Programiranje I

Funkcije

Tip void

- Vidjeli smo da funkcija koja ne vraća vrijednost može da se deklariše kao void.
- Promjenljiva koja se deklariše kao void a nema vrijednost i gotovo se nikad ne koristi.
- Funkcija koja se deklariše da vraća void * vraća pokazivač na neodređeni tip podatka.
- Promjenljiva koja se deklariše kao pokazivač na void tj. void *b zapravo pokazuje na promjenljivu koja je nekog od mogućih tipova podataka.
- Ovo će biti jasnije kada budemo učili dinamičku alokaciju i dealokaciju memorije.

Funkcija – Neki detalji

- Unutar funkcije se mogu deklarisati pomoćne promjenljive i to na njenom početku prije bilo koje druge naredbe. Tako se funkcija zbir1 iz naših primjera mogla deklarisati kao:

```
int zbir1(int x, int y){
        int z=x+y;
        return z;
}
```

Promjenljiva z se nakon završetka funkcije dealocira – briše iz memorije.

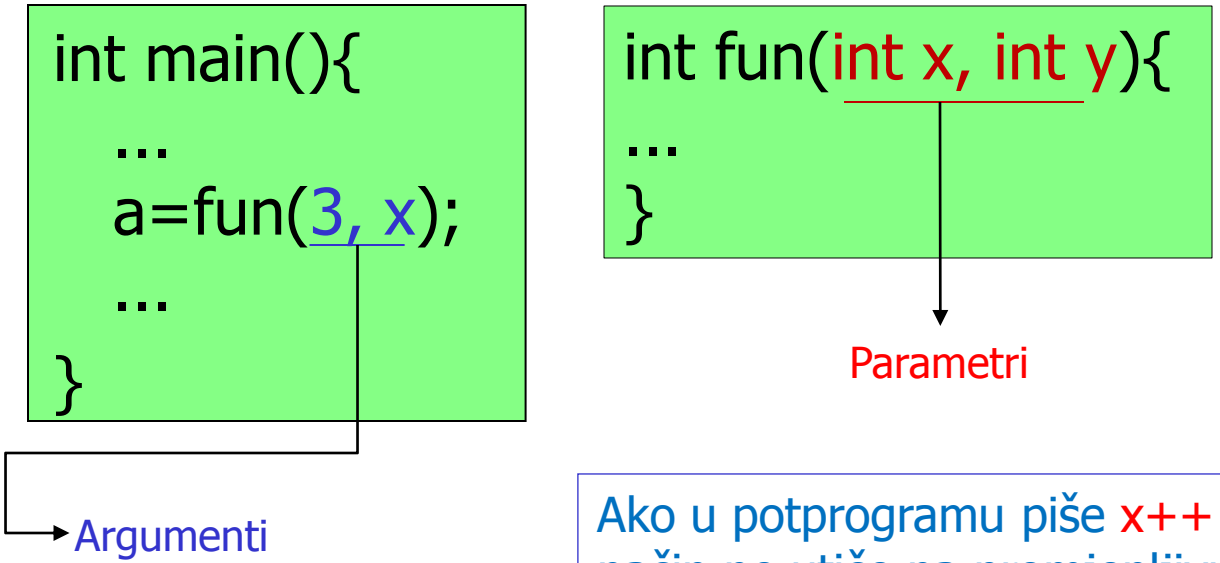
- Naredba return može da vrati vrijednost bilo kog osnovnog tipa uključujući i karakter, na primjer:

```
return 'A';
```

- Funkcije koje su tipa void mogu se završiti svojom posljednjom naredbom ili nasilno prekinuti sa return, naravno, bez argumenta nakon return.

Parametri i argumenti funkcije

- Parametri funkcije su oni koji se navode u listi argumenata u zaglavlju, dok su argumenti one vrijednosti koje su joj proslijeđene iz programa ili drugih funkcija:



Ako u potprogramu piše x++ ta promjena ni na koji način ne utiče na promjenljivu x u glavnom programu, jer su to različite promjenljive!

