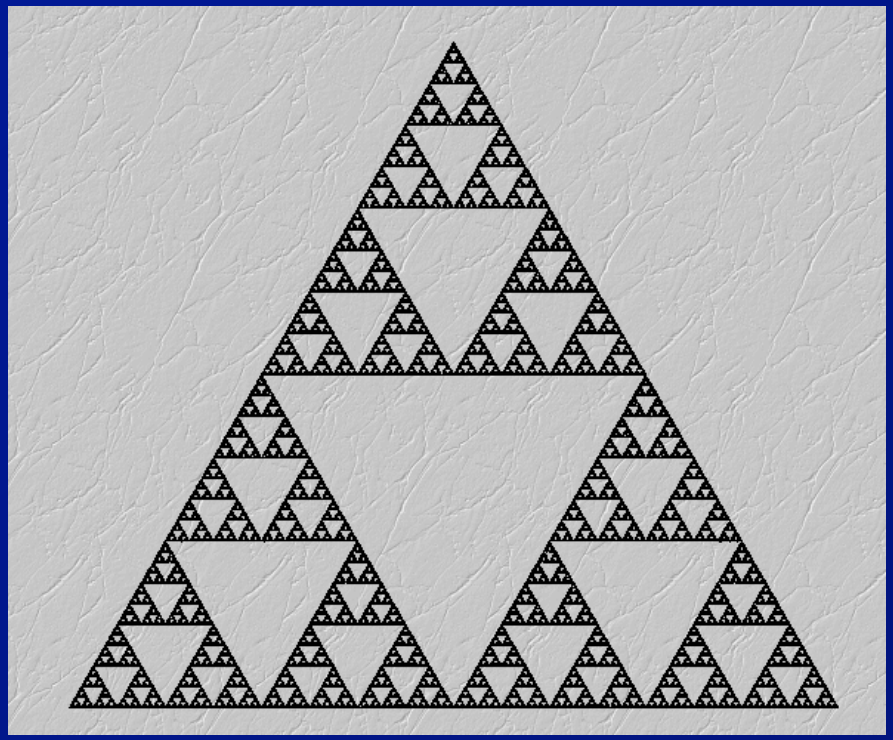
Rekurzije



Kratke informacije

- moodle.Carnet račun za SPA: goo.gl/Df4DZ
- email za domaće zadaće: sasa.ceci@gmail.com
- Kolokviji: 100 posto bodova, 50 posto je potrebno za prolaz
- Domaće zadaće:
  - prvih pet koji točno riješe i pošalju zadaću dobiva 10 posto koji se pribrajaju bodovima na kolokviju
  - ne pomaže za prolaz, ali pomaže za bolju konačnu ocjenu
  - maksimum je tri takve zadaće u semestru
  - i one zadaće nakon prvih pet pomažu za konačnu ocjenu (zalaganje)

Definicija rekurzije i primjeri

- Rekurzija - računanje elemenata niza koristeći vrijednosti prethodnih elemenata

- Eksplicitno (nerekurzivno) zadana formula za izračunavanje općeg člana niza:

  $a_n = 2^n$

  $a_n = (3^n - 1) / 2$

- Rekurzivno zadana formula: vrijednost n-tog člana ovisi o prethodnim članova niz

  $a_n = 2\, a_{n-1}$, $n \geq 2$, $a_1 = 2$

  $a_n = 3\, a_{n-1} + 1$, $n \geq 2$, $a_1 = 1$

- Aritmetički niz:

  $a, b \in R$, $b \neq 0$

  $a_1 = a$, $a_n = a_{n-1} + b \Rightarrow [a_n = a + (n-1)\, b]$

- Geometrijski niz:

  $a, q \in R$, $q \neq 0$ ni 1

  $a_1 = a$, $a_n = q\, a_{n-1} \Rightarrow [a_n = a\, q^{n-1}]$

Rekurzije u programiranju

- Rekurzija u programu poziva same sebe (rutine, funkcije, algoritma, pravila)
- Rekurzivni programi su kraći, ali je izvođenje programa u pravilu dulje
- Rekurzije koriste stog (struktura podataka) za pohranjivanje rezultata i povratak iz rekurzije
- Stariji kompjuterski jezici (ranije verzije FORTRAN-a) nisu podržavali rekurzije
- Primjer1 : funkcija koja računa zbroj prvih n prirodnih brojeva:

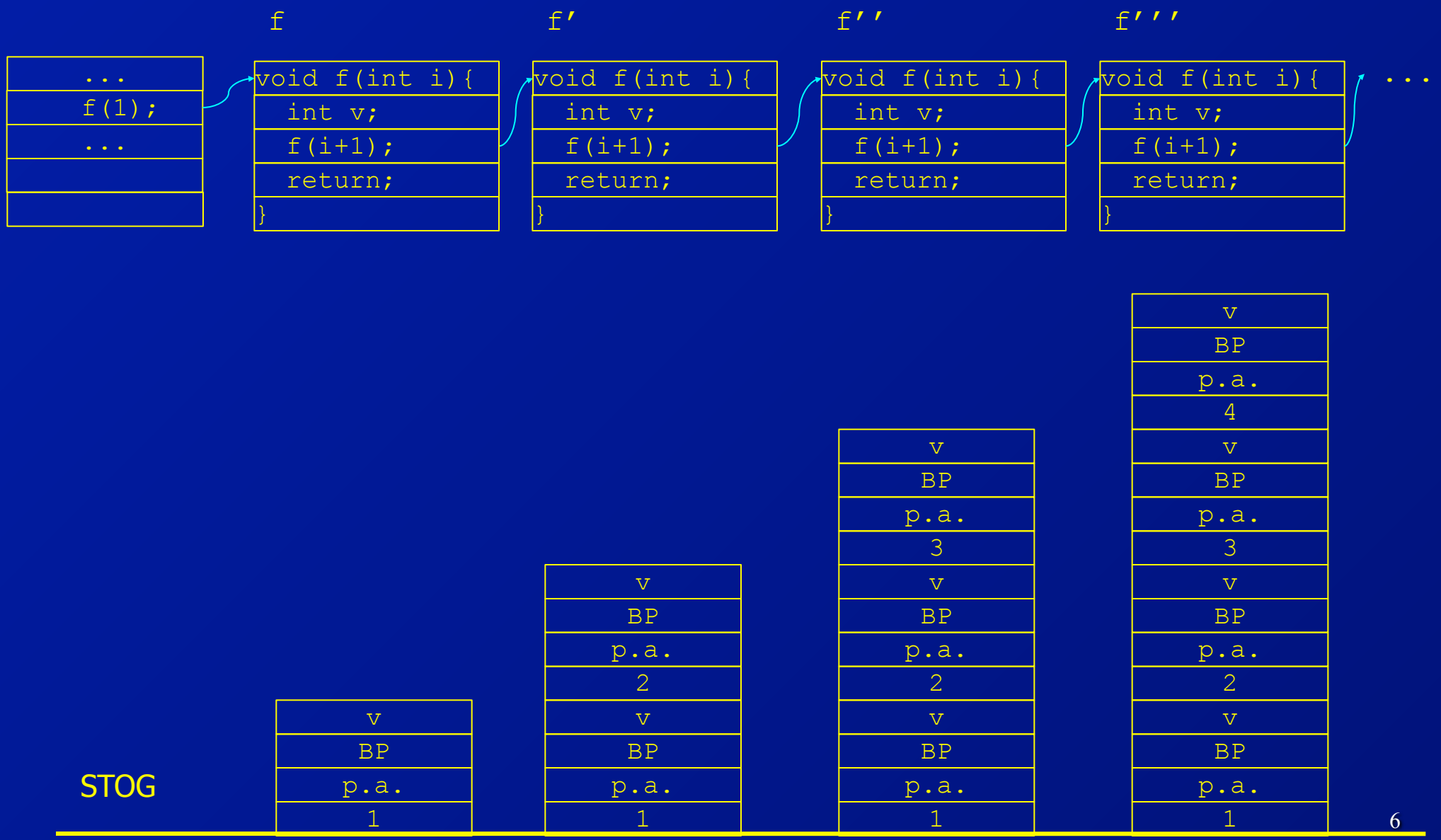
```
int ZbrojPrvih (int n) {
   if (n == 1) return 1;
   return ZbrojPrvih (n – 1) + n;
}
```

