

RI203

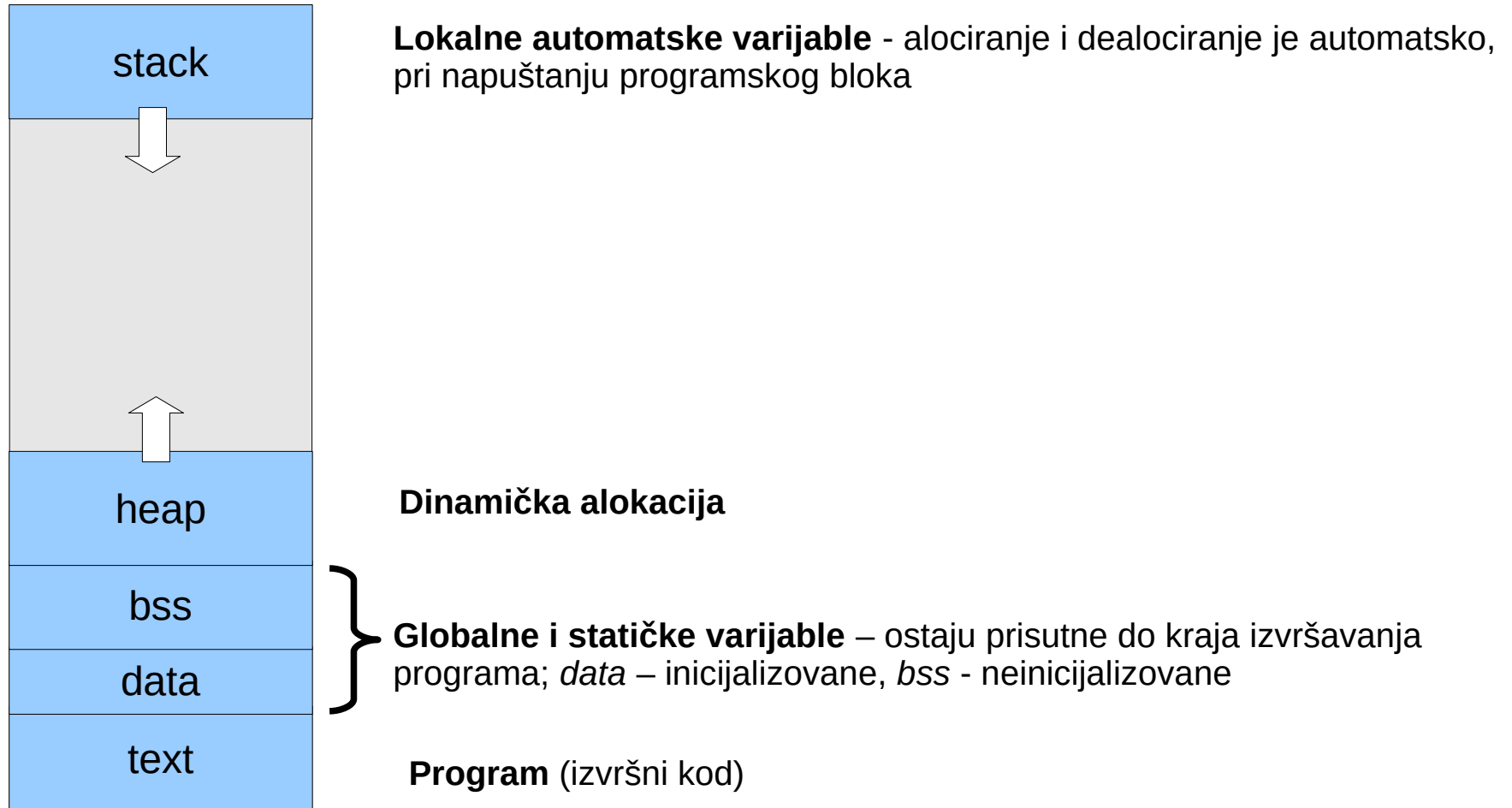
# Uvod u računarske algoritme

dr.sc. Edin Pjanić

# Pregled predavanja

- Poziv funkcije i sistemski stack procesa
- Rekurzija (engl. recursion)