Prototipovi funkcija

- Da bismo osigurali provjeru poziva funkcije, tj. da se ne dozvoli poziv funkcije sa neočekivanim argumentima, prije definisanja funkcije (i prije funkcije **main**) se navodi prototip funkcije.
- Prototip liči na zaglavlje, s tim što se ne moraju navoditi imena argumenata. Prototip se završava sa ;
- Primjer prototipa:

```
int zbir(int, int);          /*ponekad se imena argumenata*/
                             /*navode samo da bi objasnila*/
                             /*značenja argumenata*/
```

Prototip - Upotreba

```
#include <stdio.h>

int fun(int);

int main() {
    printf("%d\n", fun());
    return 0;
}

int fun(int n) {
    if (n == 0)
        return 1;
    else
        return n * fun(n - 1);
}
```

Prototip funkcije

Pokušaj poziva funkcije **fun** bez argumenata dovodi do greške jer ta funkcija očekuje za argument cio broj.

Bez prototipa se ne vrši provjera prilikom poziva funkcije, pa poziv može rezultovati u besmislenim rezultatima ili „pucanju“ programa.

Poziv po referenci

- Pri radu sa nizovima, ne možemo proslijediti potprogramu svaki član niza pojedinačno.
- Stoga je programerska praksa razvila call-by-reference, gdje se potprogramu prosleđuje čitav memorijski objekat (npr. niz), a ne njegova kopija.
- Tada se promjene koje se obave nad referencom (memorijskim objektom) reflektuju na promjenljivu u glavnom programu.
- Programski jezik C ovo ne može da uradi direktnim putem, već mora da simulira preko pokazivača.

Poziv po referenci - Primjer

```
int zbir(int *x, int *y){
        (*x)++;
        (*y)++;
        return (*x)+(*y);
}
```

promjenljive x i y su pokazivači - adrese

x i y i dalje ostaju iste adrese, ali se sad mijenja (uvećava za jedan) ono što se nalazi na tim adresama. To ima efekat i na glavni program. *x je ono što se nalazi na adresi x.

Iz glavnog programa se poziv obavlja:

```
int x=1,y=2;
...
c=zbir(&x,&y);
```

argument je adresa promjenljive

Poziv po referenci – Tipična primjena

- Prethodni primjer nije tipičan kada se koristi poziv po referenci (odnosno njegova simulacija preko pokazivača).
- Dajmo sad dva mnogo karakterističnija primjera. Prvi je sumiranje članova niza:

```
int sumaniza(int *a, int n) {
        int i, s=0;
        for(i=0; i<n; i++)
                s+=a[i];
        return s;
}
```

potprogramu prosljeđujemo adresu niza, odnosno adresu njegovog prvog člana, kao i broj članova niza (kod nizova gotovo uvijek postoji jedan ovakav par argumenata funkcije)

alternativna deklaracija kod nizova
int sumaniza(int a[], int n)

članovima niza se pristupa na uobičajeni način

Svaki drugi - duplo

- U ovom primjeru ćemo svaki drugi član niza duplirati:

```
void duplo(int *a, int n){
    int i;
    for(i=0;i<n;i+=2)
        a[i]*=2;
}
```

Funkcija je deklarisana kao void jer ne vraća rezultat, ali sve promjene koje se dogode na nizu se odražavaju i na argument (niz) u glavnom programu.

- Druga tipična primjena pokazivača kod funkcija je u slučaju da funkcija treba da vrati više od jednog rezultata. Samo jedan rezultat se može vratiti direktno preko return-a, a ostali se rezultati moraju vraćati preko pokazivača.

Maksimum i pozicija maksimuma

- **Primjer.** Napisati funkciju koja vraća maksimum i poziciju maksimuma niza. Samo jedan od rezultata se može vratiti preko return-a, a drugi preko pokazivača. Moguća realizacija:

```
int max(int *a, int n, int *poz){
    int i, m=a[0];
    *poz=0;
    for(i=1;i<n;i++)
        if(a[i]>m)
            {m=a[i]; *poz=i;}
    return m;
}
```

poz je pokazivač preko kojeg se vraća pozicija maksimuma. Sami protumačite kod funkcije.

