Blok 01
Struktura podataka i algoritmi

prof. Dr Vidan Marković

Univerzitet Singidunum
2023/2024

Sadržaj

1. Cilj predmeta

2. Runtime vs development time

3. Efikasnost algoritama u run time režimu rada

Cilj predmeta

- SPA je od fundamentalnog značaja za ljude koji se bave informatičkim naukama i razvojem softvera.

- Strukture podataka i algoritmi su fundamentalni koncepti u informatici i softverskom inženjerstvu. Iako možda deluju apstraktno i složeno, oni igraju ključnu ulogu u dizajniranju efikasnih i skalabilnih softverskih sistema.

- Strukture podataka i algoritmi obezbeđuju sistematski pristup rešavanju problema u informatici. Oni omogućavaju programerima da pišu kod koji je efikasan, skalabilan i jednostavan za održavanje. Učenjem struktura podataka i algoritama možete da poboljšate svoje veštine rešavanja problema i primenite ih kako biste izgradili bolja softverska rešenja.

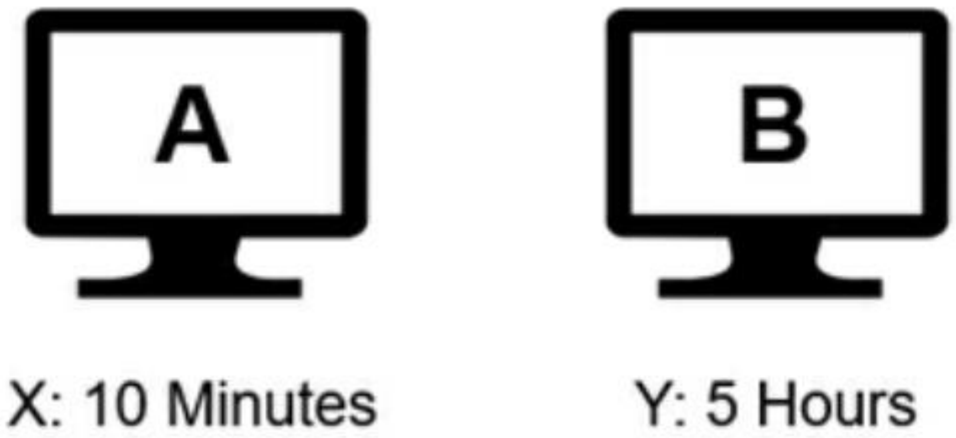
- Efikasan kôd je od kritičnog značaja za izradu softverskih aplikacija koje mogu da rukuju velikom količinom podataka i korisničkim saobraćajem. Strukture podataka i algoritmi obezbeđuju alatke za optimizaciju koda za performanse i skalabilnost. Korišćenjem ovih alatki možete da napišete kôd koji radi brže, koristi manje memorije i sveukupno je efikasniji.

Ispit

- K1 – do 15 bodova teorija + 15 bodova zadaci
- K2 – do 30 bodova zadaci
- Prisustvo i aktivnost – do 10 bodova
- Ispit – do 30 bodova teorija (Mtutor test)
- 50 + % na svakoj obavezi da bi se položila

Running Time – kompleksnost

- Šta je runtime?
- Šta prestavlja kompleksnot runtime-a?
- Da li je X algoritam efikasniji od Y algoritma?



Running Time – kompleksnost

Da bi poredili efikasnost algoritama moramo staviti sve na isti tas virtuelne vage!

- Naš računar ima samo jedan CPU
- Svi podaci su uskladišteni u RAM memoriji
- Svaki pristup memoriji zahteva potpuno isto vreme

Razlika između vremenske i runtime kompleksnosti je u tome što ne gledamo vremenske jedinice već u količinu takozvanih **primitivnih operacija**.

Running Time – primitivne operacije

- Dodeljivanje vrednosti promenljivoj
- Aritmetičke operacije
- Logičke operacije
- If-izrazi i poređenja
- itd...

U suštini sve što iza sebe nema kompleksan algoritam može se smatrati primitivnim.

Running Time – primitivne operacije

```
list := [list of random numbers]
sort list using merge sort
for each element in list:
    print(element)
```

- U ovom pseudokod primeru inicijalizovanje random brojeva nije primitivna operacija, merge sort nije primitivna operacija, ali print možemo poslatrati kao primitivnu.

Running Time – kalkulacije

```
list = [list of n numbers]
a := 0
for each element in list:
    a = a + 1
    print(a)
```

```
list = [list of n numbers] (1 PO)
a := 0 (1 PO)
for each element in list: (n times)
    a = a + 1 (1 PO)
    print(a) (1 PO)
```

- Da probamo jednostavniji primer
- Rezultat je: $2 + 2n$

Running Time – kalkulacije

```
list = [list of n numbers] (1 PO)
a := 0 (1 PO)
for each element in list: (n times)
    if element is even: (1 PO)
        a = a + 1 (1 PO)
        print(a) (1 PO)
```

- Da probamo da malo iskomplikujemo primer. Šta se promenilo? Da li je sada rezultat uvek 2 + 2n? Zašto?

- Zato moramo pogledati tri različite kompleksnosti izvršavanja: najbolji slučaj, prosečan slučaj i kompleksnost najgoreg slučaja.

Running Time – kalkulacije

- Zato moramo pogledati tri različite kompleksnosti izvršavanja: najbolji slučaj, prosečan slučaj i kompleksnost najgoreg slučaja.

