

Rekurzija

Osnovna ideja

- Rekurzije u matematici – primjer su nizovi brojeva
- Zadana eksplicitna formula za izračunavanje općeg člana niza
$$a_n = 2^n \quad ; \quad a_n = (3^n - 1) / 2$$
- Rekurzivno zadavanje: vrijednost n-tog dana preko prethodnih članova niza
$$a_k = 2 * a_{k-1}, k \geq 2, a_1 = 2 \quad ; \quad a_k = 3 * a_{k-1} + 1, k \geq 2, a_1 = 1$$
- Aritmetički niz: a, d realni brojevi, d nije 0, $a_1 = a, a_n = a_{n-1} + d$
- Geometrijski niz: a, q realni brojevi, q nije 0 i 1, $a_1 = a, a_n = q * a_{n-1}$
- Računanje elemenata niza: uporaba vrijednosti prethodnih elemenata
- Problem traženja rješenja rekurzivne jednadžbe – eksplicitni izraz iz rekurzivnog
- Rekurzija se često koristi u računarstvu
- Procedura poziva samu sebe
- Mora postojati završetak
- Neki jezici (npr. stare verzije jezika *FORTRAN*) ne podržavaju rekurziju.
- Rekurzivni programi su kraći, ali izvođenje programa je najčešće dulje.
- Koristi se struktura podataka *stog* za pohranjivanje rezultata i povratak iz rekurzije.

- Primjer: funkcija koja računa sumu prvih n prirodnih brojeva:

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

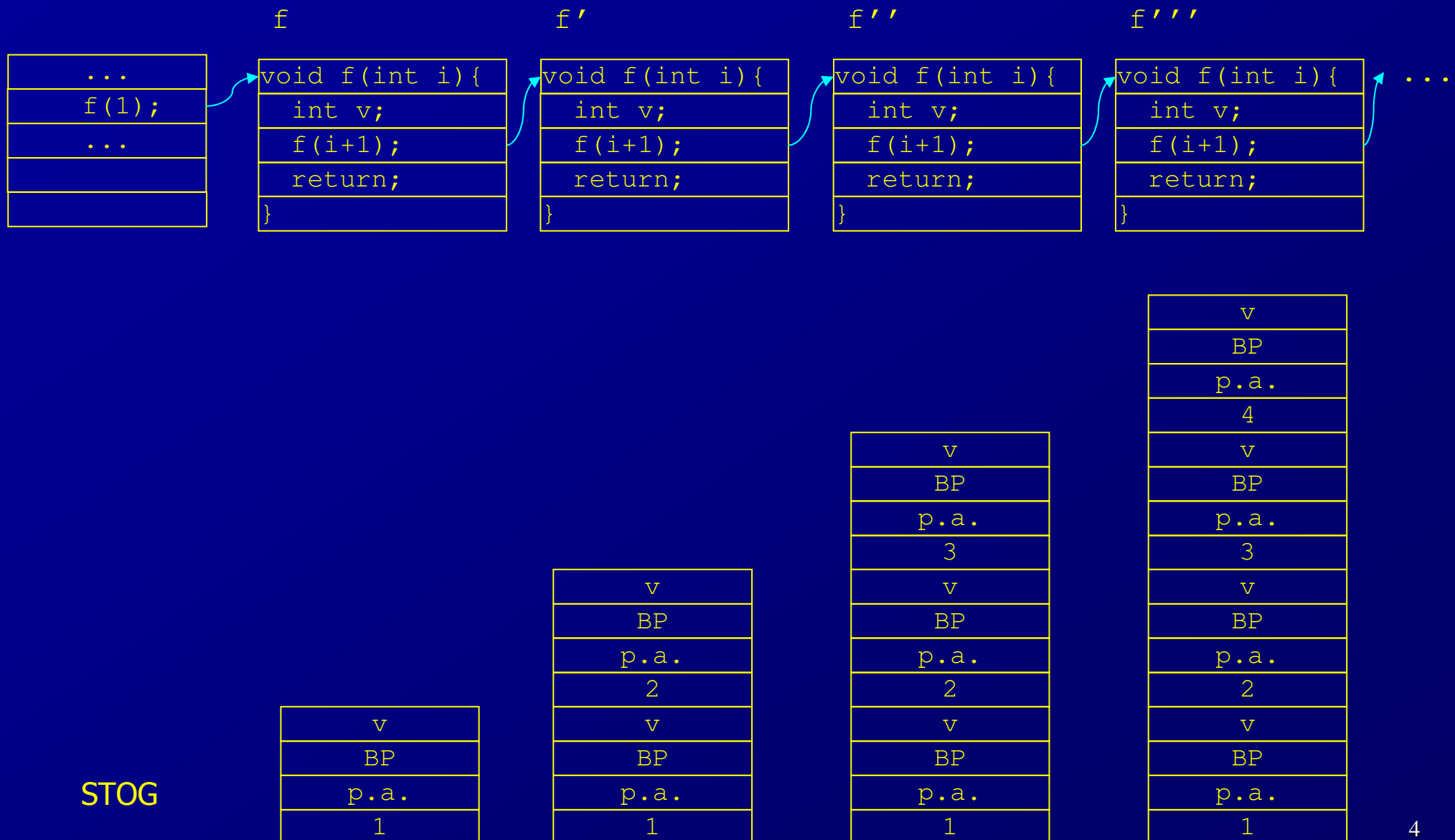
- Za $n = 5$, suma je $1 + 2 + 3 + 4 + 5 = 15$