Funkcija se iz glavnog programa može pozvati kao: a=max(x,n,&p). Što su sada x, n i p?

String kao argument funkcije

- String se funkciji prosljeđuje kao argument tipa **char ***, ali sada nije potrebno naglašavati njegovu dužinu.
- Razlog je taj što funkcija implicitno zna dužinu stringa preko terminacionog karaktera.
- Na primjer, realizacija naše verzije strlen funkcije bi bila:

```
int duzinaStringa(char *s){
        int i=0;
      while(s[i++] != '\0') ;
        return i-1;}
```

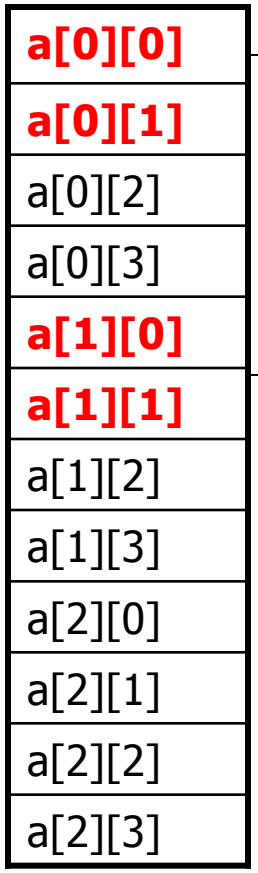
Protumačite kako ciklus koji sadrži samo praznu naredbu obavlja po nas korisnu aktivnost.

Matrica kao argument funkcije

- Na prvi pogled ne očekuje se ništa što bi razlikovalo rad sa matricama od rada sa nizovima, jer se očekuje da se elementima matrice u potprogramu pristupa preko pokazivača na isti način kao što je slučaj sa nizovima.
- Postoji, ipak, jedna sitna izmjena. Neka je u programu deklarisana matrica:
  `float m[3][4];`
- Dimenzionisanje se po pravilu vrši na najveću veličinu očekivane matrice u programu. Neka korisnik koristi dio matrice N×M=2×2.

Matrica kao argument funkcije

- Podsjetimo se kako su elementi matrice alocirani:



Kod matrice može da se dogodi da su djelovi koji se koriste razdvojeni u memoriji. Stoga funkciji treba sugerisati kakav je razmak (zato se često kaže da funkcija mora da zna kako da odbrojava po memoriji). To se može postići deklaracijom:

tip fun(float a[][4], int N, int M, ...)

Pravi broj vrsta i kolona

Argument je niz pokazivača na nizove od 4 realna broja, odnosno, kompajler zna je razmak između vrsta 4 podatka tipa float

Eventualno dodatni argumenti funkcije. Napominjemo da postoji operator ... koji sugeriše da funkcija ima proizvoljan broj argumenata, ali ga mi nećemo koristiti, već ovo znači da funkcija ima još argumenata.

Matrica - argument funkcije

- Alternativna deklaracija je:
  tip fun(float (*a)[4], int N, int M, ...)
- Poziv naše funkcije **fun** u programu bi bio:
  fun(a,2,2, ...)
- Rad sa matricama u funkcijama očigledno nosi određene probleme, ali sa dosta opreza nije problem raditi ni sa ovim tipom argumenata funkcije.
- Vidjeli smo da funkcija može da vrati rezultat koji je standardnog tipa podataka uključujući modifikacije tipa unsigned, long, itd.
- Funkcija može da vrati i pokazivač.