- Najbolji (svi neparni): 2 + n (if statement mora da se protrči n puta)

- Najgori: 2 + 3n (svi parni)

- Srednji: $2 + 3\frac{n}{2} + \frac{n}{2} = 2 + \frac{4n}{2} = 2 + 2n$

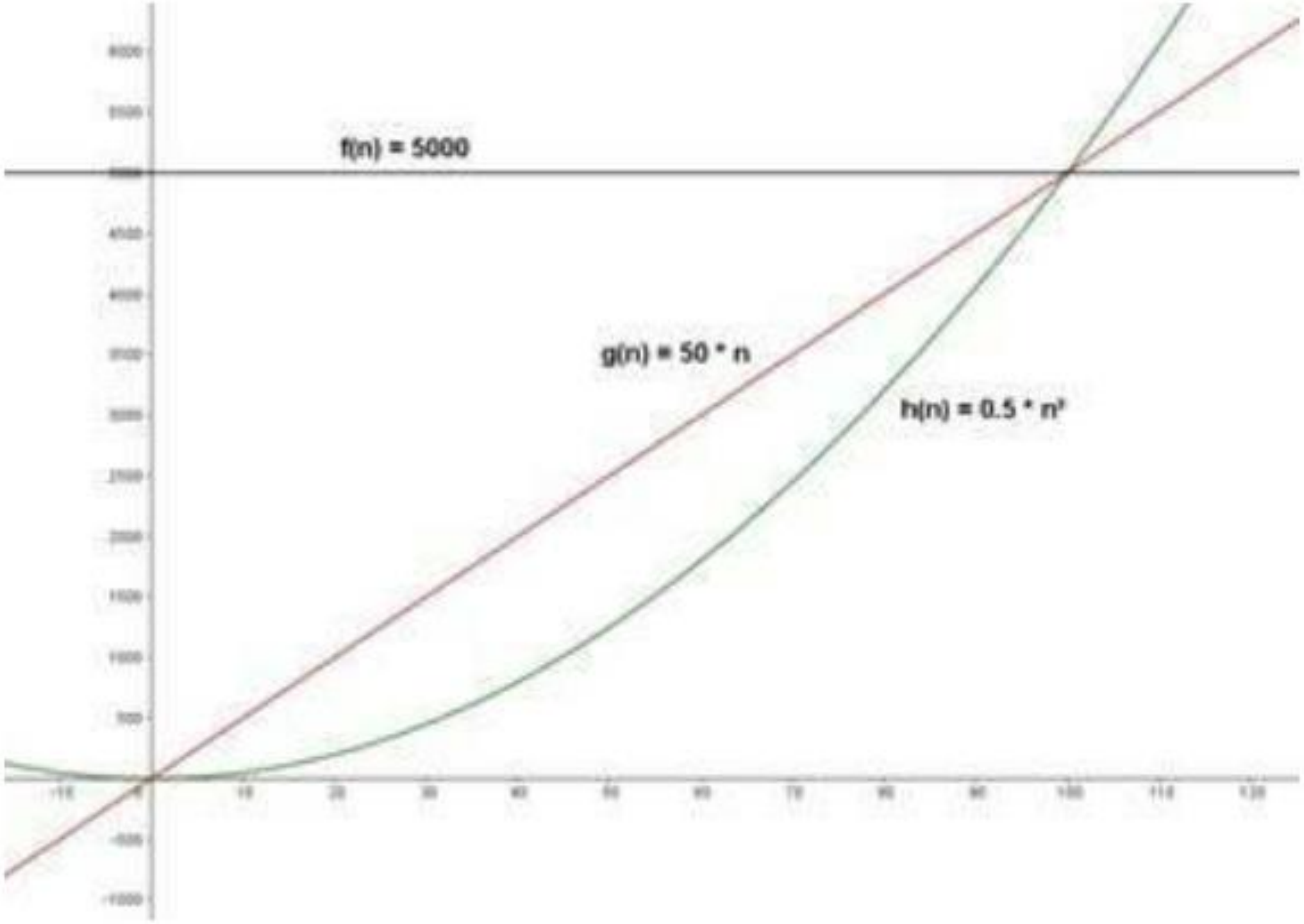
Running Time – asimptotski rast funkcija

- Nije nas briga za male slučajeve problema
- Nije nas briga za konsante u proračunima
- Nije nas briga za stalne faktore

Ukratko: naš fokus je isključivo na tome koliko je vreme izvršenja povećano, kada dodamo nove elemente problemu. Ne zanimaju nas tačne vrednosti. **Zainteresovani smo za vrstu rasta.**

Running Time – asimptotski rast funkcija

Šta nam je najbolja opcija?



Running Time – Big O

- Kada govorimo o složenosti algoritama u matematičkom kontekstu, gotovo uvek ćemo koristiti takozvanu Big-O notaciju.
- Ova notacija nam u suštini daje informacije o tome kako neke funkcije ograničavaju druge funkcije.
- Ako imamo funkciju n^2 i funkciju $100n$, u jednom trenutku funkcija n^2 će poslužiti kao gornja granica za funkciju od $100n$. Linearna funkcija će uvek ostati ispod n^2, kada se prođe određena vrednost za n.

$$\exists N > 0: \quad \forall n > N: \quad n^2 > 100n$$

Running Time – Big O

$$\exists N > 0 \land \exists C > 0: \quad \forall n > N: \; f(n) < g(n) * C$$

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

- Ako postoji početni indeks N i proizvoljna konstanta C (oba veća od nule) tako da je za sve n koji su veći od ovog početnog indeksa funkcija f uvek manja od funkcije g pomnožena sa ovom konstantom, možemo reći da je **f u Big-O od g**.