- Primjer 2: funkcija rezervira mjesto za polje cijelih brojeva i poziva samu sebe

```
void f (int i) {
        int v[1000];
        printf("%d ",i);
        f (i+1);
       return;
}
```

- Što se događa kad u svom programu pozovete ovu funkciju?
- Izvršava se dok se ne popuni dopušteni memorijski prostor za stog
- Kako to izgleda u memoriji?

Elementarna rekurzija i stog



Izračunavanje faktorijela

- Primjer 3: Jedan od jednostavnih rekurzivnih algoritama jest izračunavanje faktorijela.

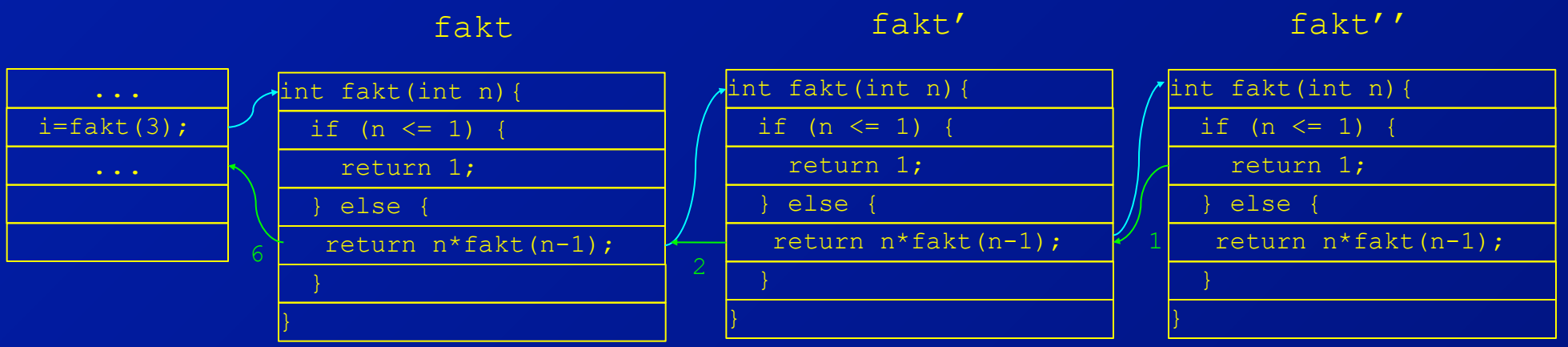
0! = 1
1! = 1
…
n! = n (n-1)!

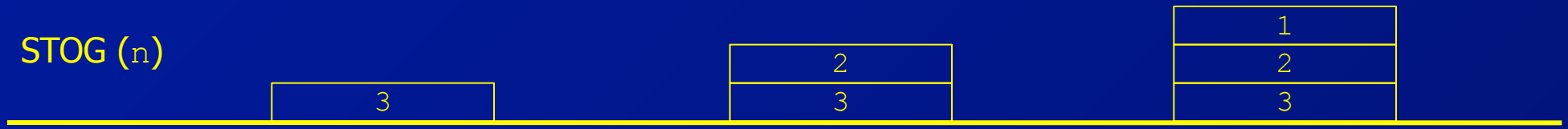
```
int fakt(int n){
     if (n <= 1) {
         return 1;
    } else {
         return n * fakt(n-1);
    }
}
```

Izvršavanje programa za slučaj n = 4:

```
rezultat = fakt(4);
         = 4 * fakt(3);
         = 4 * 3 * fakt(2);
         = 4 * 3 * 2 * fakt(1);
         = 4 * 3 * 2 * 1;
```

Izračunavanje faktorijela (primjer n=3)





Fibonaccijev niz

- Leonardo iz Pise (a.k.a. Fibonacci) uveo je početkom 13. stoljeća pojam nizova u tadašnjoj Europi

