Algoritmi i strukture podataka

- predavanja -

3. Rekurzija

Osnovna ideja rekurzije

- Rekurzija:
  jednadžba ili nejednadžba koja opisuje funkciju korištenjem njezine vrijednosti izračunate za manji skup podataka
- funkcija poziva samu sebe, ali definicija ne smije biti cirkularna (tj. mora imati završetak)
- procedura poziva samu sebe:

rekurzija:
vidi: rekurzija

rekurzija:
ako nije jasno što je to,
vidi: rekurzija

Implementacija

- rekurzivni programi (programski kod) su kraći, ali je izvođenje programa dulje
- za pohranjivanje rezultata i povratak iz rekurzije koristi se struktura podataka stog

Rekurzivna definicija

- **Rekurzivna definicija funkcije:** funkcija poziva samu sebe, ali definicija ne smije biti cirkularna (tj. mora imati završetak)
- Primjer: rekurzivna definicija skupa prirodnih brojeva $\boldsymbol{N}$
  1. $1 \in \boldsymbol{N}$
  2. ako je $n \in \boldsymbol{N}$,
     onda je $n + 1 \in \boldsymbol{N}$
- Primjer:
  - Zašto se dug napravljen kreditnom karticom ne može platiti istom kreditnom karticom?

Primjeri rekurzija

- faktorijeli:

$$n! = n \cdot (n - 1)! = n \cdot (n - 1) \cdots 2 \cdot 1$$

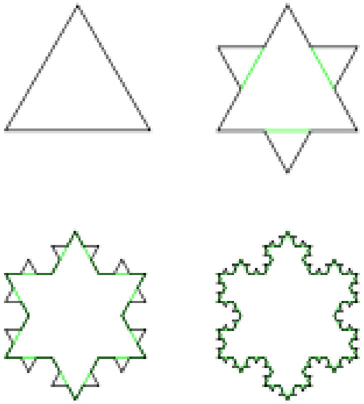
- Fibonaccijevi brojevi:

$$F_n = F_{n\text{-}1} + F_{n\text{-}2}$$

- zlatni omjer (zlatni rez):

$$\phi = 1 + 1 / \phi = (1 + 1/ (1 + 1/ \ldots)) = (1 + \sqrt{5}) / 2 \approx 1.618$$

- Kochova pahuljica (Helge von Koch, 1904.)



https://en.wikipedia.org/wiki/Recursive_definition#/media/File:KochFlake.svg

Rješavanje rekurzije (1)

- koristi se strategija ***podijeli, pa vladaj*** (lat. *divide et impera,* engl. *divide and conquer*)
  - pripisuje se Filipu II Makedonskom, ali su istu ideju koristili Julije Cezar (također i *divide ut regnes*), Machiavelli, Napoleon, itd. (wiki)

- u kontekstu rekurzije:
  početni se problem podijeli u manje, lakše rješive potprobleme

Rješavanje rekurzije (2)