- Ili, ako funkcija g, pomnožena proizvoljno odabranom konstantnom, preraste funkciju f u bilo kom trenutku u koordinatnom sistemu (zauvek), možemo reći da je **f u Big-O od g**.

Running Time – Big O

$$f(n) = 20n^2 + 5n + 10 \qquad g(n) = n^2$$

- Očigledno, dokle god je pozitivan, f će uvek vratiti veću vrednost od g. Ali ovo nije bitno. Želimo da analiziramo **asimptomatski odnos**.
- Dakle, ako uspemo da pronađemo konstantu i početni indeks, tako da matematičko stanje iznad bude ispunjeno, i dalje možemo reći da je f vezan iznad g.
- Bez obzira koliko je konstanta koju izaberemo velika, kada je n jednaka nuli, f će uvek biti veća od g, pošto ima konstantnu sumu deset. Pa hajde da izaberemo jedan kao naš početni indeks N. Koji faktor, bi trebalo da izaberemo za C, da bismo uvek imali veću vrednost na desnoj strani?

Running Time – Big O

$$20n^2 + 5n + 10 < n^2 * C \quad (n \geq 1)$$

- Da podelimo sve sa n na kvardat: $20 + \frac{5}{n} + \frac{10}{n^2} < C \quad (n \geq 1)$

- Hajde da testiramo sa 1 za n jer će svako veće n dati manju vrednost za C

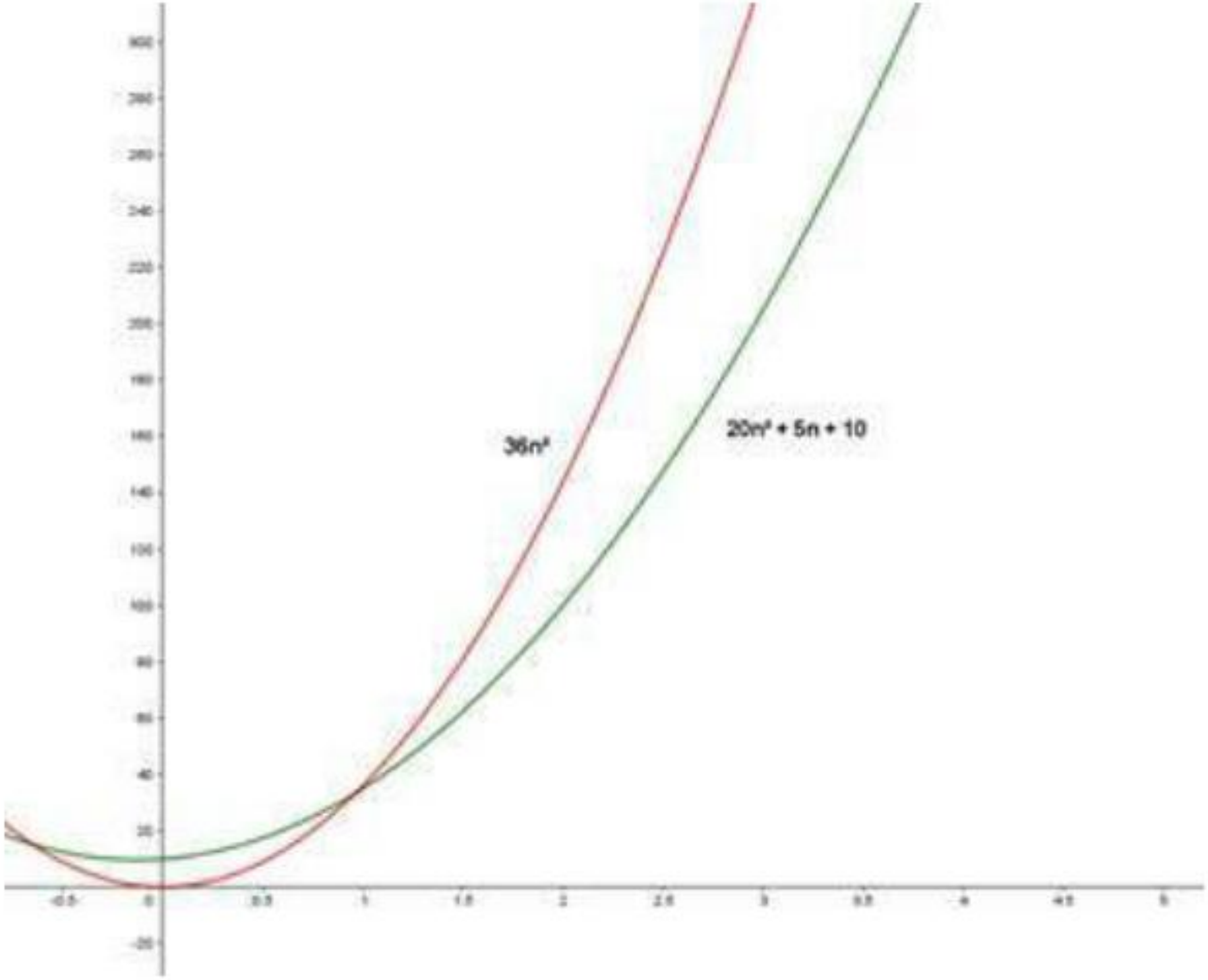
$$20 + \frac{5}{1} + \frac{10}{1} = 35 < C$$

- Dakle C može biti 36

Running Time – Big O

- Na slici možete videti da oko n = 1, naša novoizgrađena funkcija prerasta datu funkciju.
- Stoga je **f u Big-O od g**.

$$20n^2 + 5n + 10 = O(n^2)$$



Running Time – Big O

- Ovo ne važi za bilo koju funkciju!
- Na primer n^3 nikada se ne moze vezati za n^2 bez obzira koje konstante se odaberu, pošto n^3 raste brže od n^2 i dodatni faktor n, uvek će prevazići svaku datu konstantu.