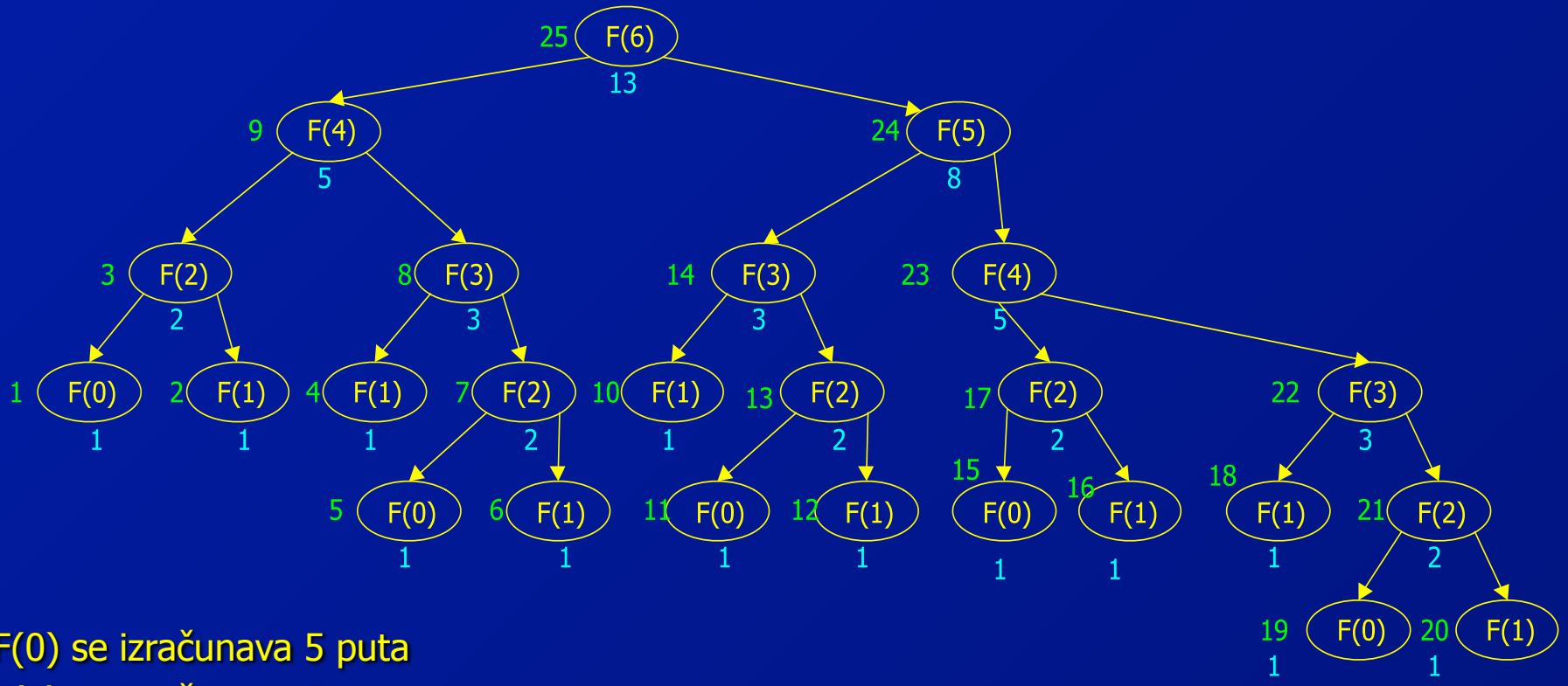
- Niz koji nosi njegovo ime dobiva se kao rješenje sljedećeg (bitno pojednostavljenog) problema: Ako je svaki par zečeva plodan nakon mjesec dana, a na svijet donose potomstvo nakon sljedećih mjesec dana, i ako je to potomstvo uvijek par zečića (jedan mužjak i jedna ženka), koliko će parova zečeva biti o d jednog para nakon N mjeseci?

F_0 = 0 sp + 1 np (np = novi par)
F_1 = 1 sp + 0 np (sp = stari par)
F_2 = 1 sp + 1 np
F_3 = 2 sp + 1 np
F_4 = 3 sp + 2 np
F_5 = 5 sp + 3 np
$F_0 = F_1 = 1$
$F_n = F_{n-1} + F_{n-2}$

```
int F(int n) {
    if (n <= 1) return 1;
    else return F(n-2) + F(n-1);
}
```

- Rješenje: broj parova povećava se prema ovom nizu 1, 1, 2, 3, 5, 8, 13, 21, 34,...
- Ovo nije efikasan algoritam (rutina direktno prati matematički algoritam)

Fibonaccijevi brojevi - efikasnost



- F(0) se izračunava 5 puta
- F(1) se izračunava 8 puta
- F(2) se izračunava 5 puta
- F(3) se izračunava 3 puta
- F(4) se izračunava 2 puta
- F(5) se izračunava 1 puta
- F(6) se izračunava 1 puta
- Ukupno izračunavanja: 25

Najveća zajednička mjera

- Jedan od najstarijih i najpoznatijih algoritama je Euklidov postupak za pronalaženje najveće zajedničke mjere (nzm) dva prirodna (nenegativna) cijela broja:

1. Ako je a < b zamijeni a i b
2. r = a % b [a mod b, ostatak dijeljenja a s b]
3. a = b
4. b = r
5. Ako je r ≠ 0, idi na 2
6. Rezultat je a

```
int nzm(int a, int b) {
  if(b == 0) return a;
  return nzm (b, a % b);
}
```

- Primjeri:
  - nzm(22,8) = nzm(8,6) = nzm(6,2) = nzm(2,0) = 2
  - nzm(21,13) = nzm(13,8) = nzm(8,5) = nzm(5,3) = nzm(3,2) = nzm(2,1) = nzm(1,0) = 1

Traženje elementa u polju

- Rekurziju možemo koristiti i za traženje nekog elementa u polju.
- Malo konkretnije:
  - tražimo indeks i prvog člana
  - u jednodimenzionalnom polju A od n elemenata
  - koji ima vrijednost x
  - Ako nema takvoga, rezultat je -1

```
int trazi (nekitip A[], nekitip x, int n, int i) {
            if(i >= n) return -1;
            if(A[i] == x) return i;
            return trazi (A,x,n,i+1);
        }
```

  - Pretraživanje počinje pozivom funkcije `trazi(A,x,n,0)`.
  - I ovo možemo malo ubrzati...

