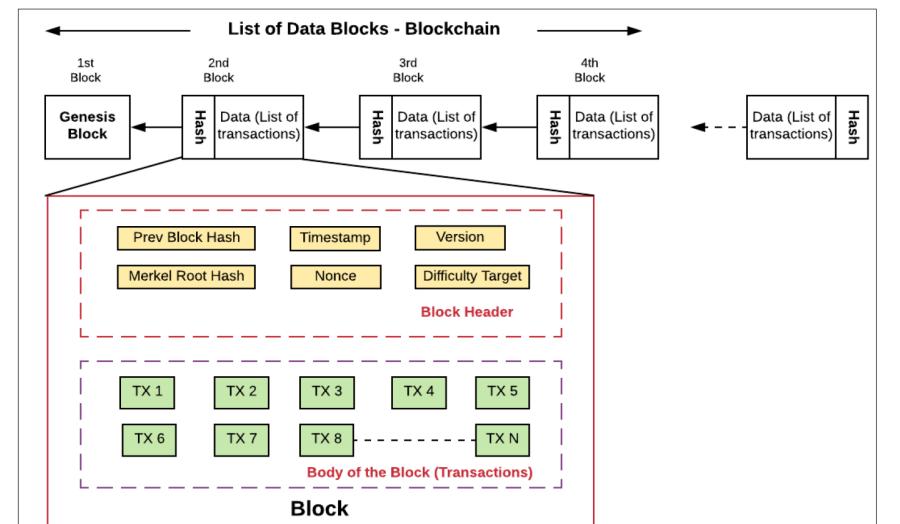
**Slika 78. Matematika iza DSA algoritma**

## Lanac blokova (*blockchain*).

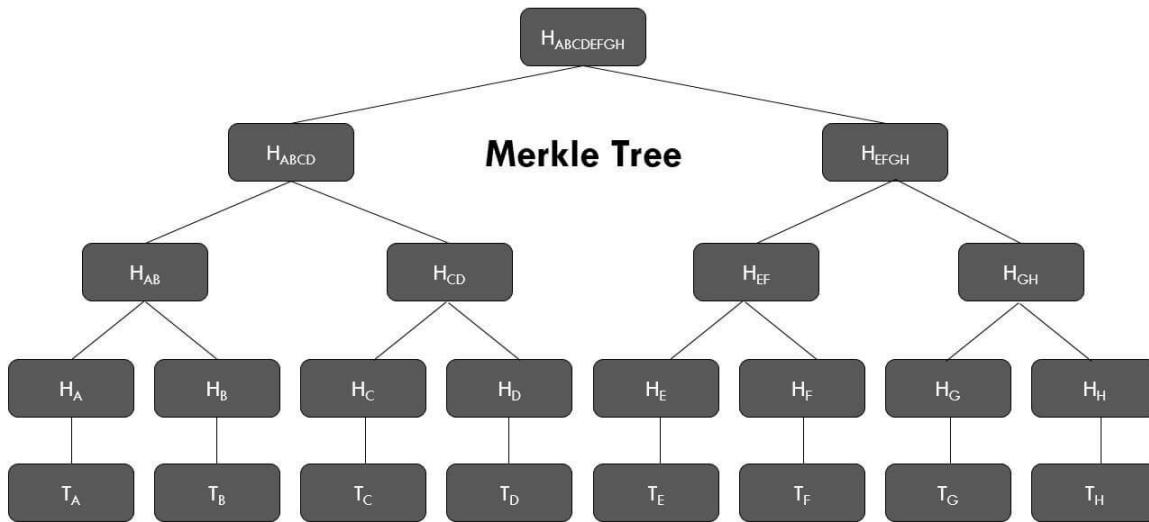
**Blockchain (lanac blokova)** predstavlja decentralizovanu, distribuiranu i javnu arhitekturu koja se sastoji od **blokova** - skupa različitih podataka koji se različito interpretiraju, u zavisnost od vrste samog **blockchain-a**.

Svaki blok, suštinski, sadrži nekoliko stvari:

- **Veličina** - veličina bloka, obično 4 bajta.
- **Broj transakcija** - broj transakcija koje su do sada smještene u blok (**blockchain arhitekture vezane za finansije**).
- **Transakcije (podatke)** - podaci o transakcijama smješteni u blok (**blockchain arhitekture vezane za finansije**).
- **Zaglavlje** - sadrži polja koja definisu blok, među kojima su:
  - **Pokazivač na prethodni blok** - sadrži **heš vrijednost** prethodnog bloka.
  - **Korenska heš vrijednost** - "otisak prsta" bloka, identifikacioni dio bloka koji je jedinstven. Promjene unutar bloka pokreću i samu promjenu **heša**, te se tako može detektovati promjena blokova. Predstavlja heš vrijednost svih transakcija u bloku koja se dobija pomoću **Merkle Tree strukture**.
  - **Težina** - parametar koji određuje koliko je vremena i električne energije utrošeno za kreiranje bloka.
  - **Brojač** - jedinstvena, nasumična vrijednost koja se može samo jednom iskoristiti za kreiranje bloka. Povećava se ukoliko generisana heš vrijednost nije validna za kreiranje novog bloka.
  - **Vrijeme** - trenutak kreiranja bloka.
  - **Verzija** - određuje validaciona pravila.



**Slika 79.** Blockchain struktura



**Slika 80.** Merkle tree

Prvi blok u **blockchain** arhitekturi naziva se **genesis blok**.