# Memorija izvršnog programa (procesa)



# Stack frame

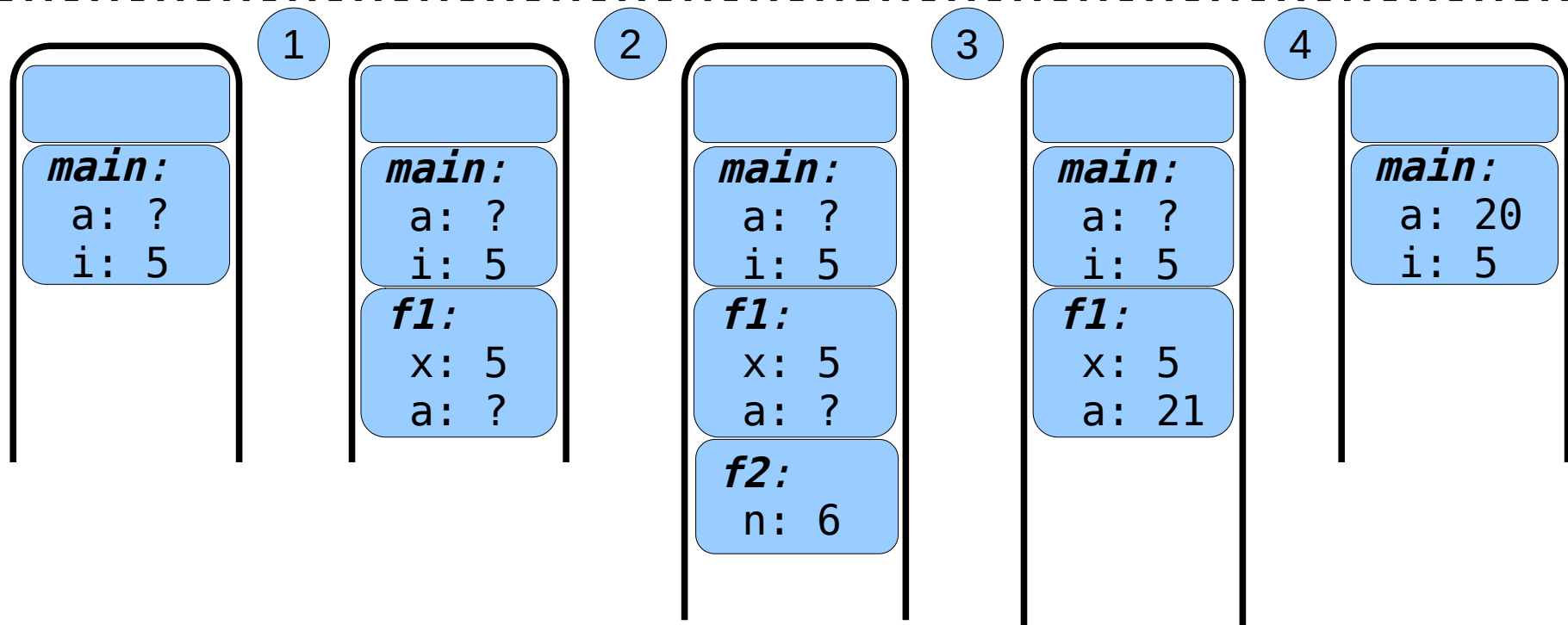
- Lokalni objekti svake funkcije se smještaju na sistemski stack.
- Svaki poziv funkcije ima svoj tzv. **stack frame** za lokalne objekte (varijable). Funkciji su dostupni samo objekti na njenom stack frameu i globalni objekti.
- Prilikom pozivanja funkcije parametri se kopiraju na stack u odgovarajuće lokalne varijable koje služe za prihvatanje parametara.
- Prilikom povratka iz funkcije, sa stacka se uklanjaju njeni lokalni objekti.

# Sistemiški stack pri pozivu funkcije

```
int main()
{
    int a, i=5;
    a = f1(i);
    cout << a;
    return 0;
}

int f1(int x)
{
    int a;
    a = (x+2) * f2(x+1);
    return a-1;
}

int f2(int n)
{
    return n-3;
}
```



STANJE SISTEMSKOG STACKA

# Rekurzija – prisjetimo se

- Rekurzija (imenica) – vidi pod rekurzija
- Rekurzija u matematici i računarstvu je metoda definisanja funkcija pri čemu se unutar definicije funkcije koristi i funkcija koja se definiše.
- Rekurzija je usko povezana sa algoritmima baziranim na paradigmi “divide and conquer” (“podijeli pa savladaj”).