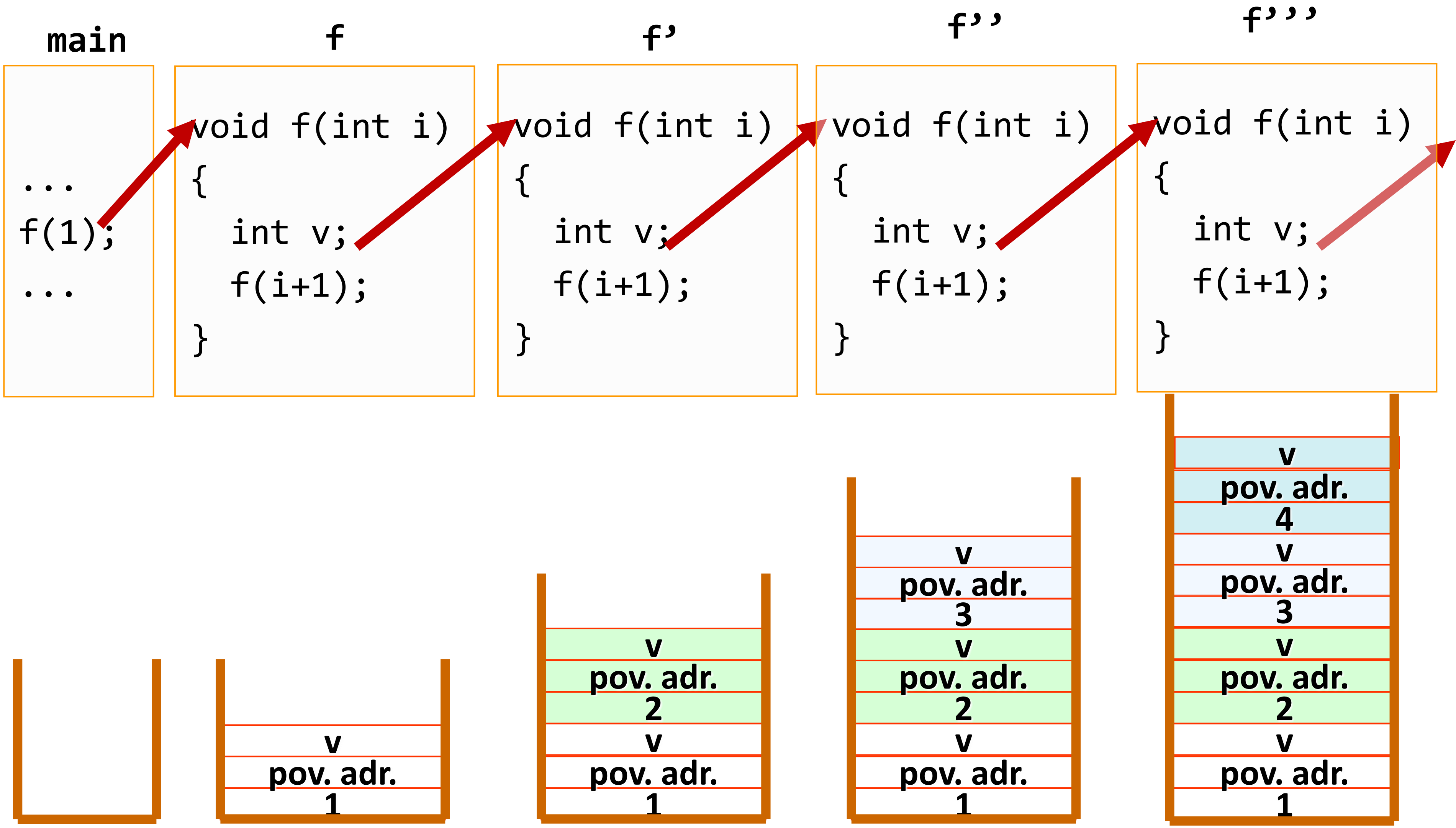
- **svaki korak rekurzije sastoji se od tri dijela** (Cormen *et al*. 2009):
  - podijeli početni problem u manje potprobleme
    - za potproblem:
      ulazni skup je podskup originalnog ulaznog skupa podataka, gdje podskup može biti samo za jedan član manji od početnog skupa ili može biti npr. trećina, polovica, itd. osnovnog skupa podataka
  - riješi potproblem ili rekurzivno ili izravno (ako je potproblem *dovoljno malen*, tj. ako se došlo do osnovnog slučaja)
  - riješi početni problem kombiniranjem rješenja potproblema

Rješavanje rekurzije (3)

- osnovni slučajevi
  - uvijek moraju postojati osnovni slučajevi koji se rješavaju bez rekurzije
- napredovanje
  - za slučajeve koji se rješavaju rekurzivno, svaki sljedeći rekurzivni poziv mora se približiti osnovnim slučajevima
- pravilo projektiranja
  - podrazumijeva se da svaki rekurzivni poziv funkcionira
- pravilo neponavljanja
  - ne dopustiti da se isti problem rješava odvojenim rekurzivnim pozivima, jer to rezultira umnažanjem posla (npr. Fibonaccijevi brojevi, str. 18)