- Drugi primjer: funkcija rezervira mjesto za polje cijelih brojeva i poziva samu sebe

```
void f (int i) {  
    int v[10000];  
    printf("%d ",i);  
    f (i+1);  
    return;  
}
```

- Izvršava se dok ne popuni dopušteni memorijski prostor za stog

Elementarna rekurzija i stog



Izračunavanje faktoriijela

- Jedan od jednostavnih rekurzivnih algoritama jest izračunavanje $n!$ za $n \geq 0$.

$$0! = 1$$

$$1! = 1$$

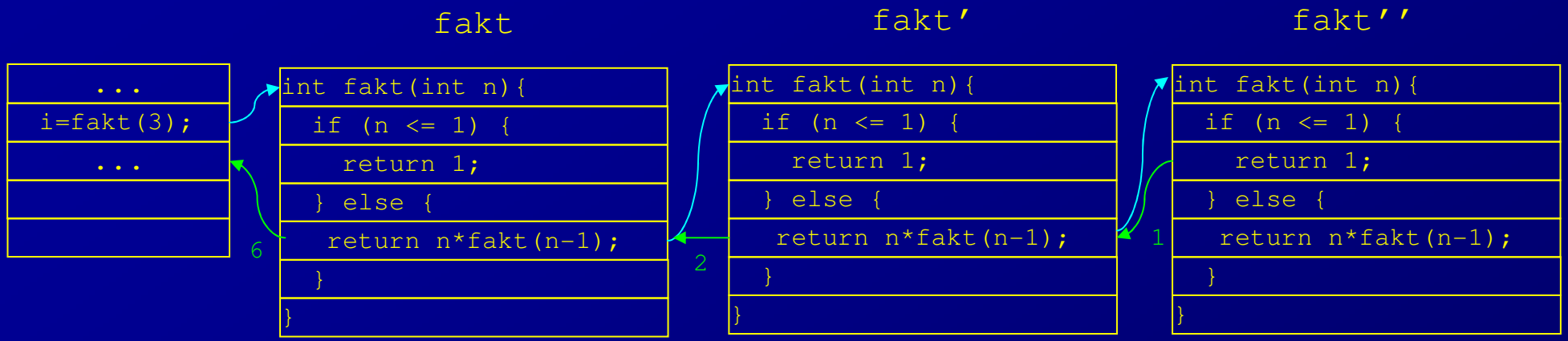
$$n! = n * (n-1)!$$

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

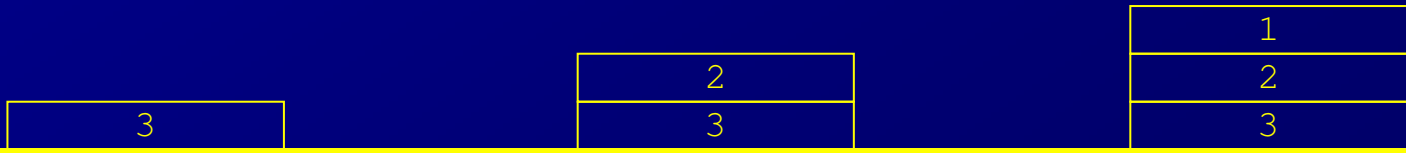
Primjer: 4!