# Rekurzija – prisjetimo se

- Rekurzivan proces se može definisati preko dvije karakteristike:
  - osnovni slučaj, obično jednostavan (prekid rekurzije)
  - skup pravila koja sve ostale slučajeve reduciraju i vode ka osnovnom slučaju
- Poznati primjeri rekurzivnih definicija:
  - Faktorijel:  $n! = n \cdot (n-1)!$
  - Fibonaccijevi brojevi:  $F(n) = F(n-1) + F(n-2)$
  - Fraktali



In nature:



In geometry:



In algebra:



# Rekurzivan algoritam – karakteristike

- Rekurzivan algoritam mora implementirati tri karakteristike:
  - Rekurzivan algoritam mora pozivati samog sebe.
  - Rekurzivan algoritam mora imati osnovni slučaj.
  - Rekurzivan algoritam mora mijenjati svoje stanje tako da se kreće ka osnovnom slučaju.



# Rješavanje rekurzivnih problema

- Mnogi složeni problemi se mogu riješiti rekurzijom.
- Princip pri rješavanju takvih problema je:
  - definisati problem reduciranjem na isti problem nižeg reda koji se približava osnovnom slučaju
  - definisati osnovni slučaj koji se rješava bez rekurzije tako da sprečava beskonačnu petlju i vodi ka konačnom rješenju
- Treba krenuti od cjelokupnog problema (reda  $n$ ), napisati rekurzivno rješenje na ispravan način i onda... vjerovati u rekurziju :)
- Često je proces detaljne analize poziva funkcija rekurzivnog rješenja previše složen.

# Rješavanje rekurzivnih problema u praksi

- Razraditi nekoliko slučajeva i upoznati se sa problemom .
- Identifikovati osnovni slučaj (to je onaj najjednostavniji).
- Utvrditi na koji način izvršiti redukciju problema.
  - Kako problem većeg reda ovisi od manjeg?
- Opisati (napisati) rekurzivno rješenje u opštem obliku.
- Napisati programski kod.

# Primjer: Izračunavanje faktoriјela

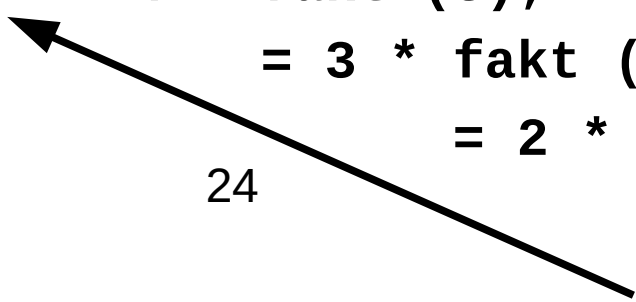
- jedan od jednostavnih rekurzivnih algoritama jeste izračunavanje  $n!$  za  $n \geq 0$

- $0! = 1$
- $1! = 1$
- $n! = n * (n-1)!$

```
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
```



24

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

# Izračunavanje faktoriјela

main

```
...  
i=fakt(3);  
...
```

6

fakt

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

fakt'

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

2

fakt''

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

1

fakt

n=1

fakt

n=2

fakt

n=3

main

i= 6

# Fibonaccijevi brojevi

- U zapadnu nauku uveo Leonardo Pisano Bigollo (oko 1170. – oko 1250.), mada su bili poznati u indijskoj nauci.
- Povezani su sa mnogim pojavama u prirodi: rast i raspored listova, plodova, broj populacije, zlatni presjek...
- Fibonaccijevi brojevi:  
0, 1, 1, 2, 3, 5, 8, 13, 21, 34,...
- Definicija:
  - $F_n = F_{n-2} + F_{n-1}$ , za  $n > 1$
  - $F_0 = 0$ ,  $F_1 = 1$