Složene deklaracije funkcija

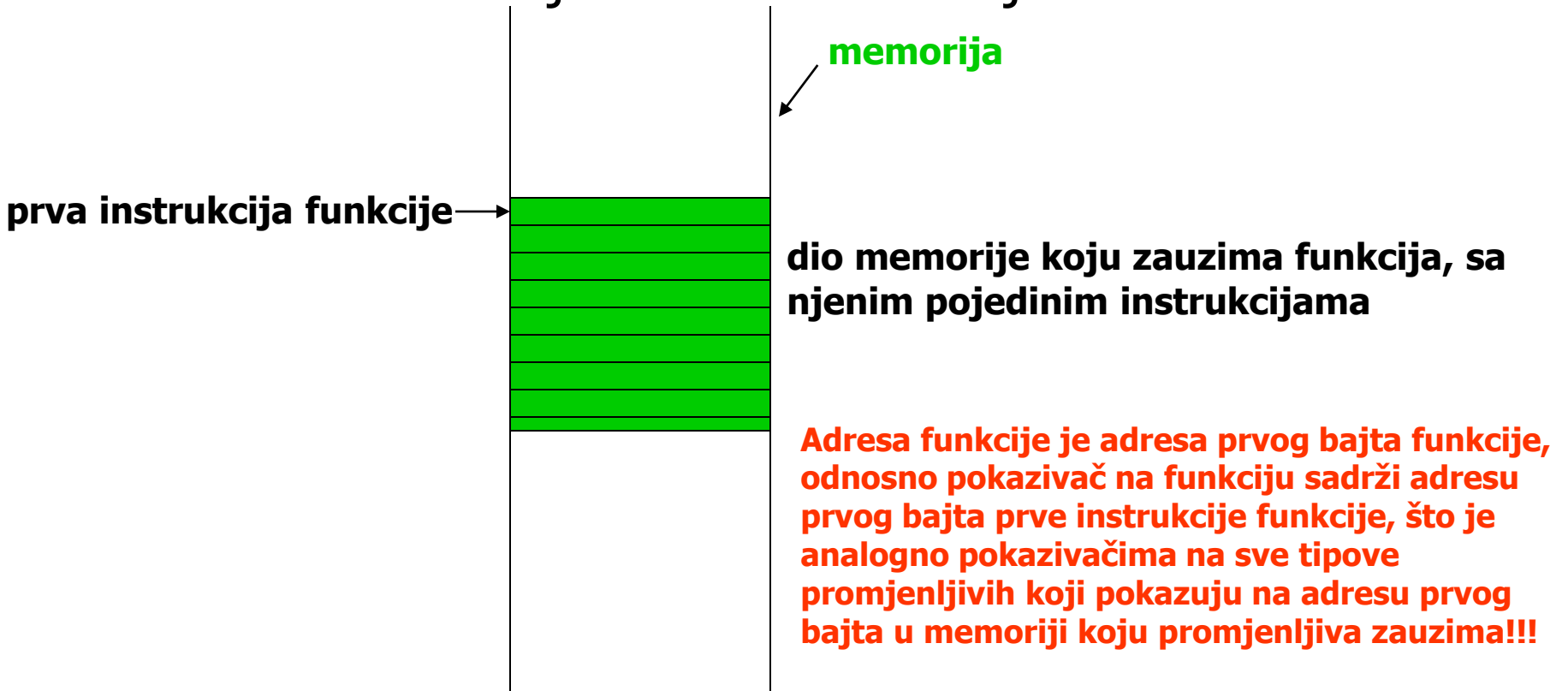
- int *f() čitamo kao funkciju koja vraća pokazivač na cijeli broj.
- Ovo zahtjeva veliki oprez u radu, jer promjenljiva na koju dati pokazivač pokazuje ne smije da bude privremena i da nestane izlaskom iz funkcije.
- Dakle, rezultat rada funkcije koja vraća pokazivač mora da bude adresa promjenljive koju koristi funkcija (ili glavni program) koja je pozvala predmetnu funkciju.
- Česta je greška da se u funkciji deklariše niz i da funkcija vraća pokazivač na taj niz. Izlaskom iz funkcije, dealociraju se sve promjenljive alocirane u funkciji.
- Mi ćemo, kad god je to moguće (a neće biti uvijek moguće), izbjegavati ovakve funkcije.

Složene deklaracije funkcija

- Kombinacija niz-pokazivač-funkcija dovodi do veoma komplikovanih deklaracija.
- Pored pravila koja se koriste kod nizova uvodi se dodatno pravilo:
  - Par [] ima isti prioritet kao par ().
- Primjer. **Zanemarimo za trenutak argumente funkcije** int (*f) ()
  - U prvom koraku eliminišemo desnu zagradu; dakle zaključujemo da je ***f** funkcija koja vraća cijeli broj.
  - U drugom koraku eliminišemo * i dolazimo da toga da je **f** pokazivač na cjelobrojnu funkciju, odnosno adresa cjelobrojne funkcije.
- Pogledajte ostale primjere iz knjige.