Running Time – Omega notacije

- Cela magija radi i za donje granice. Ako želimo da kažemo da je funkcija vezana ispod drugom funkcijom, koristimo omega notaciju umesto Big-O notacije.

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

$$\exists N > 0 \land \exists C > 0: \quad \forall n > N: \quad f(n) > g(n) * C$$

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

Running Time – Omega notacije

- Ovde želimo da pokažemo da 200 * n^2 služi kao donja granica n^2. Da bismo to uradili, potrebno je da pronađemo dovoljno malu konstantu i polaznu tačku, za koju je to slučaj. Hajde da izaberemo nulu kao polaznu tačku i nađemo konstantu, koja zadovoljava nejednakost.

$$f(n) = n^2 \qquad g(n) = 200 * n^2$$

$$n^2 > 200 * n^2 * C \qquad (n \geq 0)$$

$$1 > 200 * C \qquad (n \geq 0)$$

$$\frac{1}{200} > C \qquad (n \geq 0)$$

Running Time – Teta notacije

- Neke funkcije služe kao gornja i kao donja granica za istu funkciju.
- Kada je to slučaj obe funkcije su asimpatički ekvivalentne i imamo posebnu notu za to – theta notacija.

$$f(n) = O\big(g(n)\big) \wedge f(n) = \Omega(g(n))$$

$$\rightarrow f(n) = \theta(n)$$

Running Time – konstantno vreme

- Konstantno vreme: složenost izvršenja ne raste.
- Ista količina koraka je potrebna bez obzira na veličinu unosa.
- Ovde nije bitno da li je konst. suma jedna ili pet hiljada.

Zapamtiti: Konstante su nebitne!

- Kada želimo da kažemo da algoritam rešava problem u konst. vremenu, to možemo da uradimo koristeći Big-O ili theta notation:

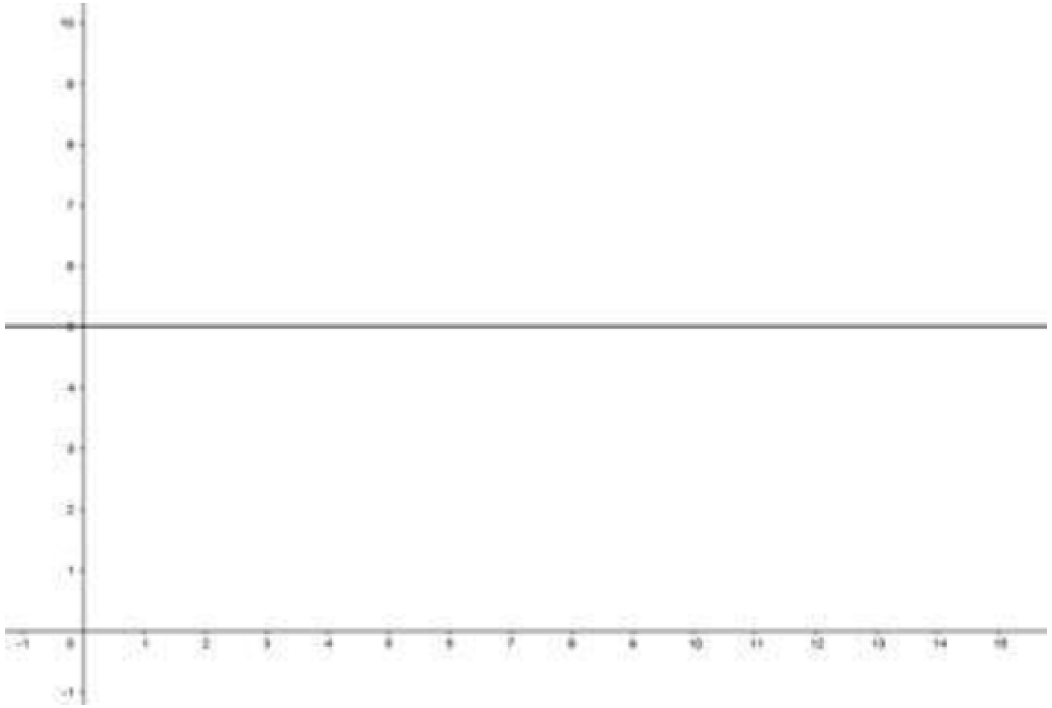
$$O(1) \qquad \theta(1)$$

Running Time –logaritamsko vreme

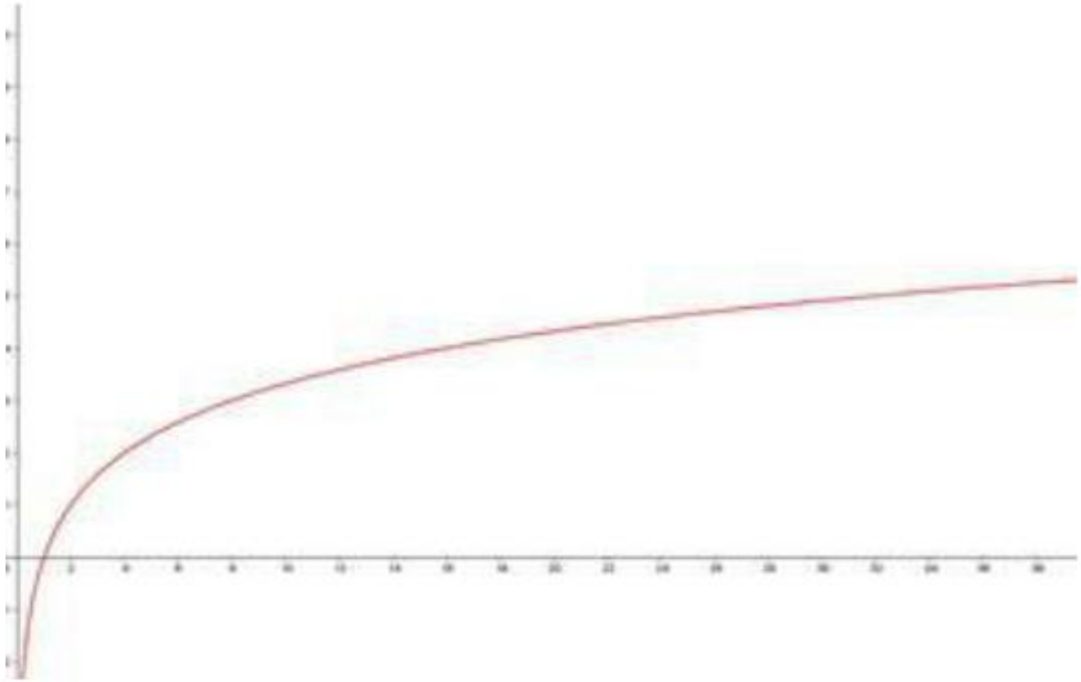
- Ukratko, to govori koliko puta morate da pomnožite bazu sa sobom da biste dobili dati broj kao rezultat toga. Npr. logaritam baze 2 od 32 je pet jer je dva na pet 32.
- Obično naiđemo na logaritamsko vreme kad god je veličina problema prepolovljena sa svakom iteracijom. Binarna pretraga na sortiranoj listi je savršen primer za to.

$$O(\log n) \qquad \theta(\log n)$$

Running Time – konst. i logaritamsko vreme



$O(1)$ $\theta(1)$



$O(\log n)$ $\theta(\log n)$

Running Time – linearno vreme

- Rast vremena je konstantan i zavisi od broja ulaznih elemenata, složenost koraka algoritma je ista.
- Primer za linearno vreme bi bio pronalaženje maksimalne vrednosti neuređene liste. Potrebno je da prođete kroz svaki element da biste dobili rezultat. Više elemenata na listi znači više iteracija i poređenja