Primjer: elementarna rekurzija i sistemski stog

SimpleRecursion.cpp



Primjer: izračunavanje faktorijela (1)

- jedan od jednostavnih rekurzivnih algoritama jest izračunavanje **n!** za **n >= 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)
  = 4 * 3 * fakt (2)
  = 4 * 3 * 2 * fakt (1)
  = 4 * 3 * 2 * 1
```

$$T(n) = \begin{cases} \Theta(1) & ako\ je\ n = 0 \\ c + T(n-1) & ako\ je\ n > 0 \end{cases}$$

$$\begin{aligned} T(n) &= c + T(n - 1) = c + c + T(n - 2) \\ &= c + c + \ldots + T(1) \\ &= nc + \Theta(1) = \Theta(n) \end{aligned}$$

Primjer: izračunavanje faktorijela (2)

Factorial.cpp

main

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

fakt

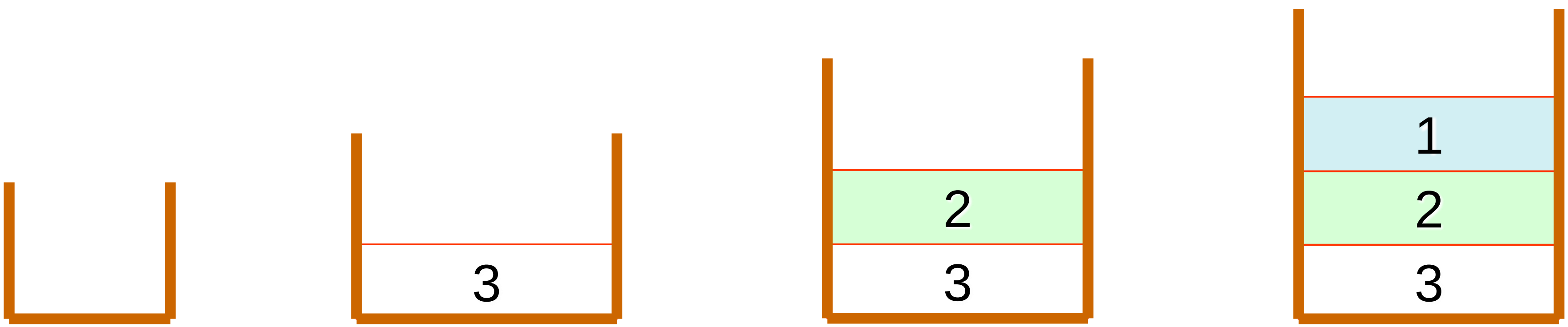
```
int fakt(int n){
  if (n <= 1) {
    return 1;
  } else {
    return
      n * fakt(n-1);
  }
}
```

fakt’

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

fakt’’

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



Zadatak za vježbu: potenciranje rekurzijom (1)

Exponentiation.cpp

- Napisati funkciju koja prima dva cjelobrojna argumenta **x** i **y** i vraća preko povratne vrijednosti **x^y^**.
- Poziv funkcije:

```
k = powr(2, 4);
  = 2 * powr(2, 3)
  = 2 * 2 * powr(2, 2)
  = 2 * 2 * 2 * powr(2, 1)
  = 2 * 2 * 2 * 2 * powr(2, 0)
  = 2 * 2 * 2 * 2 * 1
```

$$T(y) = \begin{cases} \Theta(1) & ako\ je\ y = 0 \\ c + T(y-1) & ako\ je\ y > 0 \end{cases}$$

Složenost: $\Theta(y)$

Zadatak za vježbu: potenciranje rekurzijom (2)

Sadržaj stoga za poziv funkcije: pot(2,5)

powr (2,5)	powr (2,4)	powr (2,3)	powr (2,2)	powr (2,1)	powr (2,0)	return 1	return 2*1	return 2*2	return 2*4	return 2*8	return 2*16
					(2,0)	1					
				(2,1)	(2,1)	(2,1)	2				
			(2,2)	(2,2)	(2,2)	(2,2)	(2,2)	4			
		(2,3)	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)	8		
	(2,4)	(2,4)	(2,4)	(2,4)	(2,4)	(2,4)	(2,4)	(2,4)	(2,4)	16	
(2,5)	(2,5)	(2,5)	(2,5)	(2,5)	(2,5)	(2,5)	(2,5)	(2,5)	(2,5)	(2,5)	32

Zadatak za vježbu: potenciranje rekurzijom (3)

- Što bi se dogodilo kada bi bila izostavljena linija:

```
if (y <= 0) return 1;
```

- funkcija bi samu sebe pozivala beskonačno puta i nikada ne bi vratila neku vrijednost u glavni program, npr.

```
powr(2,4);
      = 2 * powr(2,3)
            = 2 * powr(2,2)
                  = 2 * powr(2,1)
                        = 2 * powr(2,0)
                              = 2 * powr(2,-1)
                                    = 2 * powr(2,-2)
                                                ...
```

Zadatci za vježbu

1. Napisati <u>nerekurzivnu</u> funkciju koja prima dva cjelobrojna argumenta **x** i **y** i vraća vrijednost x^y.

```
int pownr(long x, long y) {
   int retval = 1;
   for (int i = 0; i < y; i++)
      retval *= x;
   return retval;
}
```

2. Napisati funkcije koja ispisuju sve brojeve do **n** (rastući niz) ili od **n** (padajući niz) na razne načine. **PrintRecursive.cpp**

3. Napisati rekurzivnu funkciju koja računa **n-ti** član aritmetičkog niza:

$$a_n = a_1 + (n - 1)d$$

ArithmeticSequence.cpp

Tipovi rekurzija

Prema trenutku u kojemu se funkcija rekurzivno poziva:

- na početku funkcije (engl. *head recursion*): prva naredba je rek. poziv