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

Izračunavanje faktorijela



STOG (n)



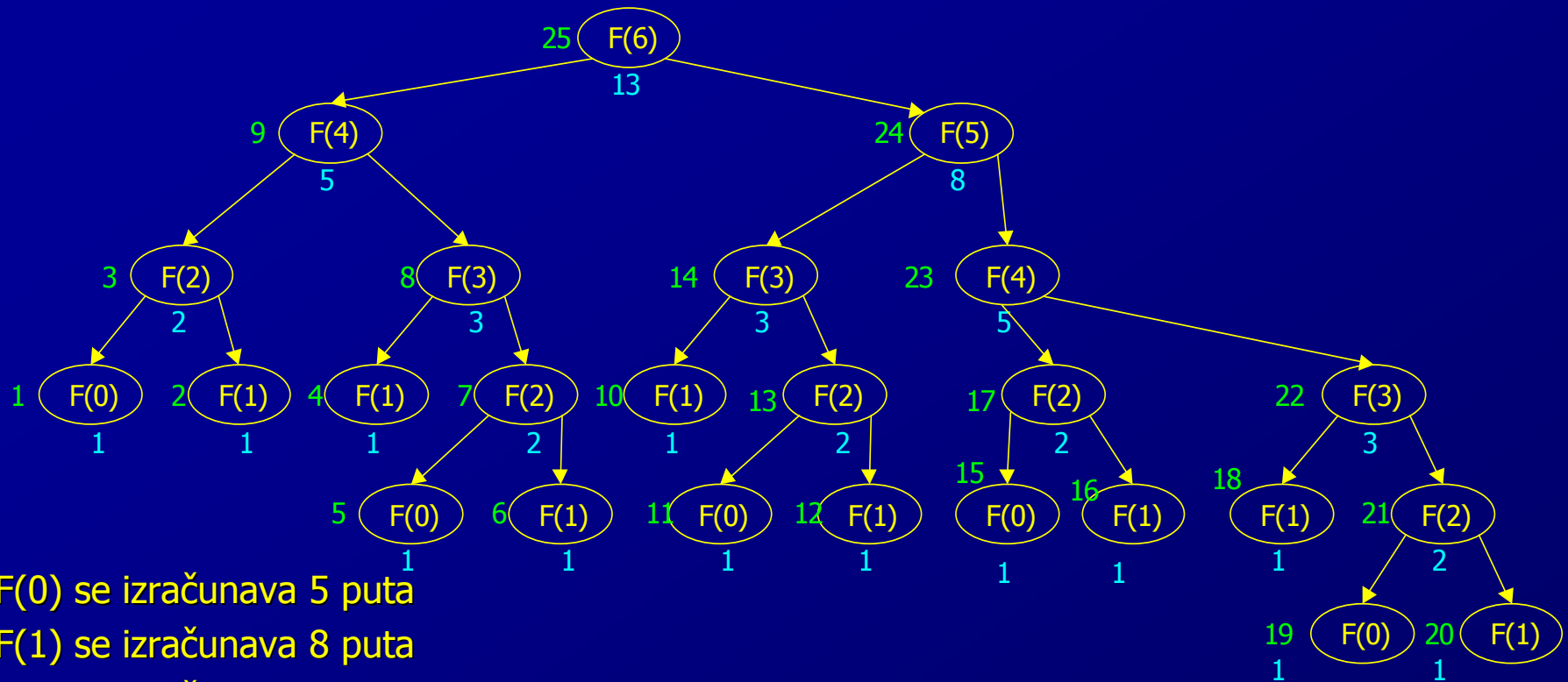
Fibonaccijevi brojevi

- U 13 stoljeću Leonardo Fibonacci iz Pise se bavio problemom razmnožavanja zečeva
- Problem: svaki par zečeva starih bar 2 mjeseca dobiju par zečića svaki mjesec, pri čemu je jedno dijete mužjak, drugo ženka. Koliko će biti zečeva nakon n mjeseci od jednog para roditelja ?
- Odgovor: broj zečeva se povećava prema ovom nizu 1, 1, 2, 3, 5, 8, 13, 21, 34,...
- $F_0 = F_1 = 1$
 $F_n = F_{n-2} + F_{n-1}; n > 1$

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

- Program je vrlo kratak i potpuno odgovara matematičkoj definiciji. Efikasnost je vrlo niska.