Traženje elementa u polju (2)

- Pretraživanje je brže ako se polje A proširi za jedan element i na kraj stavi tražena vrijednost x

```
int trazix1 (tip A[], tip x, int i){
      if(A[i] == x) return i;
      return trazix1 (A, x, i+1)
      }
```

- Primjer:

```
int i;
const int n = 10;
double x=7, A[n]={1,2,3,4,5,6,7,8,9,10};
A[n] = x; //trik: ako je i=n, to znači da nema x-a u polju
if ((i = trazix1(A,x,0)) == n){}
printf("i = %d", i);
```

Traženje najvećeg člana polja

- Određivanje indeksa najvećeg člana u polju od n članova

```
int maxel(int A[], int i, int n) {
    int imax;
    if (i >= n-1) return n-1;
    imax = maxel(A, i + 1, n);
    if (A[i] > A[imax]) return i;
    return imax;
}
```

- Primjer: `A=[2,5,9,1,3]`

```
                    i = 0, 1, 2, 3, 4
                imax = 2
                       2
                          2
                             4
                                4
```

- U predhodnom slučaju nije bio zadovoljen princip strukturiranog programiranja da u jednom modulu ima samo jedan izlaz.
- Strukturirana verzija koda je:

```
int maxclan1(int A[], int i, int n) {
    int imax, ret;
    if (i >= n-1) ret = n-1;
    else {
         imax = maxclan1 (A, i + 1, n);
         if (A[i] > A[imax]) ret = i;
         else ret = imax;
         }
   return ret;
}
```

Karakteristike rekurzije

- **Osnovni slučaj**: uvijek mora postojati osnovni slučaj koji se rješava bez rekurzije
- **Napredovanje**: za slučajeve koji se rješavaju rekurzivno, svaki sljedeći rekurzivni poziv mora biti približavanje osnovnom slučaju
- **Funkcionalnost**: podrazumijeva se da svaki rekurzivni poziv funkcionira
- **Ne ponavljati**: nije dobro da se isti problem rješava odvojenim rekurzivnim pozivima

- Primjer za pogrešku

```
int los (int n) {
    if (n == 0) return 0;
    return los(n/3 + 1) + n - 1;
}
```

- Prva pogreška je u tome što je za vrijednost n=1 rekurzivni poziv ponovo s argumentom 1, što znači da nema napredovanja prema osnovnom slučaju. Program, međutim, neće raditi ni za druge vrijednosti argumenta. Npr. za n=4, rekurzivno se poziva los s argumentom 4/3+1=2, zatim 2/3+1=1 i dalje stalno 1/3+1=1.

Uklanjanje rekurzije

- Prednosti rekurzije:
  - koncizniji opis algoritma
  - lakše je dokazati ispravnost

- Nedostaci:
  - produljeno vrijeme izvođenja (u pravilu)
  - neki jezici ne podržavaju rekurziju

- Formalna pravila za uklanjanje rekurzije:

1. Na početku funkcije umetne se deklaracija stoga, inicijalno praznog. Stog služi za pohranu svih podataka vezanih uz rekurzivni poziv: argumenata, vrijednosti funkcije, povratne adrese
2. Prva izvršna naredba dobije oznaku `L1`.

- Svaki rekurzivni poziv zamijenjen je naredbama koje obavljaju sljedeće:

3. Pohrani na stog vrijednosti svih argumenata i lokalnih varijabli.

4. Kreiraj `i`-tu novu oznaku naredbe `Li` i pohrani `i` na stog. Vrijednost `i` će se koristiti za izračun povratne adrese. U pravilu 7. je opisano gdje se u programu smješta ta oznaka.
5. Izračunaj vrijednosti argumenata ovog poziva i pridruži ih odgovarajućim formalnim argumentima.
6. Umetni bezuvjetni skok na početak funkcije.
7. Oznaku kreiranu u točki 4. pridruži naredbi koja uzima vrijednost funkcije s vrha stoga. Dodaj programski kod koji tu vrijednost koristi na način opisan rekurzijom.

- Ovime su uklonjeni svi rekurzivni pozivi. Umjesto naredbe return napravi sljedeće:

8. Ako je stog prazan, obavi normalnu naredbu `return`.
9. Inače, uzmi vrijednosti svih izlaznih argumenata i stavi ih na vrh stoga.
10. Odstrani sa stoga povratnu adresu ako je bila stavljena, i pohrani je u neku varijablu.
11. Uzmi sa stoga vrijednosti za sve lokalne varijable i argumente.
12. Umetni naredbe za izračun izraza koji slijedi neposredno iza naredbe `return` i pohrani rezultat na stog.
13. Idi na naredbu s oznakom povratne adrese.

- Primjer
  - Uklanjanje rekurzije iz strukturiranog programa za traženje indeksa najvećeg elementa u polju.
  - Početni kod je promijenjen u:

```
if(i < n-1){
      imax = maxclan2(A,i+1,n);
      if(A[i] > A[imax]) return i;
      else return imax;
} else return n-1;
```

```
int maxclan2 (int A[], int i, int n) {
      int imax, k, adresa, vrh, *stog;                    //Pravilo  1
      vrh = -1;
      stog = (int *) malloc (2 * n * sizeof(int));
      L1:                                                 //Pravilo  2
        if (i < n-1) {
             vrh++; stog[vrh] = i;                        //Pravilo  3
             vrh++; stog[vrh] = 2;                        //Pravilo  4
             i++;                                          //Pravilo  5
             goto L1;                                     //Pravilo  6
             L2:                                          //Pravilo  7
             imax = stog[vrh]; vrh--;
             if (A[i] > A[imax])
                 k = i;
              else
                 k = imax;
        } else {
             k = n-1;
        }
        if (vrh == -1) {                                   //Pravilo  8
             free (stog);
             return k;
        } else {                                           //Pravilo  9
             adresa = stog[vrh]; vrh--;                    //Pravilo 10
             i = stog[vrh]; vrh--;                         //Pravilo 11
             vrh++; stog[vrh] = k;                         //Pravilo 12
             if (adresa == 2) goto L2;                     //Pravilo 13
       }
}
```

- Uklanjanje rekurzije je moguće obaviti i na jednostavniji način, (yeah! :O)) ako znamo i dobro razumijemo kako radi rekurzivna funkcija
  - Najjednostavnije je zamijeniti rekurzivni algoritam iterativnim algoritmom
  - Evo dva takva primjera

```
int maxclan3 (int A[], int n) {
  int i, imax;
  i = n-1;
  imax = n-1;                          // zadnji je najveći
  while (i > 0) {
    i--;
    if (A[i] > A[imax]) imax = i;
  }
 return imax;
}

int maxclan4 (int A[], int n) {
 int i, imax = 0;                      // prvi je najveći
 for (i = 1; i < n; i++) {
     if (A[i] > A[imax]) imax = i; }   // trebaju li nam {}?
 return imax;
}
```