- na sredini funkcije (engl. *middle recursion*): rek. poziv je u sredini funkcije
- na kraju (repu) funkcije (engl. *tail recursion*): rekurzivni poziv je posljednja akcija u funkciji, tj. povratna vrijednost se nakon rek. poziva odmah vraća iz funkcije

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

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

Prednost rekurzivnog poziva na kraju funkcije

Tko želi znati više ☺

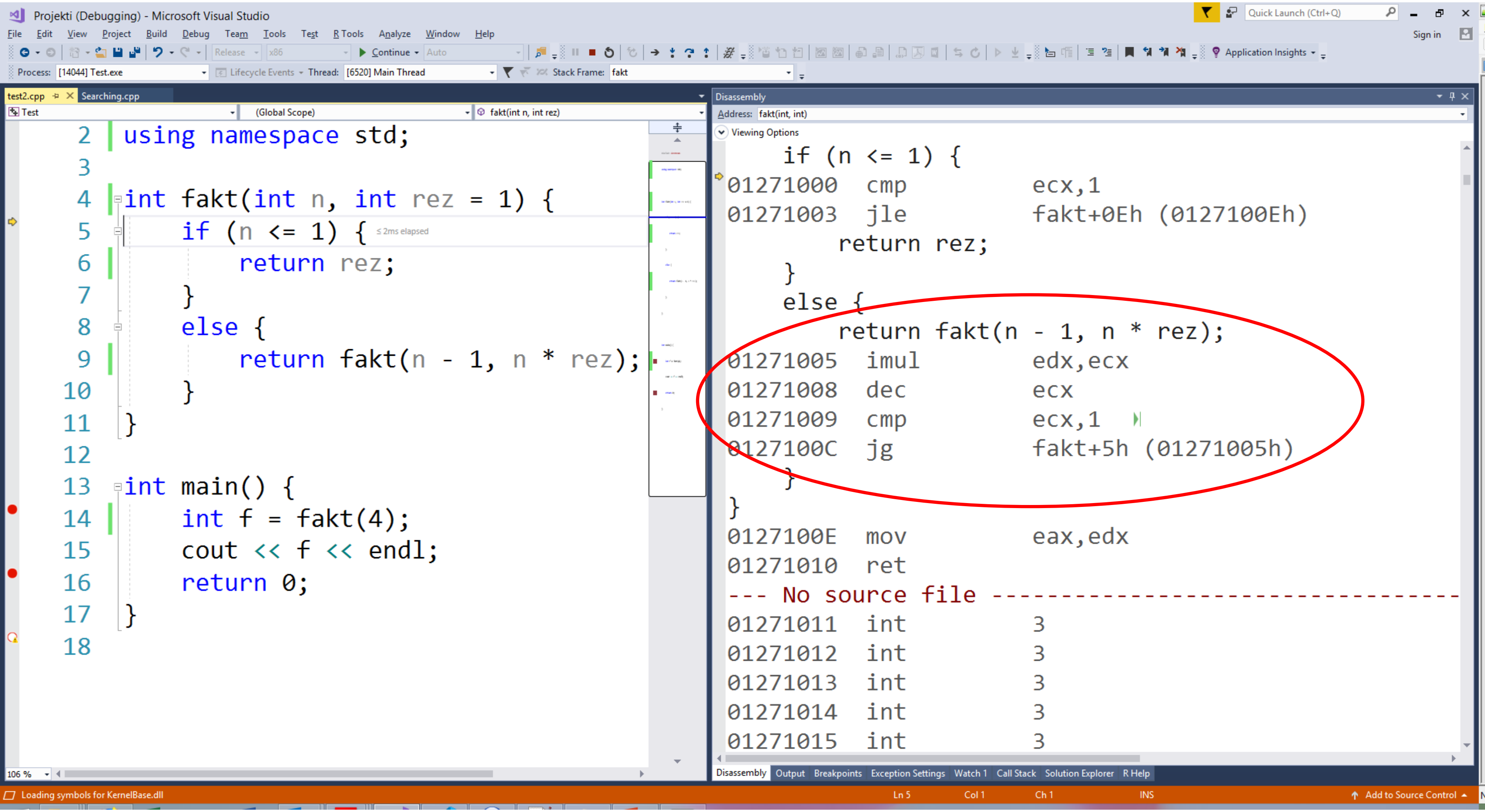
- poziv rekurzije na kraju funkcije zahtijeva da se sva stanja algoritma prate preko argumenata funkcije, tj. ne koriste se lokalne varijable
- ovakav poziv rekurzije na kraju funkcije (engl. *tail recursion*) pretpostavlja micanje okvira stoga prije sljedećeg rek. poziva
- zbog toga, takva je rekurzija **slična petlji ili goto naredbi** (goto prva_naredba_u_funkciji)
- većina prevoditelja (engl. *compiler*) prepoznaje tu situaciju i obavlja optimizaciju programskog koda, tj. uklanja rekurziju
- ako prevoditelj ne prepozna situaciju, onda se obavlja rekurzivni poziv

Rekurzivni poziv u sredini funkcije