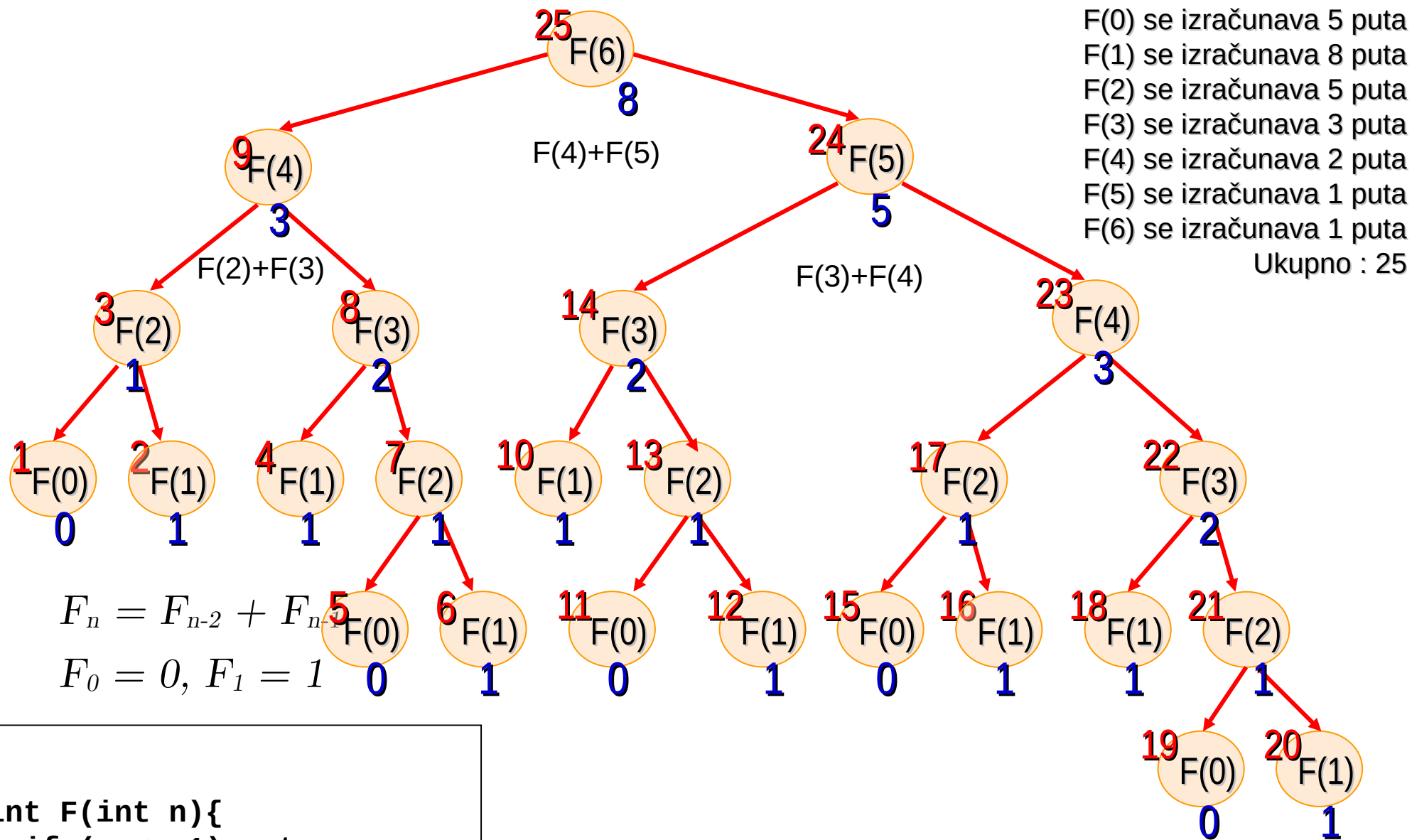
# Fibonačijevi brojevi – C/C++ funkcija

Definicija sekvenca brojeva:

- $F_n = F_{n-2} + F_{n-1}$ , za  $n > 1$
- $F_0 = 0$ ,  $F_1 = 1$

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

# Fibonaccijevi brojevi – stablo izvođenja programa



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

Algoritam je neefikasan:  $O(2^n)$

# Gornja granica za vrijeme od $F(n)$

Za  $n=0$  i  $n=1$  :  $T(0)=1 \Rightarrow O(1)$

Za  $n>1$ :  $T(n) = T(n-2) + T(n-1) + 4$

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

Uzmimo da je  $c = 4$ . Uvažimo da je  $T(n-2) \leq T(n-1)$

Tražimo gornji granicu  $\Rightarrow$  aproksimiraćemo  $T(n-2) \sim T(n-1)$

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + c \\ &= 2T(n-1) + c \quad // \text{ iz aproksimacije } T(n-1) \sim T(n-2) \\ &= 2*(\mathbf{2T(n-2) + c}) + c \\ &= 4T(n-2) + 3c = 4T(\mathbf{2T(n-3)+c}) + 7c \\ &= 8T(n-3) + 7c = 8T(\mathbf{2T(n-4)+c}) + 15c \\ &= 16T(n-4) + 15c = 16T(\mathbf{2T(n-5)+c}) + 15c \quad \text{itd.} \\ &= 2^k * T(n - k) + (2^k - 1)*c \end{aligned}$$

