Teorija Algoritama

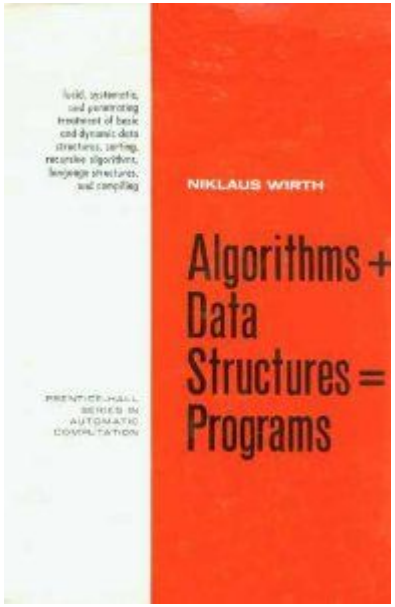
Uvod u algoritme

Šta je algoritam?

- Definicija – *Algoritam je precizno definisan postupak sa konačnom listom koraka za rešavanje nekog problema.*
- Naziv duguje arapskom filozofu *Al Khwarizmi*-ju
- Kao način predstavljanja (definisanja) algoritama, na ovim vežbama, koristićemo pseudo-kod i programski jezik C

Strukture podataka

- Niklaus Wirth (tvorac Pascal-a):
  - Algorithms + Data Structures = Programs
- Algoritmi su neodvojivo vezani sa strukturama podataka i obrnuto:
  - Algoritam obično na ulazu koristi neke podatke i kao rezultat produkuje podatke u nekom obliku.
  - Rad sa bilo kojom strukturom podataka zahteva poznavanje odgovarajućih algoritama za manipulaciju podataka unutar te strukture (pristup, brisanje, dodavanje, …).



Osobine algoritama

- Korektnost (ispravnost)
  - Da li tačno radi ono što treba da radi?
  - Da li uvek radi?
  - Da li se uvek završava?
- Performansa (efikasnost)
  - Koliko mu vremena treba da se izvrši? (Vremenska složenost)
  - Koliko memorije zauzima? (Prostorna složenost)
- Kompleksnost (za razumevanje)
  - Koliko je lak/težak za implementaciju?

Korektnost algoritma

- Postoje brojne metode za formalno dokazivanje korektnosti algoritma.
- Jedna od najintuitivnijih metoda koristi princip matematičke indukcije.
- Opšti postupak dokazivanja korektnosti matematičkom indukcijom sadrži:
  - Dokaz baze indukcije – za polazni slučaj;
  - Definisanje induktivne hipoteze;
  - Dokaz u opštem slučaju – korak indukcije.

Korektnost algoritma – invarijanta petlje

- Korektnost algoritma ili dela algoritma koji je predstavljen pomoću petlje, najčešće se izvodi metodom invarijante petlje.
- Invarijanta petlje – tvrđenje koje je tačno:
  - pre svakog ulaska u petlju (iteraciju);
  - posle svake izvršene iteracije;
  - nakon završetka petlje.
- Iz tačnosti invarijante petlje na kraju izvršenja algoritma sledi korektnost algoritma.

Korektnost algoritma - primeri

- Suma prvih *N* prirodnih brojeva;
- Konverzija celih brojeva iz dekadnog u binarni brojčani sistem;
- Minimalni/maksimalni element;
- *Selection sort*;
- Najveći zajednički delilac.

Suma prvih *N* prirodnih brojeva - pseudo-kod

```
S = 0
for i = 1 to N
    S = S + i
```

Ukoliko je *N* dati prirodan broj, potrebno je dokazati da nakon završetka algoritma definisanog pseudo-kodom iznad, promenljiva *S* ima vrednost jednaku zbiru prvih *N* prirodnih brojeva.