ANALIZA SLOŽENOSTI ALGORITAMA

Pojam algoritma

- Najjednostavnije rečeno, algoritam je niz uputa za rješavanje nekog problema
- Ime dolazi od perzijskog matematičara iz 9. st. po imenu Muḥammad ibn Mūsā al-Khwārizmīi (prezime se čitalo nešto kao Al-Khwarithmi). Dodatno, riječ iz naslova njegove najvažnije knjige (...al-jabr...) i danas koristimo.
- Primjer algoritma: recepti u kulinarstvu
- Algoritmi se danas najviše koriste u računarstvu
- Profesor računalih znanosti sa Stanforda, Donald E. Knuth, sastavio je listu od pet opće prihvaćenih svojstava algoritama:
  1) **Konačni su** – moraju završiti nakon konačnog broja (konačnih) koraka
  2) **Dobro su definirani** – svaki korak sadržava dobro jasne i nedvojbene upute
  3) **Primaju vanjske podatke** - mogu, ali i ne moraju
  4) **Daju (i neki) rezultat** - osim što završe
  5) **Funkcionalni su** – provedivi su uz pomoć olovke i papira u konačnom vremenu

- Algoritam je postupak za rješavanje masovnog problema
  - Masovni problem je općenito pitanje na koje je potrebno naći odgovor, a koje ima parametre koji kod zadavanja problema ostaju neodređeni

- Pri definiciji masovnog problema potrebno je zadati
  1) generalni opis svih parametara
  2) ograničenja koja rješenje masovnog problema mora zadovoljavati

- Specificiranjem svih parametara masovnog problema dobiva se instanca problema

- Algoritam rješava masovni problem ako daje rješenje za svaku pojedinu instancu problema

Složenost algoritama

- Za rješavanje istog problema se mogu naći različiti algoritmi
  - Pitanje je - kako odabrati?
  - Potrebno je pronaći mjeru efikasnosti algoritama da bi se moglo odrediti koji je bolji
  - Mjeri se količina resursa koje algoritam troši da bi riješio problem

- Dvije mjere koje se najčešće koriste su vrijeme potrebno za izvršenje algoritma i prostor potreban za pohranjivanje ulaznih podataka, međurezultata i izlaznih rezultata
  - Govorimo tako o vremenskoj i prostornoj složenosti algoritma
  - Mi ćemo se pozabaviti samo vremenskom složenošću algoritama (koja se češće koristi u ocjenjivanju efikasnosti algoritma)

- Vremenska složenost algoritma (VSA):
  - VSA određujemo da bi procijenili potrebno vrijeme za proračun i utvrdili efikasnost algoritma
  - Prvi problem je pronalaženje mjere koja ne ovisi o brzini računala na kojem se algoritam izvodi, već o samom algoritmu
  - Tako VSA nećemo izražavati vremenom potrebnim da se izračuna rezultat, već brojem elementarnih operacija koje se moraju izvesti
  - U određivanju VSA tako pretpostavljamo da je vrijeme dohvata sadržaja memorijske lokacije je fiksno, a vrijeme obavljanja elementarnih operacija (aritmetičke, logičke, pridruživanje, poziv funkcije) ograničeno nekom konstantom kao gornjom granicom
  - Broj operacija koje algoritam izvodi ovisi o veličini ulaza
  - Instance problema različitih dimenzija zahtijevaju različiti broj operacija
  - Naravno, složenost algoritma nije funkcija isključivo veličine ulaza - mogu postojati različite instance iste duljine ulaza, ali različitih složenosti (različite vrijednosti ulaza)
- Izbor skupova podataka za iscrpno testiranje algoritma:
  - Najbolji slučaj
  - Najgori slučaj (najčešće se koristi ova ocjena)
  - Prosječan slučaj - matematičko očekivanje, najispravniji pristup, ali najsloženija analiza
- Točan račun složenosti se ne isplati raditi, traži se aproksimativna jednostavna funkcija koja opisuje kako se složenost algoritma mijenja s veličinom ulaza
- A priori analiza daje trajanje izvođenja algoritma kao vrijednost funkcije nekih relevantnih argumenata (teorija)
- A posteriori analiza je statistika dobivena mjerenjem na računalu (praksa)

Analiza a priori

- A priori: procjena vremena izvođenja, nezavisno od računala, programskog jezika, prevoditelja (compilera)
- Ako se promatra algoritam koji sortira niz brojeva, tada se njegovo vrijeme izvršavanja izražava u obliku T(n), gdje je n duljina niza
- Ako se promatra algoritam za invertiranje matrice, tada se vrijeme izvršavanja izražava kao T(n), gdje je n red matrice

- Primjeri:

a)
```
x += y;
```
1

b)
```
for(i=1; i<=n; i++) x += y;
```
n

c)
```
for(i=1; i<=n; i++) {
    for(j=1; j<=n; j++) x += y;
}
```
n^2

```
// trebamo li {}?
```

Funkcija složenosti algoritma

- Uvodi se notacija koja jednostavno opisuje brzinu rasta složenosti algoritma
- Asimptotska ocjena rasta funkcije: pri izračunavanju složenosti aproksimacija se vrši tako da se koristi aproksimativna funkcija koja je jednostavnija od same funkcije složenosti, a dobro asimptotski opisuje rast funkcije složenosti
- Za mali n, aproksimacija ne mora vrijediti (ali to je nebitno)
- Nas ovdje ne zanima stvarni iznos funkcije složenosti, već samo koliko brzo ta funkcija raste (i koji je red veličine)
- Primjer: vrijeme izvršavanja Gaussovog algoritma za invertiranje matrice je proporcionalno s n^3, što znači da ako se red matrice udvostruči, invertiranje može trajati do 8 puta dulje

0-notacija

- $f(n) = O(g(n))$, ako i samo ako postoje dvije pozitivne konstante c i n_0 takve da vrijedi $|f(n)| \leq c|g(n)|$ za svaki $n \geq n_0$
- Tražimo najmanji $g(n)$ za koji to vrijedi
- Ovo znači da funkcija f(n) ne raste brže od funkcije g(n)
- Ako je broj izvođenja operacija nekog algoritma ovisan o nekom ulaznom argumentu n, koji ima oblik polinoma m-tog stupnja, onda je vrijeme izvođenja za taj algoritam $O(n^m)$.
  - Dokaz:

  Ako je vrijeme izvođenja određeno polinomom:

  $A(n) = a_m n^m + ... + a_1 n + a_0$

  onda vrijedi:

  $|A(n)| \leq |a_m| n^m + ... + |a_1| n + |a_0|$

  $|A(n)| \leq (|a_m| + |a_{m-1}|/ n + ... + |a_1| / n^{m-1} + |a_0| / n^m) n^m$

  $|A(n)| \leq (|a_m| + ... + |a_1| + |a_0|) n^m$, za svaki $n \geq 1$

  Uz:

  $c = |a_m| + ... + |a_1| + |a_0|$ i $n_0 = 1$

  tvrdnja je dokazana.

