Algoritmi

1 Osnove

- Algoritam je konkretan niz instrukcija po kojem se riješava problem ili vrši neko računanje.
- Prilikom opisa algoritama koristi se pseudo-kôd koji je dovoljno jasan da prikaže ideju algoritma ali i da je dovoljno sličan poznatim programskim jezicima. Npr. bubble sort algoritam je:

Algorithm 1 Bubble sort

```
Require: A : 0-indexed list of sortable items
function BubbleSort(A: list)
    n ← length(A)
    swapped ← true
    while swapped do
        swapped ← false
        for i ← 1 to n − 1 inclusive do
            if A[i − 1] > A[i] then
                swap(A[i − 1], A[i])
                swapped ← true
            end if
        end for
    end while
end function
```

2 Analiza izvršavanja algoritama

- Kako bi se moglo znati da li je jedan algoritam bolji od drugog algoritma, koristi se **analiza kompleksnosti** izvršavanja.
- Ideja analize kompleksnosti je prikazivanje koliko će algoritam morati obaviti operacija u poređenju na promjene na ulaznim parametrima algoritma.
- Pod "obaviti operacija" ne misli se na tačan broj operacija već na "odokativan" broj operacija.

Primjer: Napisati algoritam koji, za ulazni broj n računa zbir svih brojeva od 1 do n.

Algorithm 2 Sum

```
Require: n : number n ≥ 0
function Sum(n: number)
    sum ← 0
    for i ← 1 to n inclusive do
        sum ← sum + i
    end for
    return sum
end function
```

Ukoliko je $n = 5$ onda se može jednostavno primjetiti da će funkcija `Sum()` obaviti 5 operacija. Ukoliko je $n = 10$, ona će obaviti 10 operacija. Tj. za neko n na ulazu, funkcija će obaviti n operacija. Tako se može reći da je algoritam u funkciji `Sum()`, **u linearnoj zavisnosti** sa ulazom n.

Ukoliko se primjeti da postoji formula za rješavanje ove vrste problema, onda:

Algorithm 3 Sum2

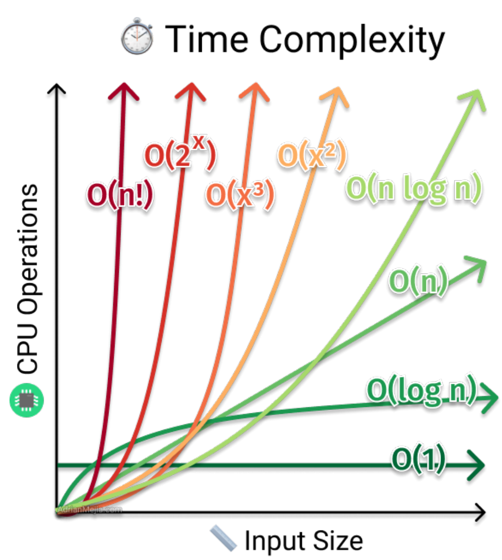
Require: n : number $n \geq 0$

function SUM2(n: number)
  $sum \leftarrow (n * (n+1))/2$
  return sum
end function

Analizirajući ovaj algoritam, vidimo da nema petlje i da se za bilo koje n na ulazu, izvršava tačno određen, i konstantan broj operacija. Tako se može reći da je za izvršavanje `Sum2()` potreban konstantan broj operacija i da nije ovisan o ulaznom parametru n.

3 big-O notacija

- Kako bi se analiza algoritama zapisala u prepoznatljivom obliku, koristi se big-O notacija.
- Zapisuje se kao $\mathcal{O}(x)$ gdje x predstavlja vrstu zavisnosti koju algoritam ima.
- Postoji nekoliko poznatih i prepoznatljivih zavisnosti:
  1. $\mathcal{O}(1)$ — konstantan: algoritam uvijek ima konstantan broj operacija
  2. $\mathcal{O}(n)$ — linearan: broj operacija algoritma raste kako raste i n
  3. $\mathcal{O}(log(n))$ — logaritamska: broj operacija algoritma raste kao što raste i $log(n)$. S tim da baza logaritma ovisi od same prirode algoritma.
  4. $\mathcal{O}(n * log(n))$ — logaritamska po svakom n
  5. $\mathcal{O}(n^2)$ — kvadratna
  6. $\mathcal{O}(2^n)$ — eksponencijalna
  7. $\mathcal{O}(n!)$ — faktorijelna
- Ako se ovo vizuelizira, onda:



- Kako bi se pojednostavilo pisanje kompleksnosti algoritma vrijede pravila big-O notacije:
  1. $\mathcal{O}(1) + \mathcal{O}(n) = \mathcal{O}(1+n) = \mathcal{O}(n)$ — uzima se veća kompleksnost
  2. $2 * \mathcal{O}(n) = \mathcal{O}(n)$ — konstanta se zanemaruje, jer je bitna brzina rasta u odnosu na n, jer konstanta samo može praviti konfuziju

Primjer: Za algoritam 3, može se reći da funkcija `Sum2` ima kompleksnost $\mathcal{O}(1)$ ili da se **izvršava za konstantno vrijeme**.

Primjer: Za algoritam 2, kaže se da se **izvršava za linearno vrijeme u odnosu na** n, ili $\mathcal{O}(n)$.

Primjer: Analizirati **insertion sort**.

Algorithm 4 Insertion Sort

Require: A : 0-indexed list of sortable items

```
function InsertionSort(A: list)
    i ← 1
    while i < length(A) do
        j ← i
        while j > 0 and A[j − 1] > A[j] do
            swap(A[j], A[j − 1])
            j ← j − 1
        end while
        i ← i + 1
    end while
end function
```

Insertion sort je algoritam koji simulira sortiranje karata u ruci. Radi tako što uzima element koji nije najmanji, te pomjera sve elemente koji su veći od njega za jedno mjesto desno.

Primjetno je da vanjska petlja ima $n = length(A)$ iteracija. Unutrašnja petlja je uslovljena time da li je posmatrani element manji od elemenata sa lijeve strane.

Ako se posmatra najlošiji slučaj tj. elementi u ulazu su postavljeni kao najveći lijevo a najmanji desno (npr. $5, 4, 3, 2, 1$), onda će za svaku iteraciju vanjske petlje, unutrašnja petlja obaviti do n operacija. To daje kompleksnost od $\mathcal{O}(n^2)$.

Međutim, ukoliko je ulazni niz već sortiran (npr. $1, 2, 3, 4, 5$), onda se unutrašnja petlja neće uopšte izvršavati jer uslov $A[j-1] > A[j]$ nikad neće biti zadovoljen. Te u ovom slučaju kompleksnost će biti $\mathcal{O}(n)$.

Tako se kaže da se ovaj algoritam izvršava za linearno vrijeme u najboljem slučaju $\mathcal{O}(n)$ i za kvadratno vrijeme u najgorem slučaju $\mathcal{O}(n^2)$.

Primjer: Napisati i analizirati algoritam **binarnog pretraživanja** nad sortiranim ulaznim nizom.