Suma prvih *N* prirodnih brojeva - dokaz

- Invarijanta petlje:
  - Na kraju svake iteracije *k*, promenljiva *S* sadrži zbir prvih *k* prirodnih brojeva.
  - Matematički: $S_k = \sum_{i=1}^{k} i$
- Dokaz baze indukcije
  - U prvoj iteraciji petlje vrednosti su *S = 0*, *i = 1*.
  - **Provera**: Da li je, nakon izvršenja prve iteracije, *S* jednaka zbiru prvih *k = i = 1* brojeva? Da, promenljiva *S* će biti uvećana za *i* (0 + 1 = 1)
  - **Zaključak**: Invarijanta važi pri izvršenju prve iteracije.
- Definisanje induktivne hipoteze – Pretpostavimo da na kraju neke proizvoljne iteracije *k*, invarijanta važi, tj. da će nakon *k*-te iteracije u promenljivoj *S* biti zbir prvih *k* prirodnih brojeva.

Suma prvih *N* prirodnih brojeva - dokaz

- Dokaz u opštem slučaju – Moramo dokazati da ako invarijanta važi na kraju *k*-te iteracije, važiće i nakon *(k+1)*-e.
- Unutar petlje se dešava: $S_{novo} = S_{staro} + i \Leftrightarrow S_{k+1} = S_k + i$
- Pošto znamo da je: $S_k = \sum_{j=1}^{k} j$
- Onda je: $S_{k+1} = \left(\sum_{j=1}^{k} j\right) + i = \sum_{j=1}^{k+1} j$
- Nakon izvršenja tela petlje *i se uvećava*: $i_{novo} = i_{staro} + 1$
- Možemo konstatovati da na kraju *(k+1)*-e iteracije, nova suma *S* sadrži zbir prvih *k + 1* elemenata.
- **Zaključak**: Invarijanta je očuvana.

Suma prvih *N* prirodnih brojeva - dokaz

- Završetak – *Šta se dešava nakon završetka petlje?*
- Petlja se prekida kada uslov i ≤ n prestane da važi. Dakle, poslednja iteracija koja će se izvršiti je *n*-ta
- To se dešava tačno kada je *k* = *i* = *n*
- Pošto invarijanta važi i nakon poslednje izvršene iteracije, zamenimo *k* sa *n* u našoj formuli: $S_n = \sum_{j=1}^{n} j$
- **Konačni dokaz**: Dobili smo da je *S* upravo zbir prvih *N* prirodnih brojeva, što dokazuje da je algoritam korektan.

Konverzija celih brojeva iz dekadnog u binarni brojčani sistem – pseudo-kod

```
ostatak_niz = []
privremeni_n = n
while privremeni_n > 0:
    ostatak = privremeni_n % 2
    ostatak_niz.append(ostatak)
    privremeni_n = privremeni_n // 2
# Rezultat su ostaci pročitani unazad
```

Konverzija celih brojeva iz dekadnog u binarni brojčani sistem – dokaz

- Invarijanta petlje:
  - Nakon proizvoljne *k*-te iteracije, originalni broj: $n = (privremeni\_n \cdot 2^k) + V(ostatak\_niz)$
    Gde je *V(ostatak_niz)* vrednost koju formiraju do sada prikupljeni ostaci (binarno interpetirani)
- Dokaz baze indukcije:
  - Pre prvog ulaska (k = 0) – *privremeni_n = n, a ostatak_niz* je prazan, pa je njegova vrednost 0.
  - Formula: $n = (n \cdot 2^0) + 0 = n \cdot 1 = n$
  - **Zaključak**: Invarijanta važi na samom početku.
  - *Primetiti da je dokaz baze indukcije moguće definisati sa stanjem nakon prve iteracije (k=1) kao u prethodnom primeru i sa stanjem pre prve iteracije (k=0) kao u ovom primeru.*
- Definisanje induktivne hipoteze – Pretpostavimo da nakon *k* koraka važi da smo “izvukli” jedan deo broja u binarne cifre, a da je ostatak vrednosti sačuvan u *privremeni_n* pomnoženom težinom 2^k.