- Kao c se može koristiti bilo koja konstanta veća od $|a_m|$ ako je n dovoljno velik
- Ako neki algoritam ima k odsječaka čija trajanja iznose:
  $c_1 n^{m1}, \ c_2 n^{m2}, \ \ldots \ c_k n^{mk}$
  onda je ukupno trajanje
  $c_1 n^{m1} + c_2 n^{m2} + \ldots + c_k n^{mk}$
- To znači da je za taj algoritam vrijeme izvođenja jednako $O(n^m)$, gdje je
  $m = max\{m_i\}, \ i = 1,\ldots,k$

Korisne napomene:

1) Ako je a<b, onda n^a raste sporije od n^b
2) Za svaka 2 broja (a, b) iz skupa cijelih brojeva različitih od 1 vrijedi da $\log_a n$ raste jednako brzo kao $\log_b n$ (zato se često baza zanemari i pise se $\lg n$ – logaritam po proizvoljnoj bazi)
3) n^a raste sporije od $n^a \lg n$, koji pak raste sporije od n^{a+1}. Iz ovoga slijedi da logaritmi asimptotski rastu sporije od linearne funkcije
4) Za (a,b) cijele brojeve različiti od 1, a<b znači da n^a raste sporije od b^n. To znači da eksponencijalna funkcija raste brže od (svakog konačnog) polinoma
5) a,b cijeli brojevi različiti od 1, za a < b vrijedi da a^n raste sporije od b^n

- Dakle, za dovoljno veliki *n* vrijedi:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < \ldots < O(2^n)$$

  - $O(1)$ znači da je vrijeme izvođenja ograničeno konstantom
  - Ostale vrijednosti, osim zadnje, predstavljaju polinomna vremena izvođenja algoritma
  - Svako sljedeće vrijeme izvođenja je veće za red veličine
  - Zadnji izraz predstavlja eksponencijalno vrijeme izvođenja
    - Ne postoji polinom koji bi ga mogao ograničiti jer za dovoljno veliki *n* ova funkcija premašuje bilo koji polinom
    - Algoritmi koji zahtijevaju eksponencijalno vrijeme mogu biti nerješivi u razumnom vremenu, bez obzira na brzinu računala

- Donja granica za vrijeme izvođenja algoritma $f(n) = \Omega(g(n))$, ako i samo ako postoje dvije pozitivne konstante c i n_0 takve da vrijedi $|f(n)| \geq c\ |g(n)|$ za sve $n>n_0$.
  - Traži se najveći $g(n)$ za koji to vrijedi. Funkcija f ne raste sporije od funkcije g.
  - Ukoliko su gornja i donja granica jednake ($O = \Omega$), koristi se notacija:

    $f(n) = \Theta(g(n))$ ako i samo ako postoje pozitivne konstante c_1, c_2 i n_0 takve da vrijedi $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$ za sve $n>n_0$. Funkcije f i g rastu jednako brzo.

- $f(n) = o(g(n))$, ako i samo ako postoje dvije pozitivne konstante c i n_0 takve da vrijedi $|f(n)| < c|g(n)|$ za sve $n \geq n_0$. Funkcija f raste sporije od funkcije g.

- $f(n) = \omega(g(n))$, ako i samo ako postoje dvije pozitivne konstante c i n_0 takve da vrijedi $|f(n)| > c|g(n)|$ za sve $n \geq n_0$. Funkcija f raste brže od funkcije g.

- Primjer: traženje najvećeg člana u polju

```
int maxclan (int A[], int n) {
   int i, imax;
   i = n-1; imax = n-1;      //petlja se uvijek obavi n-1 puta →
    while (i > 0) {                  //Ω (n) = O(n) = θ (n)
            i--;
            if (A[i] > A[imax]) imax = i;}
   return imax;}
```

- Primjer: traženje člana u polju

```
// A-polje x-trazeni i-indeks od kojeg se trazi
int trazi (int A[], int x, int n, int i) {
   int ret;
   if (i >= n) ret = -1;
   else if (A[i] == x) ret = i;
   else ret = trazi (A, x, n, i+1);
  return ret;
}
```

  - vrijeme izvođenja je $O(n)$, ali je donja granica $\Omega(1)$. U najboljem slučaju u prvom koraku nađe traženi član polja, a u najgorem mora pregledati sve članove polja.

- Primjer: složenost Euklidovog algoritma (u najgorem slučaju)
  1. Ako je `a < b`, zamijeni `a` i `b`
  2. `r = a % b` (`a mod b`, ostatak dijeljenja `a` s `b`)
  3. `a = b` i `b = r`
  4. Ako je r različit od `0` idi na `2`
  5. Vrati `a`
- Koraci od 2 – 4 čine petlju koja se izvršava najviše b puta (ostatak od dijeljenja je uvijek manji od djelitelja, pa je r uvijek manji od b, što znači da se b uvijek smanjuje barem za 1, pa će b = 0 biti ispunjeno nakon najviše b koraka)
- Izvršavanje petlje traje konstantno vrijeme c_1, posljednji korak se izvodi jednom i njegova složenost neka je c_2
- Duljina ulaza n definira se kao duljina manjeg od ulaznih brojeva, pa slijedi:

$$T^{Euklid}_{max}(n) = c_1 * n + c_2 \quad \text{tj.}$$

$$T^{Euklid}_{max}(n) = \Theta(n)$$

Asimptotsko vrijeme izvođenja

- Asimptotsko vrijeme izvođenja je $f(n) \sim \theta(g(n))$
  - (čita se: "$f(n)$ je asimptotsko funkciji $g(n)$") ako je $\lim_{n\to\infty} f(n)/g(n) = 1$
  - Precizniji je opis vremena izvođenja nego 0-notacijom
  - Zna se i red veličine vodećeg člana i konstanta koja ga množi.
  - Primjer:

    Ako je
    $f(n) = a_k n^k + \ldots + a_0$, tada je
    $f(n) = O(n^k)$ te
    $f(n) \sim \theta(a_k\, n^k)$