Fibonaccijevi brojevi



- $F(0)$ 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

Nalaženje najveće zajedničke mjere

- 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. $r = a \% b$ ($a \bmod b$, ostatak dijeljenja a s b)
3. $a = b$ i $b = r$
4. Ako je r različit od 0 idi na 2
5. Vрати a

- **Primjer:**

$$\text{nzm}(22, 8) = \text{nzm}(8, 6) = \text{nzm}(6, 2) = \text{nzm}(2, 0) = 2$$

$$\text{nzm}(21, 13) = \text{nzm}(13, 8) = \text{nzm}(8, 5) = \text{nzm}(5, 3) = \text{nzm}(3, 2) = \text{nzm}(2, 1) = \text{nzm}(1, 0) = 1$$

- **Rekurzivna funkcija:**

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

Traženje člana polja

- **Rekurzivni postupak za traženje indeksa prvog člana jednodimenzionalnog polja od n članova koji ima vrijednost x . Ako takvoga nema, rezultat je -1.**

```
int trazi (tip A[], tip 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)`.**
- **Pretraživanje je brže ako se prethodno u polje prošireno za jedan član stavi tzv. ograničivač `A[n] = x`;**

```
int trazi1 (tip A[], tip x, int i){  
    if(A[i] == x) return i;  
    return trazi1 (A, x, i+1)  
}
```

- **Poziv:** `tip i; A[n] = x; if ((i = trazi1 (A, x, 0)) == n) ...`

- **Primjer:** `A=[1,2,3,4,5]` , `x=3` ; `x=7`

`i=0,1,2 vraća 2 ; i=0,1,2,3,4 vraća -1`

`A=[1,2,3,4,5,7]`

`i=0,1,2 vraća 2 ; i=0,1,2,3,4,5 vraća 5`

Traženje najvećeg člana polja

- Određivanje indeksa najvećeg člana u polju od n članova

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

Nije zadovoljen princip strukturiranog programiranja da iz nekog modula vodi 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 slučajevi:* uvijek moraju postojati osnovni slučajevi koji se rješavaju bez rekurzije
- *Napredovanje:* Za slučajeve koji se rješavaju rekurzivno, svaki sljedeći rekurzivni poziv mora biti približavanje osnovnim slučajevima.
- *Pravilo projektiranja:* Podrazumijeva se da svaki rekurzivni poziv funkcionira.
- *Pravilo ne ponavljanja:* Ne valja dopustiti da se isti problem rješava odvojenim rekurzivnim pozivima. (To rezultira umnažanjem posla, vidi npr. Fibonaccijevi brojevi).

- Primjer za pogrešku

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

- Pogreška jest u tome što za vrijednost $n = 1$ rekurzivni poziv je opet s argumentom 1, znači nema napredovanja prema osnovnom slučaju. Program međutim ne radi niti 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 korektnost
- Nedostatci:
 - uvećano vrijeme izvođenja (najčešće)
 - 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 L_1 .
 - 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 L_i 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 svake naredbe `return` dolazi:
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 člana u polju

- pri tom je početni kod promijenjen u:

```
if (i < n-1) {  
    imax = maxclan (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;
```

```
    vrh = -1; //Pravilo 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, poznavanjem i razumijevanjem rada rekurzivne funkcije
- Jednostavno je smisliti algoritam i napraviti kod koji isti problem rješava iterativnim postupkom:

```
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;  
    return imax;  
}
```