Maliciozna promjena jednog bloka ima za posljedicu da naredni blokovi nemaju validan **pokazivač na taj blok** jer se njegova **heš vrijednost** promijenila. Takođe, dolazi do promjene njihovih **heš vrijednosti**, a uz današnju brzinu računara, maliciozni korisnici su u mogućnosti ponovno izračunati te **heš vrijednosti** i tako prikriti svoj napad na arhitekturu. U tu svrhu stvaraju se **algoritmi konsenzusa**:

- **Proof of Work** - najpoznatiji algoritam konsenzusa. Radi na principu da je "rješenje teško pronaći, ali ga je lako potvrditi" - mijenjanje novog bloka (te i ostalih kasnije) zahtjeva dug vremenski period. Ovaj algoritam se koristi i u **rudarenju - rješavanju kriptografskih zadataka** od strane **rudara** kako bi se njihov blok smjestio i bio validan u čitavoj **blockchain** arhitekturi. Ovo je jako težak algoritam u smislu energetske efikasnosti jer zahtjeva jaku procesorsku moć.
- **Proof of Stake** - radi na principu ulaganja kriptovaluta u mrežu. **Validatori** su zaduženi za kreiranje blokova na način da svaki od njih ulaze u mrežu određen iznos u kriptovaluti, a onaj koji uloži najviše ima **najveću šansu** da bude **izabran** i smjesti **svoj blok** u arhitekturu. Lakši je od **Proof of Work** u smislu energetske efikasnosti.
- **Proof of Authority** - obično za odobrene knjige identiteti korisnika moraju biti poznati i verifikovani. Mogućnost objavljivanja novih blokova je diktirana korisničkim dozvolama.

Pored algoritama konsenzusa, **blockchain** koristi **Peer-To-Peer** arhitekturu, u kojoj svaki korisnik dobija kopiju **čitave blockchain arhitekture**. Ako postoji korisnik koji mijenja blok ili želi dodati svoj blok, on mora proći proces **verifikacije kod svih**

**korisnika blockchain arhitekture** i ako dobije odobrenje od svih, tek tada se može priključiti mreži.

Aplikacije **blockchain** arhitektura su raznorodne:

- Finansijske organizacije
- Osiguravajuća društva
- Zdravstvene organizacije
- Cyber bezbjednost

U **blockchain** branši koja se dotiče finansijske strane, postoje takozvani **pametni ugovori** - **nepromjenljivi** i **distribuirani** posrednici (programi) u finansijskim transakcijama između dva ili više korisnika. Zbog same sigurnosti **blockchain** tehnologije, **pametni ugovori** rješavaju problem "vjerovanja trećoj osobi (u realnom životu, bankama) da će izvršiti transakciju između osobe A i osobe B". Dakle, bez odobrenja čitave **blockchain** mreže, izvršavanje transakcije nije moguće.

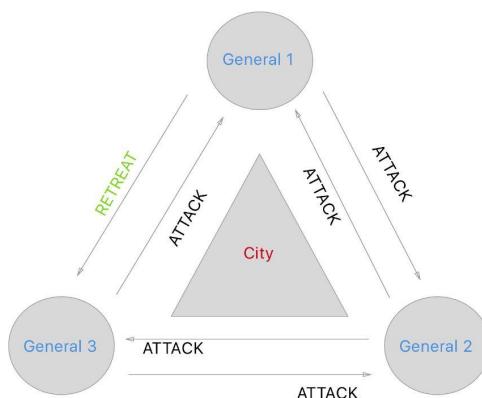
Najpopularnija **blockchain** arhitektura koja podržava **pametne ugovore** jeste **Etherium**, za čiji razvoj je razvijen poseban programski jezik - **Solidity**.

### Problem vizantijskih generala.

**Problem vizantijskih generala** je definisan još davne 1982. godine i ilustruje problematiku komunikacije preko posrednika koji nisu pouzdani.

Zamislimo da vizantijski generali opsedaju neprijateljski grad i da treba da se dogovore oko zajedničkog plana akcije. Da bi napad uspeo, potrebno je da svi napadnu u isto vreme. Ako neki od generala ne izvrši napad u dogovorenou vreme, veoma je verovatno da napad neće uspeti. Pošto se generali nalaze na različitim lokacijama oko grada, ne mogu se uživo dogovarati, već komuniciraju preko kurira.

**Prvi problem** je ako su neki od generala izdajnici. Oni će namerno sabotirati dogovor i preko svojih kurira slati informacije tako da i među poštenim generalima izazovu konfuziju.



**Slika 81.** Problem vizantijskih generala

Na ovom jednostavnom primeru smo videli da je dovoljno da je samo 1 od 6 kurira izdajnik da bi šanse za uspeh napada značajno smanjile. U većim i kompleksnijim sistemima, broj učesnika ja značajno veći, što samo dodatno komplikuje situaciju. Ovaj problem posebno je izražen u sistemima koji nisu **centralizovani** i gde je broj učesnika prevelik da bi svako sa svakim direktno komunicirao. Do nastanka decentralizovanih sistema se nije moglo sa sigurnošću znati da je jedan računar u mreži primio poruku i obradio je.

Kod **centralizovanih** sistema, gde problem komunikacije ne postoji, svi primaju informacije direktno sa vrha i do svih stižu iste informacije.

**Bitcoin** predstavlja upravo jedan takav sistem – veliki i decentralizovan.