- U izračunu učestalosti obavljanja nekih naredbi često se javljaju sume oblika:

$$\sum_{i=1}^{n} i = n(n+1)/2 = O(n^2)$$

$$\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6 = O(n^3)$$

$$\sum_{i=1}^{n} i^k = n^{k+1}/(k+1) + n^k/2 + \text{članovi nižeg reda} = O(n^{k+1})$$

$$\sim \Theta(n^{k+1}/(k+1))$$

Primjer: izračun vrijednosti polinoma u zadanoj točki

- Polinom n-tog stupnja je funkcija oblika (x je realan broj)

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

- Najočitiji i najjednostavniji algoritam za izračunavanje vrijednosti polinoma

1) $i = 0 ; p = 0$
2) $p = p + a_i x^i$
3) $i = i + 1$
4) ako je $i \leq n$ idi na 2
5) vrati p

- Odredimo broj računskih operacija u izvođenju ovog algoritma
  - n+1 povećavanje brojača i (zbrajanje)
  - Koraci 2-4 izvode se n+1 puta, a u svakom koraku imamo 1 zbrajanje, 1 množenje i 1 potenciranje.
  - Potenciranje odgovara nizu uzastopnih množenja. Za i > 1, to znači i – 1 množenje, što zajedno daje

$$\sum_{i=1}^{n-1} i = (n - 1)\, n / 2$$

Ukupno to sve iznosi n+1 zbrajanje i n+1+n/2*(n-1) množenja

- Primijetimo: u koraku 2 se izračunavaju sve veće potencije istog broja x. Svaka sljedeća potencija se može izračunati množenjem prethodne s x, čime se znatno smanjuje ukupan broj množenja
- Prepravljeni (brži) algoritam:

  1) $i = 0 ; p = 0 ; y = 1$

  2) $p = p + a_i * y$

  3) $i = i + 1 ; y = y * x$

  4) ako je $i \leq n$ idi na 2

  5) vrati p
- Broj operacija u ovom algoritmu:

  - n + 1 povećanje brojača (zbrajanje)
  - koraci 2-4 se izvode n+1 puta, u svakom koraku 1 zbrajanje, 2 množenja, što daje zajedno n+1 zbrajanje i 2(n+1) množenja

- Ovo je bitno brži algoritam od prošlog u kojem smo imali $(n^2 + n + 2) / 2$ množenja. Je li to najbolje?

Hornerov algoritam:

- Izluči li se u izrazu za polinom iz prvih n pribrojnika x, dobivamo

$$P_n(x) = (a_n x^{n-1} + a_{n-1} x^{n-2} + \ldots + a_1) x + a_0$$

Nastavimo li postupak još n-2 puta, dobivje se izraz

$$P_n(x) = ((\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_2)x + a_1)x + a_0$$

- Algoritam koji koristi gornji izraz:

  1) $i = n - 1$; $p = a_n$
  2) $p = p * x + a_i$
  3) $i = i - 1$
  4) ako je $i \geq 0$ idi na 2
  5) vrati p

- Broj operacija u algoritmu:

  - n smanjenja brojača (n oduzimanja = zbrajanja)
  - korake 2-4 prolazi n puta, u svakom koraku 1 zbrajanje i 1 množenje, što je ukupno n zbrajanja i n množenja

- Bitno je brži od oba prethodna algoritma, naročito prvog koji je $O(n^2)$, a od drugog je dvostruko brži

Analiza a posteriori

```
#include <stdio.h>
#include <math.h>
#include <time.h>

int main{
    float s = 0;
    time_t t1, t2; // time_t = unsigned long
    float tms;
    t1 = clock();
    for (int i=1; i<100000000; i++) {
        s = sqrt(s*s+i*i);
    }
    t2 = clock();
    tms = 1000 * ((float)t2 - (float)t1) / CLOCKS_PER_SEC;
    printf("Trajanje izvršavanja petlje je %f ms", tms);
    return 0;
}
```

Domaća zadaća 3

- Pravila za domaće zadaće:
  - Zadaće **ne pomažu** prolaznosti na kolokviju
    - (za prolaz treba imati više od 50 posto bodova),
  - Zadaće mogu povećati konačnu ocjenu
    - jer predavaču pokazuju zalaganje studenta
    - jer se slični zadaci/problemi mogu se pojaviti na kolokvijima i završnom ispitu
  - Prati se:
    - brzina (prvih pet studenata dobiva bodove, ali i ostalima se zadaće gledaju pozitivno)
    - točnost (ako zadatak u zadaći ima sintaksnu ili semantičku grešku, ne priznaje se)
    - vještina (brži algoritmi ili pametnije strukture se gledaju jako pozitivno)
    - zalaganje (više zadaća se na kraju gleda bolje nego manje)

- Domaća zadaća:
  - 1) Pronađite efikasniji algoritam od onog na predavanjima za Fibonaccijev niz i napisati funkciju u C++ koja ga računa (ne mora biti rekurzivna funkcija)
  - 2) Napravite usporedbu brzine izvođenja izračuna vrijednosti nekog polinoma u 100 točaka (recimo od 1 do 100) zadanog na klasičan način tako da napravite a posteriori testiranje. Polinomi neka budu 10-og, 20-og i 30-og reda.

Tu smo stali...

Primjer za različite složenosti istog problema

- Zadano je polje cijelih brojeva A_0, A_1, … A_{n-1}. Brojevi mogu biti i negativni. Potrebno je pronaći najveću vrijednost sume niza brojeva. Pretpostavit će se da je najveća suma 0 ako su svi brojevi negativni.

- Kubna složenost: Ispituju se svi mogući podnizovi. U vanjskoj petlji se varira prvi član podniza od nultog do zadnjeg. U srednjoj petlji varira se zadnji član podniza od prvog člana do zadnjega člana polja. U unutrašnjoj petlji varira se duljina niza od prvog člana do zadnjeg člana. Sve 3 petlje se za najgori slučaj obavljaju n puta. Zbog toga je apriorna složenost $O(n^3)$.

```
int MaxPodSumaNiza3 (int A[], int N) {
   int OvaSuma, MaxSuma, i, j, k;
   int iteracija = 0;
     MaxSuma = 0;
   for (i = 0; i < N; i++) {
            for (j = i; j < N; j++) {
                    OvaSuma = 0;
                    for (k = i; k <= j; k++) {
                            OvaSuma += A [k];
                            ++iteracija; }
                       if (OvaSuma > MaxSuma)
                            MaxSuma = OvaSuma;}
   }        return MaxSuma; }
```

- Kvadratna složenost: ako uočimo da vrijedi (POJASNI!!!)