Adresa funkcije

- Svi elementi programa zauzimaju memoriju.
- Tako i kod funkcije zauzima memoriju.



Adresa funkcije je adresa prvog bajta funkcije, odnosno pokazivač na funkciju sadrži adresu prvog bajta prve instrukcije funkcije, što je analogno pokazivačima na sve tipove promjenljivih koji pokazuju na adresu prvog bajta u memoriji koju promjenljiva zauzima!!!

Adresa funkcije

- Adresa funkcije koja se zove **fun()** je **&fun**, ali se može koristiti i samo **fun**.
- Ovo je slično adresi niza, kod kojeg je samo ime adresa **(ponekad se kaže pristupna staza)** za prvi element niza.
- Najvažnija primjena adresa funkcija je u proslijeđivanju funkcija kao argumenta drugih funkcija.
- Naime, ponekad je zgodno omogućiti jednoj funkciji da poziva više drugih funkcija u zavisnosti od situacije.

Funkcija sa argumentom funkcijom

- Zamislimo sljedeću situaciju. Napisali smo funkciju koja na osnovnu nekog sofisticiranog metoda računa integral funkcije **sin(x)**.
- U slučaju da nam treba funkcija koja računa integral kosinusa morali bi da napišemo (kopiramo) prethodnu funkciju za računanje i da editujemo funkciju koja je argument (**cos(x)**).
- Ovaj postupak bi morali obaviti za svaku funkciju za koju želimo da nađemo integral.
- Mala greška u našem početnom dizajnu dovodi do potrebe da prepravljamo sve funkcije, što je neprihvatljivo.

Funkcija sa argumentom funkcijom

- Drugi problem koji postoji kod pretpostavljenog rješenja je situacija kada želimo da našu funkciju komercijalizujemo.
- Korisniku bismo morali omogućiti da primjeni naš algoritam na bilo koju funkciju, a to bi značilo da korisnik mora biti upoznat sa našim kodom.
- Ovim bismo zapravo potpuno razotkrili svoju programersku vještinu i u potpunosti ugrozili komercijalizaciju našeg programa.
- Umjesto toga, naša funkcija treba da ima kao argument funkciju i da korisnik, pa i mi, sa istom funkcijom realizuje svaki projekat, a jedino znanje koje je tada potrebno o našoj funkciji je njeno zaglavlje (lista i tipovi argumenata).

Primjer

- Pretpostavimo sljedeći jednostavan problem (u svakom slučaju jednostavniji nego što je postavka problema o kojem smo diskutovali).
- Želimo da sumiramo niz:

$$\sum_{i=0}^{n-1} f(a[i])$$

gdje je **a[i], i=0,…,n-1** dati niz, recimo, cijelih brojeva, a **f()** proizvoljna funkcija.

Primjer - Realizacija

```
#include <stdio.h>
int sumafunkcija(int (*)(),int*,int);
int linija(int);
int kvadrat(int);
```

pretprocesor i prototipovi funkcija

```
int linija(int n){ return n;}
int kvadrat(int n) {return n*n;}
int sumafunkcija(int (*r)(),int *a,int n){
int i,s=0;
for(i=0;i<n;i++)
     s+=r(a[i]);
return s;
}
```

realizacija funkcija - obratite pažnju na funkciju sumafunkcija, a posebno na način sumiranja

```
main(){
int a[5]={1,2,2,3,1};
int b=sumafunkcija(&linija,a,5);
int c=sumafunkcija(kvadrat,a,5);
printf("%d %d",b,c);
}
```

glavni program sa pozivima funkcije - obratite pažnju na dio koji je podvučen i pokušajte da odgovorite na pitanje zašto rade ispravno obije varijante i zašto se nigdje ne naglašava da je podvučeni argument funkcija

Funkcija main

- Glavni program ima oblik bilo koje druge funkcije - ima zaglavlje, tijelo funkcije koje započinje sa sekcijom za deklaraciju, nastavlja se naredbama i završava sa return.
- Postavlja se pitanje - kome funkcija main vraća rezultat?
- Funkcija main, kao i svaka druga funkcija, vraća rezultat onome ko je tu funkciju pozvao, a u ovom slučaju je to operativni sistem.
- UNIX/LINUX operativni sistemi lijepo rade sa ovakvim rezultatima, koji obično sugerišu da je funkcija sa uspjehom obavila posao.