Konverzija celih brojeva iz dekadnog u binarni brojčani sistem – dokaz

- Dokaz u opštem slučaju – Pod pretpostavkom da invarijanta važi nakon *k*-te iteracije, treba da dokažemo da važi i nakon *(k+1)*-e iteracije:
- Uzimamo novi ostatak: $ostatak = privremeni\_n_k \% 2$
- Izračunamo novi privremeni_n: $privremeni\_n_{k+1} = privremeni\_n_k // 2$
- Matematički $privremeni\_n_k$ možemo zapisati kao: $2 \cdot privremeni\_n_{k+1} + ostatak$ i to možemo zameniti u glavnoj formuli:
  $n = \left((2 \cdot privremeni\_n_{k+1} + ostatak) \cdot 2^k\right) + V(ostatak\_niz)$
  $n = 2^{k+1} \cdot privremeni\_n_{k+1} + 2^k \cdot ostatak + V(ostatak\_niz)$
- Primetimo da je *ostatak* zapravo vrednost koja treba da se nalazi na *k*-toj poziciji u *ostatak_niz*, tako da ostajemo sa formulom:
  $n = 2^{k+1} \cdot privremeni\_n_{k+1} + V(ostatak\_niz)$
- **Zaključak**: Invarijanta je očuvana za korak *k + 1*.

Konverzija celih brojeva iz dekadnog u binarni brojčani sistem – dokaz

- Završetak – Petlja se završava kada *privremeni_n* postane 0.
- Tada naša formula postaje: $n = (0 \cdot 2^k) + V(ostatak\_niz)$

$$n = V(ostatak\_niz)$$

- **Konačni dokaz**: Vrednost originalnog broja *n* je sada u potpunosti predstavljena sumom bita (ostataka) pomnoženih odgovarajućim stepenima dvojke. To znači da je niz ostataka koji smo dobili zaista binarna reprezentacija broja *n*.

Selection Sort – pseudo-kod

```
for i from 0 to n-2:
    min_idx = i
    for j from i+1 to n-1:
        if A[j] < A[min_idx]:
            min_idx = j
    A[i], A[min_idx] = A[min_idx], A[i] # Swap
```

Selection Sort – dokaz

- Iako postoje dve ugnježdene petlje, tehnički bi trebalo da se dokazuju dve invarijante petlje (unutrašnja pa spoljašnja), međutim unutrašnja petlja je algoritam za pronalazak minimalnog elementa u nizu. S tim u vezi, u primeru ćemo se fokusirati na invarijantu spoljašnje petlje.
- Invarijanta (spoljne) petlje – Na kraju svake iteracije *k*, važe dva uslova:
  - Podniz `A[0 … k-1]` se sastoji od *k* najmanjih elemenata originalnog niza.
  - Podniz `A[0 … k-1]` je **sortiran**.
- Dokaz baze indukcije – Posle prve iteracije (*k = 1, i = 0*):
  - Pod pretpostavkom da je algoritam za pronalazak najmanjeg elementa ispravan, nakon prve iteracije pronaći ćemo najmanji element u nizu i postaviti ga na *0*-ti indeks.
  - Podniz `A[0 … k-1], tj. A[0 … 0]` je niz sa jednim elementom i kao takav je uvek sortiran.
  - Ispravnost algoritma za pronalazak najmanjeg elementa nam garantuje da smo na *0*-ti indeks stavili najmanji element niza, stoga se podniz `A[0 … k-1]` sactoji od *k = 1* najmanjih elemenata u nizu.
  - **Zaključak**: Invarijanta važi nakon prve iteracije.

Selection Sort – dokaz

- Definisanje induktivne hipoteze – pretpostavimo da nakon *k*-te iteracije, prvih *k* elemenata niza `A[0 … k-1]` čine sortiran podniz *k* najmanjih elemenata iz celog skupa.
- Dokaz u opštem slučaju:
  - Pod pretpostavkom da invarijanta važi nakon *k*-te iteracije, treba da dokažemo da važi i nakon *(k+1)*-e iteracije.
  - Algoritam za pronalazak najmanjeg elementa pronalazi najmanji element u nesortiranom delu niza: `A[k … N-1]`.
  - Algoritam zatim menja mesta elementima `A[k]` i `A[min_idx]` `(ne smeta min_idx==k)`.
  - Pošto je `A[min_idx]` bio najmanji u nesortiranom delu, on je sigurno veći ili jednak od svih elemenata u već sortiranom delu `A[0 … k-1]`.
  - Kada ga stavimo na poziciju *k*, novi podniz `A[0 … k]` je i dalje sortiran i sada sadrži k + 1 najmanjih elemenata.
  - **Zaključak**: Invarijanta je očuvana za sledeću iteraciju (*k* + *1*).