$T(0) = 1$ , stoga pronadimo  $k$  za koje je  $(n-k)=0 \Rightarrow \mathbf{k=n}$

Tada imamo da je

$$T(n) = 2^n * T(0) + (2^n - 1)*c = 2^n * (1 + c) - c$$

$T(n)$  je  $\mathbf{O(2^n)}$

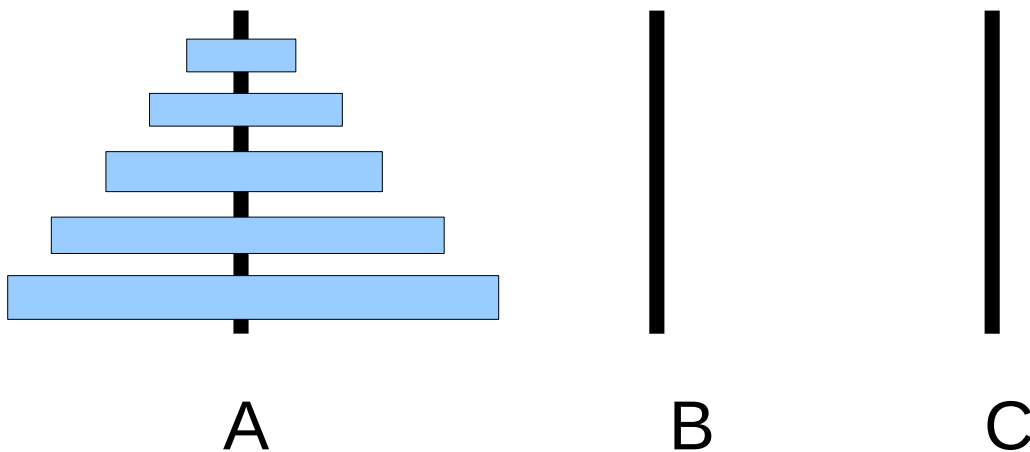




pagoda

# Tornjevi Hanoja

- Postoje 3 uspravna štapa. Na prvom štapu je **n** diskova različite veličine postavljenih tako da veći nikad ne dolazi iznad manjeg. Uz minimalni broj operacija preselite sve diskove na drugi štap, jedan po jedan.
- Disk se smije postaviti ili na prazan štap ili tako da je manji disk na većem.



Rješenje je implementirano u pratećem kodu, klasa Hanoj. Kao štapovi sa diskovima se koriste tri vektora.

Metod koji vrši rješavanje je prebaci().

Oprez: sjetimo se da je složenost  $O(2^n)$ !

Razmišljamo:

Eh, kad bismo mogli prebaciti sve diskove osim najvećeg (donjeg) na štap C pa onda prebaciti najveći disk na štap B, i onda vratiti one sa štapa C ponovo na najveći disk koji je na štapu B...

*Pa to i jeste rješenje –  
rekurzivno!!!*

# Tornjevi Hanoja - algoritam

- Algoritam rekurzivnog rješenja datog problema:

A - stap sa koga se treba prebaciti n diskova

B - stap na koji treba prebaciti te diskove

C - treći stap, može se koristiti kao pomoćni

n - broj diskova

*Prebacivanje n diskova sa štapa A na stap B uz korištenje štapa C kao pomoćnog:*

- Ako prebacujemo jedan disk (tj.  $n == 1$ ):
  - Prebaci taj jedan disk sa štapa A na B
  - Kraj postupka.
- Ako nije jedan disk nastavi.
- Prebaci  $n-1$  diskova sa A na C uz korištenje B kao pomoćnog
- Prebaci jedan disk sa štapa A na B
- Prebaci  $n-1$  diskova sa C na B uz korištenje A kao pomoćnog

# Tornjevi Hanoja - algoritam

- Pseudokod rekurzivne funkcije koja simulira dati problem:

A - stap sa koga se treba prebaciti n diskova

B - stap na koji treba prebaciti te diskove

C - pomocni (treci) stap

n - broj diskova

```
prebaci(A, B, C, n)
{
    if n == 1
    {
        prebaci_jedan(A, B)
        kraj
    }
    prebaci(A, C, B, n-1)  <- Prebaciti sve osim zadnjeg (n-1 diskova)
                           sa A na C
    prebaci_jedan(A, B)    <- Prebaciti zadnji-najveci sa A na B
    prebaci(C, B, A, n-1)  <- Prebaciti sve (n-1 diskova) sa C na B
}
```

Ovaj algoritam je implementiran u metodu `prebaci()` klase `Hanoj`.  
Cjelokupan kod se nalazi u fajlu `hanoj.cpp`.