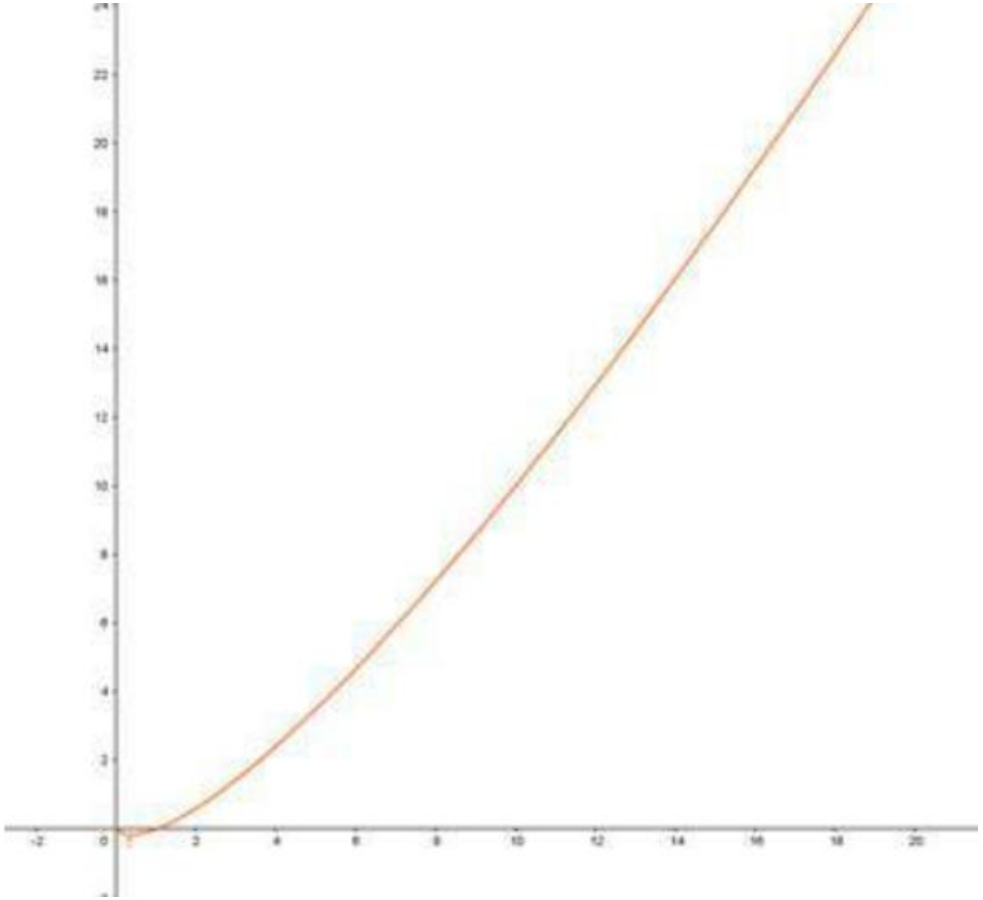
$$O(n) \qquad \theta(n)$$

Running Time – pseudo-linearno vreme

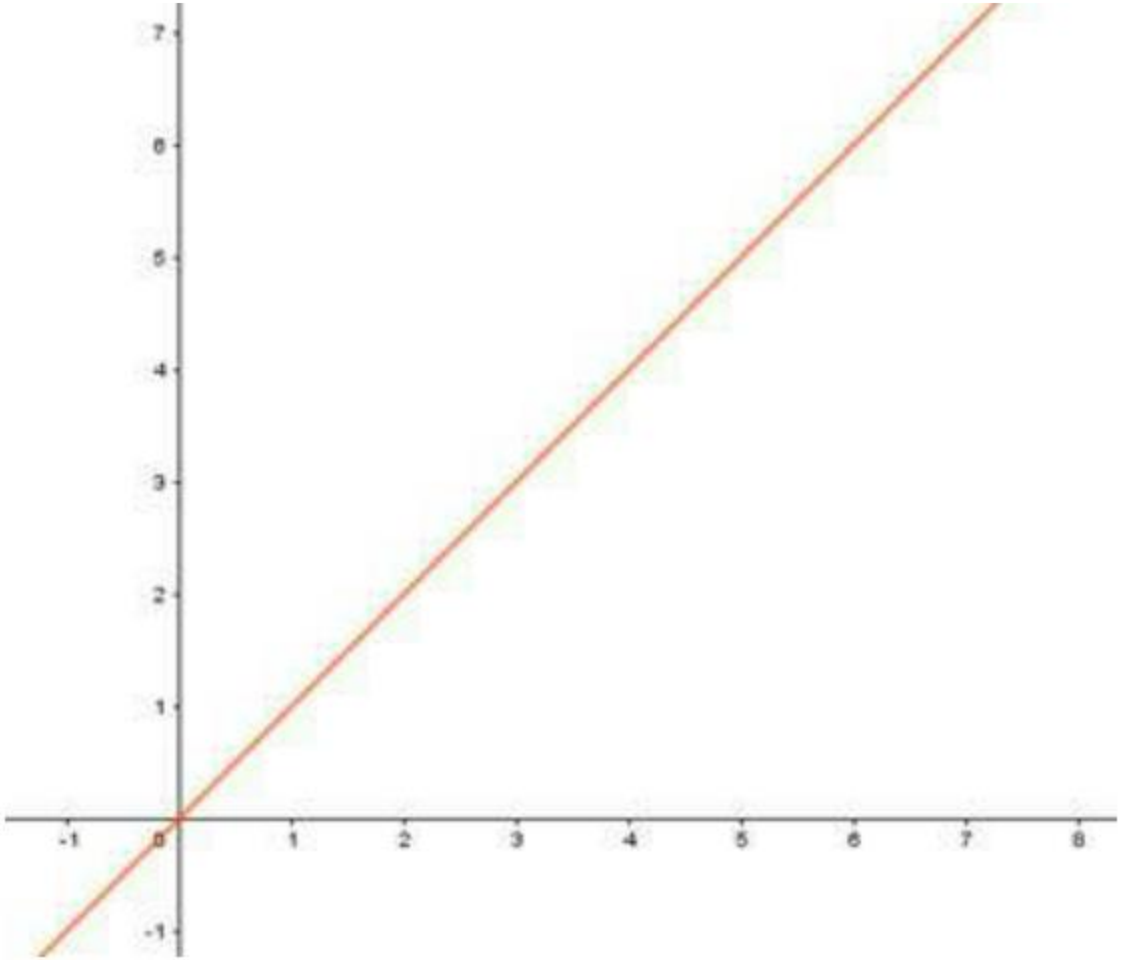
- Ako kombinujemo poslednje dve kompleksnosti izvršavanja (logaritamske i linearne), dobijamo pseudo-linearno vreme.
- Ponekad se naziva i linearitamski. Ovog puta složenost se često nalazi u **algoritmima podele i osvajanja**.
- Naići ćemo na ovo vreme izvršavanja kada dođemo do naprednijih algoritama sortiranja kao što su sortiranje, brzo sortiranje i heap sortiranje.

$$O(n \log n) \qquad \theta(n \log n)$$

Running Time – pseudo-linearno i linearno vreme



$O(n \log n)$ $\theta(n \log n)$



$O(n)$ $\theta(n)$

Running Time – kvadratno vreme

- Sad već ulazimo u neefikasnije algoritme u poređenju sa prethodnim...
- Kvadratno vreme znači da ne samo da vreme raste dodavanjem novih elemenata za obradu algoritmom nego što ih je više ubrzanije rastu...ali njime još možemo upravljati.
- Primeri za kvadratičko vreme izvršavanja su neefikasni algoritmi sortiranja kao što su sortiranje mehurića (bubble sort) sortiranje umetanja ili sortiranje selekcije, ali i prelazak preko jednostavnog 2D niza.

$$O(n^2) \qquad \theta(n^2)$$

Running Time – polinomijalno vreme

- Kad god imamo kompleksnost izvršavanja n na stepen k (k je konstanta), bavimo se polinomskom kompleksnošću izvršavanja.