Selection Sort – dokaz

- Petlja (spoljna) se završava kada *i* dostigne *n - 1*.
  - Prema našoj invarijanti to znači da podniz `A[0 … n-2]` sortiran i sadrži *n - 1* najmanjih elemenata celog niza.
  - Ako je *n - 1* elemenata na svojim pravim mestima i sortirano, preostali poslednji element (`A[n-1]`) mora biti najveći i samim tim se već nalazi na svojoj ispravnoj (poslednjoj) poziciji.
  - **Konačni dokaz**: Ceo niz `A[0 … n-1]` je sortiran.

Performansa algoritma

- Kada govorimo o performansama algoritama, uglavnom pomislimo na brzinu izvršavanja. Međutim, šta sve utiče na brzinu izvršavanja algoritma?
  - Sam algoritam;
  - Količina podataka;
  - Mašina na kojoj se izvršava.
- Kako bismo na realan način izračunali performansu algoritma, u smislu vremenske složenosti, moramo eliminisati mašinu na kojoj se izvršava algoritam iz proračuna.
- Kako to izvesti? -Umesto da merimo vreme izvršavanja, brojaćemo koliko primitivnih operacija (aritmetičke operacije, uslovni i bezuslovni skokovi, čitanje i upis u memoriju) algoritam treba da izvrši, u odnosu na količinu podataka.

Performansa algoritma - primer

Za primer merenja performansi algoritma prebrojaćemo broj osnovnih operacija koje algoritam za pronalazak minimalnog elementa u nizu dužine *N* elemenata izvrši.

```
min_idx = 0
for i=1 to N-1:
    if(A[i] < A[min_idx])
        min_idx = i
```

1. Pre same petlje imamo jednu operaciju (pisanje u memoriju);
2. U telu petlje imamo 2 operacije (poređenje i dodela);
3. Sama petlja se izvršava N-1 put.

Zaključak: Broj osnovnih operacija je:

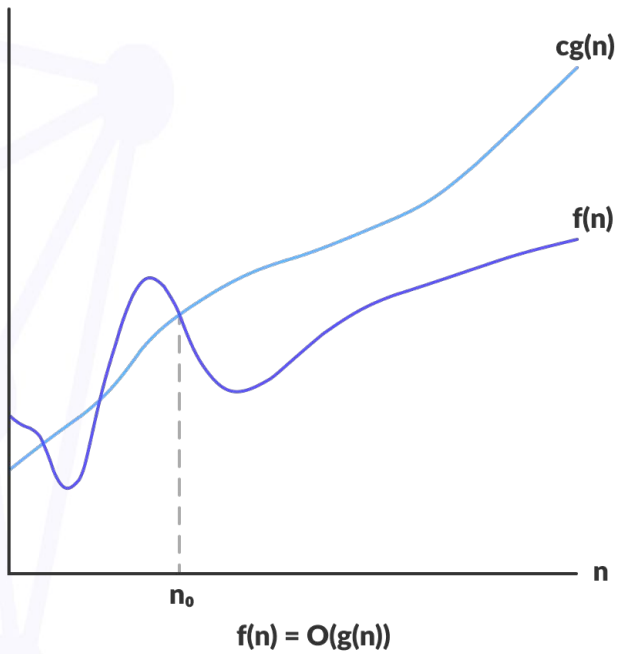
$$1 + (N - 1) \cdot 2 = 2 \cdot N - 1$$

Asimptotske notacije

- Asimptotske notacije nam nude mogućnost da ocenimo kako će se vremenska složenost algoritma ponašati sa porastom količine podataka, odnosno sa povećanjem broja *N*.
- O notacija – Gornja granica – *Worst case*
- Ω notacija – Donja granica – *Best case*
- Θ notacija – Čvrsta granica – *Average case*