# Tornjevi Hanoja – broj operacija i složenost

- Označimo sa  $T(n)$  broj operacija u zavisnosti od  $n$ , gdje je  $n$  broj naslaganih diskova koji se trebaju premjestiti sa jednog na drugi štap.
- Pri  $n=1$  imamo jednu operaciju premještanja, tj. jedan pop i jedan push, tj.  $T(1)=1$ , što se vidi u privatnom metodu `prebaci()` klase `Hanoj`
- U istom metodu se vidi da za svaki sljedeći korak imamo:  
$$T(n) = T(n-1) + 1 + T(n-1) = 2T(n-1) + 1$$
, pa je:  
$$T(2) = 2*1+1=3, T(3)=2*T(2)+1=2*3+1=7, \text{ itd.}$$
- Ako izraz napišemo na drugi način imaćemo:  
$$T(2)=2^2-1=3$$
 moći ćemo napisati  
$$T(3)=2*T(2)+1 = 2*(2^2-1)+1=2^3-2+1=2^3-1$$
  
$$T(4)=2*T(3)+1 = 2*(2^3-1)+1=2^4-2+1=2^4-1, \text{ itd.}$$
- Vidimo da je broj operacija  $T(n)=2^n-1$  (dokazujemo mat. indukcijom)
- Iz ovoga slijedi da je složenost  $O(2^n)$
- Koliko bi trajalo premještanje 10, 20 ili 30 diskova?



# Prednosti i mane rekurzije

## Prednosti rekurzije

- Pogodno za rješavanje problema koji su prirodno rekurzivni.
- Kod može biti čitljiviji i kraći.
- Pogodno za rad sa rekurzivnim strukturama podataka.

## Mane rekurzije

- Može biti veoma memorijski i vremenski zahtjevno (stek).
  - može iscrpiti memoriju računara
- Detaljna analiza izvršavanja koda može biti komplikovana.
- Može imati preveliku složenost ako se ne implementira ispravno.

# Repna rekurzija (*engl. tail recursion*)

- Posebna vrsta rekurzije u kojoj je zadnja operacija (komanda) u funkciji vraćanje vrijednosti od rekurzivnog poziva te funkcije.
- Pseudokod:

```
povr_tip rek_funkcija(...)  
{  
    ...  
    return rek_funkcija(...)  
}
```

Vraća se samo povratna vrijednost  
pozvane funkcije, a ne koriste se  
lokalne varijable.

## Eliminacija repne rekurzije (**tail recursion elimination**)

Kompajler može optimizirati ovakav poziv na način da se za rekurzivni poziv ne pravi novi stek okvir (stack frame), te se ustvari rekurzivni poziv potpuno eliminiše.

Ovakav vid optimizacije se može vršiti kod svih repnih poziva funkcije, bez obzira da li su rekurzivni ili ne.

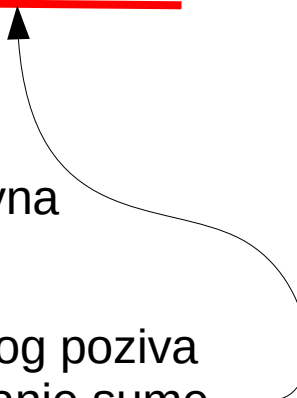
# Repna rekurzija: primjer

- Primjer rekurzivne funkcije koja računa sumu  $n$  prirodnih brojeva:

```
long g(long n){  
    if(n==0) return 0;  
    return n + g(n-1);  
}
```

NIJE repno rekurzivna

Jer poslije rekurzivnog poziva  $g(n-1)$  imamo računanje sume.



```
long f(long s, long n){  
    if(n==0) return s;  
    return f(s+n, n-1);  
}
```

JESTE repno rekurzivna

Jer poslije rekurzivnog poziva  $f(s+n, n-1)$  nemamo operacija.