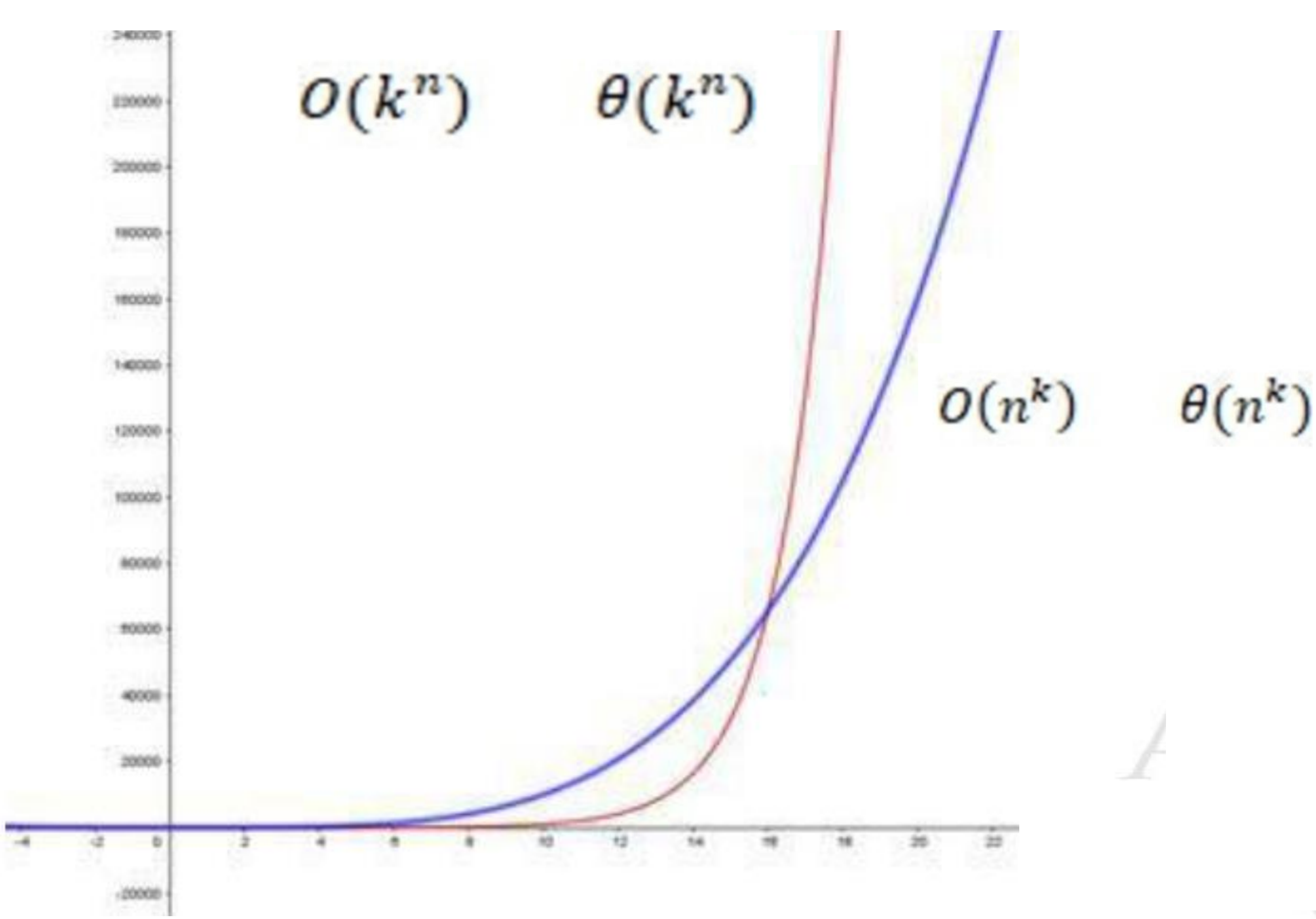
$$O(n^k) \qquad \theta(n^k)$$

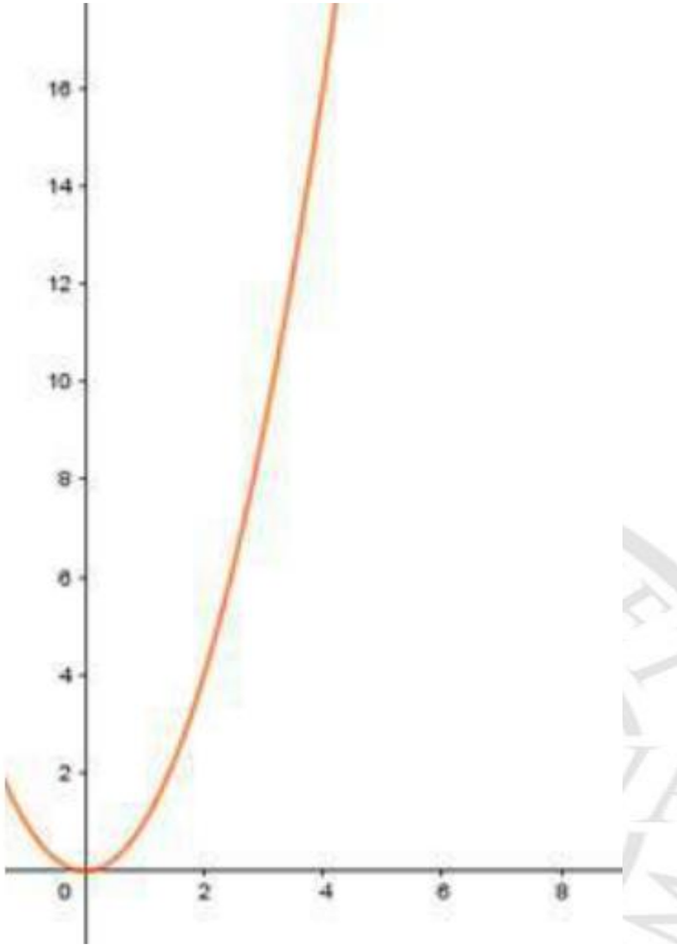
Running Time – eksponencijalno vreme

- Ovo želimo da izbegnemo, jer ne možemo da kontrolišemo.
- Primeri za probleme koji se (bar za sada) mogu rešiti samo u eksponencijalnom vremenu su brutalno forsiranje lozinki i pronalaženje suma podskupa u skupu brojeva.