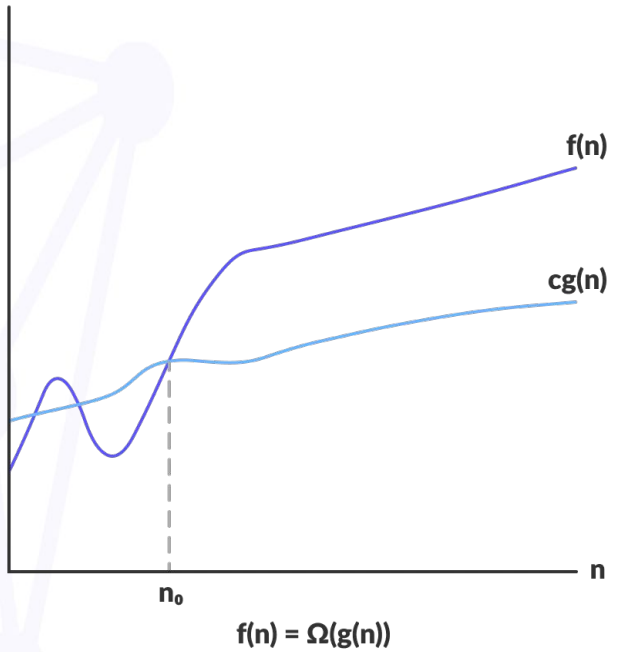
O notacija

- Ova notacija nam govori da algoritam neće raditi sporije od određene funkcije.
- **Definicija:** f(n) = O(g(n)) ako postoje konstante c i n_0 takve da je $0 \leq f(n) \leq c*g(n)$ za sve $n \geq n_0$
- **Primer:** Linearna pretraga – algoritam koji u najgorem slučaju mora da prođe kroz ceo niz – algoritam ima O(n) vremensku složenost.



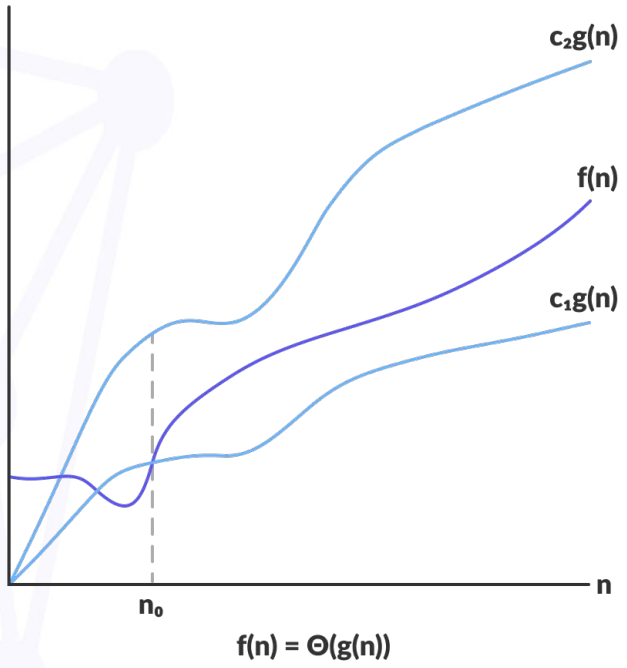
Ω notacija

- Ova notacija nam govori da algoritam neće raditi brže od određene funkcije.
- **Definicija:** f(n) = Ω(g(n)) ako postoje konstante c i n_0 takve da je $f(n) \geq c*g(n) \geq 0$ za sve $n \geq n_0$
- **Primer:** Optimizovani *Bubble Sort* – algoritam za sortiranje, koji u slučaju da sortira već sortirani niz (*best case*) ima O(n) vremensku složenost.