Argumenti f-je main

- Do sada smo koristili funkciju main bez argumenata. Međutim, ova funkcija može imati argumente. Ta dva argumenta su uvijek cijeli broj i niz pokazivača na karaktere (niz stringova).
- Ako funkcija main ima argumente, onda se deklariše kao:

```
main (int argc, char *argv[])     /* funkcija može biti int, void */
                                  /* ili nekog drugog tipa,  */
                                  /* dok se drugi argument može */
                                  /* deklarisati i kao char **argv */
```

- Postavljaju se pitanja - što su ovi argumenti i ko ih zadaje (kod drugih funkcija argumente postavlja funkcija koja poziva datu funkciju)?

Argumenti f-je main

- I kod funkcije main argumente zadaje onaj ko je poziva, a to je, u ovom slučaju, operativni sistem.
- Naš napisan program se, kada je korektno preveden, nalazi u memoriji računara u vidu EXE fajla, npr. IME.EXE. Pozivanje se može obaviti samo navođenjem imena IME.
- Zamislimo sada situaciju da je program pozvan sa:

  IME Aa Bb Cc 11

Argumenti f-je main

- U slučaju sa prethodnog slajda, vrijednost argc će biti argc=5, dok će pojedini stringovi biti
  - argv[0]="IME"
  - argv[1]="Aa"
  - argv[2]="Bb"
  - argv[3]="Cc"
  - argv[4]="11"
- Za vježbu napisati program koji sa komandne linije učitava ime programa i dva stringa. Program vrši njihovo nadovezivanje i štampanje.
- Za vježbu napisati program koji sa komandne linije učitava ime programa i niz cijelih brojeva. Potrebno je izvršiti štampanje ovog niza.

Rekurzija

- Savremeni programski jezici dozvoljavaju da funkcija pozove samu sebe **(ovo je osobina programskog jezika C, a nije nekih starijih jezika, kao što je npr. FORTRAN).**
- Rekurzivno se funkcija faktorijel može realizovati korišćenjem pravila: n!=n*(n-1)! i 0!=1.

```
int fakt(int n){
    if(n==0) return 1;
    return n*fakt(n-1);
}
```

Za n=5, funkcija vraća rezultat 5*fakt(4), poziva se funkcija fakt(4) koja vraća rezultat 4*fakt(3), funkcija fakt(3) vraća 3*fakt(2), funkcija fakt(2) vraća 2*fakt(1), funkcija fakt(1) vraća 1*fakt(0), a fakt(0) vraća 1.

Podsjetite se priče o steku i alokacionom zapisu (prva prezentacija). Pretpostavimo da glavni program poziva fakt(5), tada će u jednom trenutku na steku biti 6 jedinica.

Rekurzija

- Rekurzivna rješenja su veoma elegantna. Neki programerski problemi se veoma teško rješavaju bez korišćenja ovog postupka.
- Rekurzija može biti memorijski zahtjevna (što se dešava prilikom poziva fakt(120)), ali i vremenski, jer operacije sa stekom i alokacionim zapisom traju neko vrijeme.
- Stoga rekurzivno treba rješavati one probleme kod kojih je prirodno korisiti rekurzivni pristup (binarno pretraživanje, quick sort itd.).
- Kao primjer loše rekurzije, realizovati rekurzivnim putem računanje Fibonacci-evih brojeva, za koje važi FIB(1)=1, FIB(2)=1 i FIB(I)=FIB(I-1)+FIB(I-2) za I>2.
- Program realizujte rekurzivno i odredite broj pozivanja funkcije FIB za I=6 ili I=15. Zatim dajte alternativno iterativno rješenje.