$$O(k^n) \qquad \theta(k^n)$$

Running Time – eksponencijalno, polinomijalno i kvadratno vreme





$O(n^2)$ $\theta(n^2)$

Running Time – faktorijalno vreme

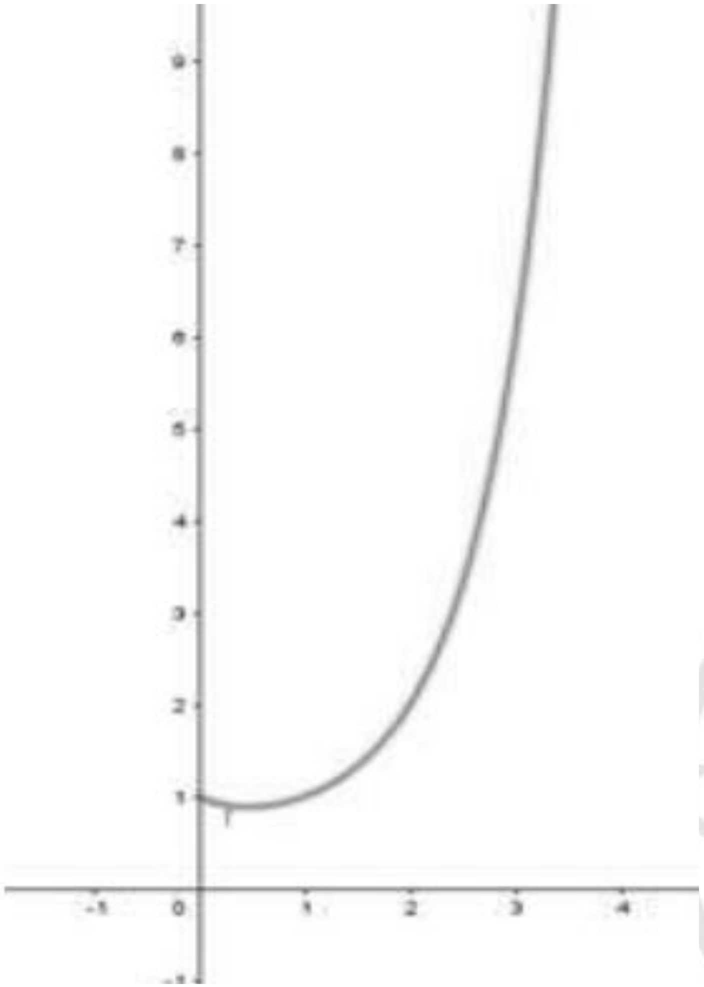
- Najgora opcija. Mi u suštini množimo n sa svim prirodnim brojevima koji su manji od n osim nule. Jedna kompleksnost izvršavanja koja je gora od ovoja je od oblika n na n-ti stepen, ali se ona skoro nikada ne javlja.

  $n! = n * (n-1) * (n-2) * \ldots * 3 * 2 * 1$

- Međutim, dolazi do faktorijalnog vremena. Na primer, kada pokušate da pronađete sve moguće permutacije datog skupa elemenata. Ali postoji i veoma praktičan i važan problem koji se rešava u faktorijalnom vremenu – problem putujućeg prodavca.

$$O(n!) \qquad \theta(n!)$$

Running Time – faktorijalno vreme



$O(n!)$ $\theta(n!)$

Running Time – algoritmi izračunavanje: Primer 1

```
k := 0 (1 PO)
for i = 1 to n: (n times)
    for j = (i + 1) to n: (???)
        k = k + j + i (1 PO)
return k (1 PO)
```

$$(n-1)+(n-2)+(n-3)+\cdots+3+2+1+0$$

$$\sum_{i=1}^{n}(\mathrm{n}-\mathrm{i}) = (n-1)+(n-2)+\cdots+(n-n)$$

$$\sum_{i=1}^{n}(\mathrm{n}-\mathrm{i}) = \frac{n*(n-1)}{2}$$

$$\frac{n*(n-1)}{2} = \frac{1}{2}*(n^2-n)$$

$$n^2-n=\theta(n^2)$$

Running Time – algoritmi izračunavanje: Primer 2

```
k := 0
i := 2^n
while i > 0:
    k = k + i²
    i = ⌊i/2⌋
return k
```

Korak 1

k := 0 **(1 PO)**
i := 2^n **(1 PO)**
while i > 0: **(???)**
  k = k + i^2 **(1 PO)**
  i = ⌊i/2⌋ **(1 PO)**

Korak 2

k := 0 **(1 PO)**
i := 2^n **(1 PO)**
while i > 0: **(n + 1 times)**
  k = k + i^2 **(1 PO)**
  i = ⌊i/2⌋ **(1 PO)**

Korak 3

$$k = (2^n)^2 + (2^{n-1})^2 + (2^{n-2})^2 + \cdots$$

$$\sum_{i=0}^{n} (2^{n-i})^2 = \frac{4^{n+1} - 1}{3}$$

$$4 * \frac{4^n - 1}{3} \rightarrow \theta(4^n)$$

Izračunavanje