```cpp
#include <iostream>

using namespace std;

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

int main() {
    int f = fakt(4);
    cout << f << endl;
    return 0;
}
```

```
int fakt(int n) {
00E31000  push        esi
00E31001  mov         esi,ecx
    if (n <= 1) {
00E31003  cmp         esi,1
00E31006  jg          fakt+0Fh (0E3100Fh)
        return 1;
00E31008  mov         eax,1
00E3100D  pop         esi
    }
}
00E3100E  ret
    }
    else {
        return n * fakt(n - 1);
00E3100F  lea         ecx,[esi-1]
00E31012  call        fakt (0E31000h)
00E31017  imul        eax,esi
00E3101A  pop         esi
    }
```

Rekurzivni poziv na kraju funkcije



Leonardo Pisano Fibonacci

- Fibonacci (Pisa, ~1170. g. – Pisa, ~1250. g.)
- godine 1202. *Liber abaci* :
  - uvođenje hindu-arapskih brojeva
    - *modus Indorum* (indijska metoda)
  - simultane linearne jednadžbe
  - trgovački matematički problemi
  - izračun profita
  - preračunavanje valuta

Fibonaccijevi brojevi

- 1, 1, 2, 3, 5, 8, 13, 21, 34, ... (koji je sljedeći?)

  $F_0 = F_1 = 1$

  $F_i = F_{i-2} + F_{i-1}; \; i > 1$
- program je vrlo kratak i potpuno odgovara matematičkoj definiciji
- učinkovitost je vrlo niska

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

Fibonaccijevi brojevi – rekurzivno rješenje

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

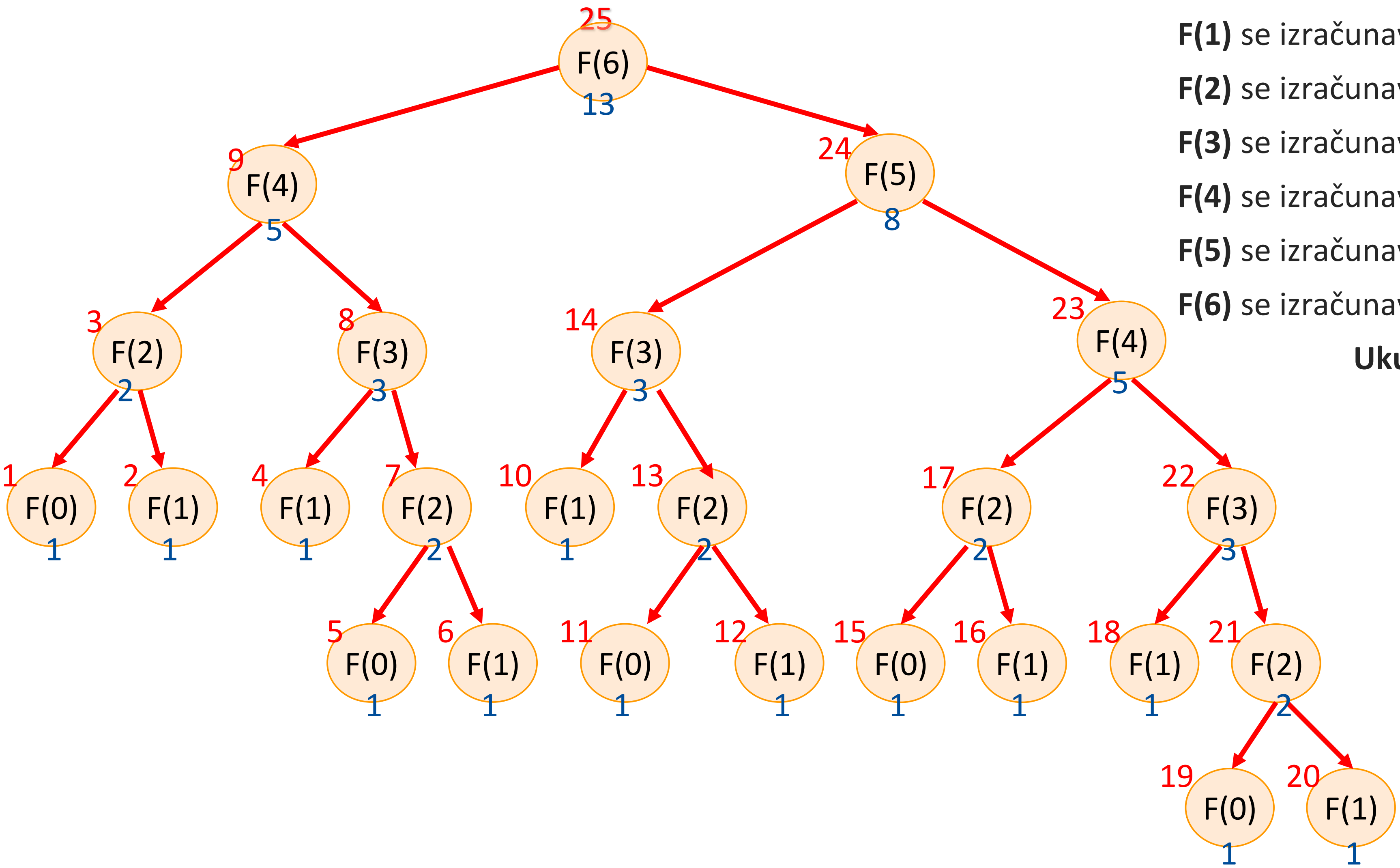
Fibonacci.cpp