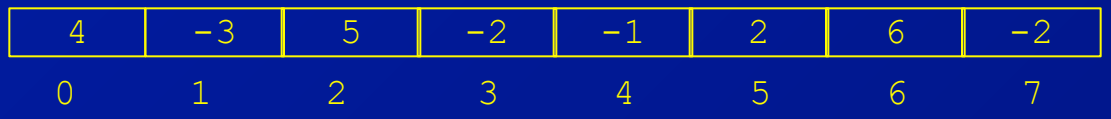
$$\sum_{k=i}^{j} A_k = A_j + \sum_{k=i}^{j-1} A_k$$

složenost se može reducirati uklanjanjem jedne petlje.
Složenost ovog algoritma je $O(n^2)$.

```
int MaxPodSumaNiza2 (int A[], int N) {
        int OvaSuma, MaxSuma, i, j;
        int iteracija = 0;
       MaxSuma = 0;
       for (i = 0; i < N; i++) {
                OvaSuma = 0;
                for (j = i; j < N; j++) {
                      OvaSuma += A[ j ];
                      ++iteracija;
                      if (OvaSuma > MaxSuma)
                            MaxSuma = OvaSuma;
                      }
               }
               return MaxSuma;
}
```

- Linearna * logaritamska složenost: $O(n \log_2 n)$
  - Relativno složeni rekurzivni postupak. Kad ne bi bilo i boljeg (linearnog) rješenja, ovo bi bio dobar primjer snage rekurzije i postupka podijeli-pa-vladaj (divide-and-conquer). Ako se ulazno polje podijeli približno po sredini, rješenje može biti takvo da je maksimalna suma u lijevom dijelu polja, ili je u desnom dijelu polja ili prolazi kroz oba dijela. Prva dva slučaja mogu biti riješena rekurzivno. Zadnji slučaj se može realizirati tako da se nađe najveća suma u lijevom dijelu koja uključuje njegov zadnji član i najveća suma u desnom dijelu koja uključuje njegov prvi član. Te se dvije sume zbroje i uspoređuju s one prve dvije. Na primjer:

Lijevi dio | Desni dio



Najveća lijeva suma je od članova 0 do 2 i iznosi 6. Najveća desna suma je od članova 5 do 6 i iznosi 8. Najveća lijeva suma koja uključuje zadnji član na lijevo je od 0 do 3 člana i iznosi 4, a najveća desno koja uključuje prvi član na desno od 4 do 6 člana i iznosi 7. Ukupno to daje sumu 11 koja je onda i najveća.

Pozivni program za početne rubove zadaje 0 i $n-1$.

```
int Max3 (int A, int B, int C) {
// racuna najveci od 3 broja: X>Y ? X:Y, A>B ? max(A,C):max(B,C)
   return A > B ? A > C ? A : C : B > C ? B : C;
}

int MaxPodSuma (int A[], int Lijeva, int Desna, int dubina) {
// trazi najvecu podsumu slijeva nadesno
   int MaxLijevaSuma, MaxDesnaSuma;
   int MaxLijevaRubnaSuma, MaxDesnaRubnaSuma;
   int LijevaRubnaSuma, DesnaRubnaSuma;
   int Sredina, i, ret;

     if (Lijeva == Desna) { // Osnovni slucaj
        if (A [Lijeva] > 0)
          ret = A [Lijeva]; // podniz od clana A[Lijeva]
     else
          ret = 0; // suma je 0 ako su svi brojevi negativni
   return ret; }
     // racun lijeve i desne podsume s obzirom na Sredina
   Sredina = (Lijeva + Desna) / 2;
   MaxLijevaSuma = MaxPodSuma (A, Lijeva, Sredina, dubina+1);
   MaxDesnaSuma = MaxPodSuma (A,Sredina + 1, Desna, dubina+1);
```

```
// najveca gledano ulijevo od sredine
   MaxLijevaRubnaSuma = 0; LijevaRubnaSuma = 0;
   for (i = Sredina; i >= Lijeva; i--) {
       LijevaRubnaSuma += A [i];
       if (LijevaRubnaSuma > MaxLijevaRubnaSuma)
              MaxLijevaRubnaSuma = LijevaRubnaSuma; }
// najveca gledano udesno od sredine
  MaxDesnaRubnaSuma = 0; DesnaRubnaSuma = 0;
  for (i = Sredina + 1; i <= Desna; i++) {
    DesnaRubnaSuma += A [i];
    if (DesnaRubnaSuma > MaxDesnaRubnaSuma)
           MaxDesnaRubnaSuma = DesnaRubnaSuma;  }
// najveca od lijeva, desna, rubna
  ret = Max3 (MaxLijevaSuma, MaxDesnaSuma,
                MaxLijevaRubnaSuma + MaxDesnaRubnaSuma);
   return ret;}

// NlogN slozenost
int MaxPodSumaNizaLog (int A [], int N) {
   return MaxPodSuma (A, 0, N - 1, 0);}
```

- Programski kod je relativno složen, ali daje za red veličine bolje rezultate od prethodnoga.
- Ako bi n bio potencija od 2 intuitivno se vidi da će sukcesivnih raspolavljanja biti $log_2\ n$. Kroz postupak prolazi n podataka, pa imamo $O(n\ log_2 n)$.
- Općenito se može reći da je trajanje algoritma $O(log_2 n)$ ako u vremenu $O(1)$ podijeli veličinu problema (obično ga raspolovi).
- Ako u pojedinom koraku reducira problem za 1, onda je njegovo trajanje $O(n)$.

- Linearna složenost: zbrajaju se svi članovi polja redom, a pamti se ona suma koja je u cijelom tijeku tog postupka bila najveća, pa je složenost algoritma O(n)

```
int MaxPodSumaNiza1 (int A[], int N) {
   int OvaSuma, MaxSuma, j;
   OvaSuma = MaxSuma = 0;
   for (j = 0; j < N; j++) {
            OvaSuma += A[j];
            if (OvaSuma > MaxSuma)
                     MaxSuma = OvaSuma;
            else if (OvaSuma < 0)
                     OvaSuma = 0;     // povećanje izgleda sljedećeg podniza
            }
   return MaxSuma;
}
```