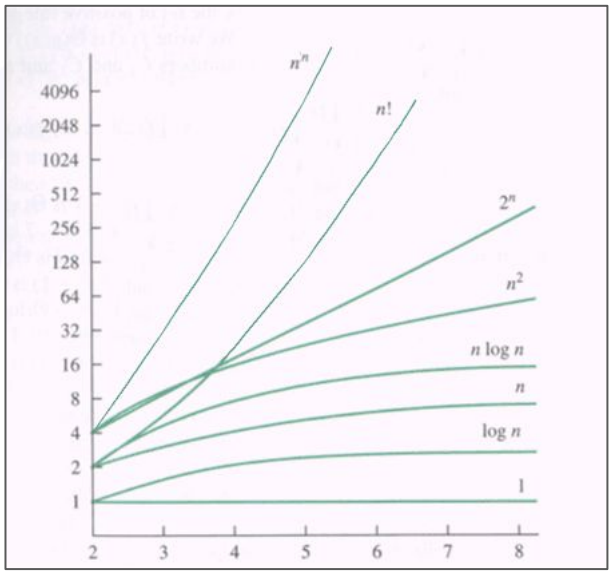
Θ notacija

- Kada su gornja i donja granica iste, koristimo ovu notaciju. Dakle algoritam se ponaša tačno po toj funkciji.
- **Definicija:** f(n) = Θ(g(n)) ako postoje konstante c_1, c_2 i n_0 takve da je $c_2 * g(n) \geq f(n) \geq c_1 * g(n) \geq 0$ za sve $n \geq n_0$
- **Primer:** Sabiranje svih brojeva u nizu uvek zahteva tačno *N* koraka – Θ(n).



Asimptotske notacije

- Prilikom određivanja vrednosti asimptotskih notacija za funkciju f(n) zanemaruju se konstante i uzima se u obzir samo najbrže rastući član funkcije.
- Primeri:
  - $f(n) = 9*\log n + 5*(\log n)^4 + 3*n^2 + 2*n^3 = O(n^3)$
  - $f(n) = 2^n + n^2 + 3*n = O(2^n)$
- Uglavnom ćemo koristiti O ili Θ notaciju pošto nas zanima vremenska složenost u najgorem slučaju.



Zadatak 1

Izračunati O notaciju za sledeće funkcije i algoritme:

1. $f(n) = \log n + n! + n^{100}$
2. $f(n) = 5*n + \log n + 15$
3. $f(n) = 32*n + n*\log n$
4. $f(n) = 2^n + n^n + n!$
5. Selection sort (pseudo-kod na slajdu 16)

Merenje vremena u C-u

- Zaglavlje `<time.h>`
- Tip `clock_t` služi za beleženje broja otkucaja procesorskog sata.
- Funkcija `clock()` vraća trenutni broj otkucaja sata.
- Simbolička konstanta `CLOCKS_PER_SEC` označava broj otkucaja sata u sekundi.
- **Primer**: `primer1.c`

Pokazivači na funkcije u C-u

- Naziv funkcije je pokazivač na funkciju.
- Prilokom definicije pokazivača na funkciju neophodno je navesti:
  - Tip povratne vrednosti;
  - Tipove i redosled parametara.
- Pokazivač na funkciju omogućava:
  - Prenos funkcije kao parametra u drugu funkciju;
  - Kreiranje niza funkcija;
  - Vraćanje funkcije kao povratne vrednosti druge funkcije.
- **Primer**: `primer2.c`

Zadatak 2

Napisati program koji poredi vreme izvršavanja binarne i linearne pretrage.

1. Napisati funkciju koja vrši binarnu pretragu.
2. Napisati funkciju koja vrši linearnu pretragu.
3. Napisati funkciju koja kao parametar prima funkciju, meri vreme njenog izvršavanja i kao povratnu vrednost vraća vreme u milisekundama.
4. Uz pomoć implementiranih funkcija sprovesti sledeći test scenario:
   a. Generisati sortirani niz od 100000 jedinstvenih elemenata celobrojnog tipa.
   b. Uporediti vreme izvršavanja pretraga za elementom na sledećim indeksima:
      i. 355
      ii. 32767
      iii. 70921

*Koristiti kostur zadatka iz datoteke `zadatak2_postavka.c` u kojoj možete pronaći i funkciju za generisanje sortiranog niza sa jedinstvenim elementima.