ANALIZA SLOŽENOSTI ALGORITAMA

Pojam algoritma

- Algoritam je striktno definirani postupak koji daje rješenje nekog problema
- Nastao u matematici, danas se često koristi u računarstvu
- D. E. Knuth je naveo 5 svojstava koje algoritam mora zadovoljavati:
 - 1) konačnost – mora završiti nakon konačnog broja koraka; algoritam mora biti opisan pomoću konačnog broja operacija od kojih svaka mora biti konačne duljine
 - 2) definitnost – svaki korak algoritma mora sadržavati nedvosmislene, rigorozno definirane operacije
 - 3) ulaz – mora imati 0 ili više ulaza
 - 4) izlaz – mora imati 1 ili više izlaza
 - 5) efektivnost – mora biti takav da se može izvesti samo uz pomoć olovke i papira u konačno vrijeme
- 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: koji koristiti ?
- Potrebno je odrediti mjeru kakvoće algoritama da bi se pri izboru algoritma moglo odrediti koji je bolji
- Mjeri se količina resursa koje algoritam treba 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
- Vremenska i prostorna složenost algoritma
- Mi ćemo se pozabaviti samo vremenskom složenošću algoritama (češće se koristi u ocjenjivanju kakvoće algoritma)
- Prvi problem kod određivanja vremenske složenosti algoritma je pronalaženje mjere koja ne ovisi o brzini računala na kojem se algoritam izvodi, već samo o samom algoritmu
- Zato se vremenska složenost algoritma ne izražava vremenom potrebnim da algoritam izračuna rezultat, već brojem elementarnih operacija koje algoritam mora izvesti u izračunu

- Svrha: predviđanje vremena izračuna i pronalaženje efikasnijih algoritama
- Pretpostavke: fiksno vrijeme dohvata sadržaja memorijske lokacije, vrijeme obavljanja elementarnih operacija (aritmetičke, logičke, pridruživanje, poziv funkcije) je ograničeno nekom konstantom kao gornjom granicom
- Broj operacija koje algoritam izvodi ovisi o veličini ulaza
- Instance problema različitih dimenzija zahtijevaju različiti broj operacija
- No složenost algoritma nije funkcija samo 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, 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.
- *A posteriori* analiza je statistika dobivena mjerenjem na računalu.

Analiza a priori

- *A priori*: procjena vremena izvođenja, nezavisno od računala, programskog jezika, prevoditelja (*compiler*)
- 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++) {` n
 `x += y;`
 `}`

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

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 aproksimativna funkcija koja je jednostavnija od same funkcije složenosti, dobro asimptotski opisuje rast funkcije složenosti
- Za male n aproksimacija ne mora vrijediti (ali to je nebitno)
- Nas ovdje ne zanima stvarni iznos funkcije složenosti, već samo koliko brzo ta funkcija raste, zanima nas njen 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

O-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 sve $n \geq n_0$. Traži se najmanji $g(n)$ za koji to vrijedi.
- Ovo znači da funkcija f ne raste brže od funkcije g
- Ako je broj izvođenja operacija nekog algoritma ovisan o nekom ulaznom argumentu n oblika polinoma m -tog stupnja, onda je vrijeme izvođenja za taj algoritam $O(n^m)$.

- Dokaz:

Ako je vrijeme izvođenja određeno polinomom:

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

onda vrijedi:

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

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

$$|A(n)| \leq (|a_m| + \dots + |a_1| + |a_0|) n^m, \text{ za svaki } n \geq 1$$

Uz :

$$c = |a_m| + \dots + |a_1| + |a_0| \text{ i } n_0 = 1$$

tvrdnja je dokazana.

- Može se kao c koristiti bilo koja konstanta veća od $|a_m|$ ako je n dovoljno velik:
- Ako neki algoritam ima k odsječaka čija vremena su:

$$c_1 n^{m_1}, c_2 n^{m_2}, \dots, c_k n^{m_k}$$

onda je ukupno vrijeme

$$c_1 n^{m_1} + c_2 n^{m_2} + \dots + c_k n^{m_k}$$

- Znači da je za taj algoritam vrijeme izvođenja jednako $O(n^m)$, gdje je $m = \max\{m_i\}, i = 1, \dots, k$

Korisno:

- 1) Ako je $a < b$ onda je n^a raste sporije od n^b
- 2) Za svaka 2 broja a, b iz skupa cijelih brojeva različita od 1 vrijedi da $\log_a n$ raste jednako brzo kao $\log_b n$ (zato se često baza zanemari i piše se $\lg n$ – logaritam po proizvoljnoj bazi)
- 3) n^a raste sporije od $n^a * \lg n$ raste sporije od n^{a+1} ; iz ovoga slijedi da logaritmi asimptotski rastu sporije od linearne funkcije
- 4) a, b cijeli brojevi različiti od 1, za $a < b$ vrijedi da n^a raste sporije od b^n ; iz ovoga slijedi da eksponencijalna funkcija raste brže od bilo koje polinomne funkcije
- 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:

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

- $O(1)$ 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 slijednog 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;  
    while (i > 0) {  
        i--;  
        if (A[i] > A[imax]) imax = i;}  
    return imax;}  
}
```

petlja se uvijek obavi $n-1$ puta \rightarrow

$$\Omega(n) = O(n) = \Theta(n)$$

- Primjer: traženje člana u polju

```
int trazi (int A[], int x, int n, int i) { // A-polje x-trazeni i-indeks od kojeg se trazi  
    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. $r = a \% b$ ($a \bmod b$, ostatak dijeljenja a s b)

3. $a = b$ i $b = r$

4. Ako je r različit od 0 idi na 2

5. Vрати 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 bar 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 se definira kao duljina manjeg od ulaznih brojeva, pa slijedi

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

$$T_{\max}^{\text{Euklid}}(n) = \Theta(n)$$

Asimptotsko vrijeme izvođenja

- *Asimptotsko vrijeme izvođenja je $f(n) \sim \Theta(g(n))$ (čita se: " $f(n)$ je asimptotsko funkciji $g(n)$ ") ako je*

$$\lim_{n \rightarrow \infty} f(n)/g(n) = 1$$

- Precizniji je opis vremena izvođenja nego O-notacijom. Zna se i red veličine vodećeg člana i konstanta koja ga množi.

Ako je na primjer

$$f(n) = a_k n^k + \dots + a_0$$

tada

$$f(n) = O(n^k)$$

i

$$f(n) \sim \Theta(a_k n^k)$$

- 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)

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

- Najočiti 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 se izvode n+1 puta, u svakom koraku 1 zbrajanje, 1 množenje i 1 potenciranje. Potenciranje odgovara nizu uzastopnih množenja. Za $i > 1$ to znači i-1 množenje, što ukupno daje broj množenja

$$n-1$$

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

$$i=1$$

Zajedno je to n+1 zbrajanje i $n+1 + (n-1)*n/2$ 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 prvotnog. Postoji li još brži ? Da.

Hornerov algoritam:

- izluči li se u izrazu za polinom iz prvih n pribrojnika x

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

Ponovi li se postupak još $n-2$ puta dobiva se izraz:

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

- 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 $O(n^2)$, od drugog je dvostruko brži

Primjer za različite složenosti istog problema

- Zadano je polje cijelih brojeva A_0, A_1, \dots, 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:

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

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			
4	-3	5	-2	-1	2	6	-2	
0	1	2	3	4	5	6	7	

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 najveći 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 najveću podsumu s lijeva na desno
    int MaxLijevaSuma, MaxDesnaSuma;
    int MaxLijevaRubnaSuma, MaxDesnaRubnaSuma;
    int LijevaRubnaSuma, DesnaRubnaSuma;
    int Sredina, i, ret;

    if (Lijeva == Desna) { // Osnovni slučaj
        if (A [Lijeva] > 0)
            ret = A [Lijeva]; // podniz od člana A[Lijeva]
        else
            ret = 0; // suma je 0 ako su svi brojevi negativni
    }
    return ret; }

// račun 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;      // povecanje izgleda sljedeceg podniza
    } return MaxSuma;}

```

Analiza a posteriori

- Stvarno vrijeme potrebno za izvođenje algoritma na konkretnom računalu.

`#include <sys\timeb.h>` gdje je deklarirano

```
struct timeb {
    time_t time; // broj sekundi od ponoći, 01.01.1970 prema UTC
    unsigned short millitm; // milisekunde
    short timezone; // razlika u minutama od UTC
    short dstflag; // <>0 ako je na snazi ljetno računanje vremena
};

void ftime(struct timeb *timeptr);
```

- U programu:

```
struct timeb vrijeme1, vrijeme2; long trajanjems;
ftime (&vrijeme1);
...
ftime (&vrijeme2);
trajanjems = 1000 * (vrijeme2.time - vrijeme1.time) +
             vrijeme2.millitm - vrijeme1.millitm;
```

- *coordinated universal time (UTC):* novi naziv za *Greenwich Mean time (GMT)*