```
struct RetValFibonacci {
	int Fn;
	int numCalls;
	RetValFibonacci() : Fn(1), numCalls(1) {}
	RetValFibonacci(int newFn) : ...
	...
	friend RetValFibonacci operator+(
		RetValFibonacci lhs,
		const RetValFibonacci& rhs) { ... }
```

Vrijeme izvođenja: *O(φ^n)*

Memorijska potrošnja: ***O(1)***

Fibonaccijevi brojevi – izvođenje programa



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 : 25

Fibonaccijevi brojevi - vrijeme izvođenja rekurzivne funkcije (1)

$$\boxed{\boldsymbol{T(n)} = \begin{cases} \Theta(\mathbf{1}) & ako\ je\ \boldsymbol{n} \leq \mathbf{1} \\ \boldsymbol{T(n-1)} + \boldsymbol{T(n-2)} + \Theta(\mathbf{1}) & ako\ je\ \boldsymbol{n} > \mathbf{0} \end{cases}}$$

$T(n) = T(n\text{-}1) + T(n\text{-}2) + \Theta(1)$

$\leq 2T(n\text{-}1) + \Theta(1)$

$\leq 2 \cdot [2T(n\text{-}2) + \Theta(1)] + \Theta(1) = 2^2T(n\text{-}2) + 2\Theta(1) + \Theta(1)$

$\leq 2^2 \cdot [2T(n\text{-}3) + \Theta(1)] + 2\Theta(1) + \Theta(1) = 2^3T(n\text{-}3) + 2^2\Theta(1) + 2\Theta(1) + \Theta(1)$

…

$\leq 2^{n\text{-}1}T(1) + (2^{n\text{-}2} + \ldots + 2 + 1)\Theta(1) = (2^{n\text{-}1} + \ldots + 2 + 1)\Theta(1)$

$= O(2^n)$

Fibonaccijevi brojevi - vrijeme izvođenja rekurzivne funkcije (2)

$$\boldsymbol{T(n)} = \begin{cases} \Theta(\mathbf{1}) & ako\ je\ \boldsymbol{n} \leq \mathbf{1} \\ \boldsymbol{T(n-1)} + \boldsymbol{T(n-2)} + \Theta(\mathbf{1}) & ako\ je\ \boldsymbol{n} > \mathbf{0} \end{cases}$$

- prema slici sa str. 18 **pretpostavimo** da je broj koraka $T(n)$ = $\boldsymbol{O(2^n)}$ (eksponencijalna složenost)
- dokaz matematičkom indukcijom:
  - za $n = 1$: $T(1) = \Theta(1)$
  - pretpostavimo da vrijedi $T(n\text{-}1) = O(2^{n-1})$

    Tada slijedi: $T(n) = T(n\text{-}1) + T(n\text{-}2) + \Theta(1)$

    $= O(2^{n-1}) + O(2^{n-2}) + \Theta(1)$

    $= O(2^{n-1} + 2^{n-2})$

    $= O(2^n/2 + 2^n/4) = O(2 \cdot 2^n/4 + 2^n/4) = O(3/4 \cdot 2^n)$

    $= O(2^n)$

Koristili smo: $O(f(n)) + O(g(n)) = O(f(n) + g(n))$

Fibonaccijevi brojevi - vrijeme izvođenja rekurzivne funkcije (3)

- za $n > 1$: $T(n) = T(n\text{-}1) + T(n\text{-}2) + \Theta(1)$

  $= T(n\text{-}1) + T(n\text{-}2)$ **(*)**

<u>Određivanja vremena izvođenja:</u>

- pretpostavimo da vrijeme izvođenja *T(n)* raste eksponencijalno s bazom *a*
- promatramo jednadžbu **(*)** kao: $a^n = a^{n-1} + a^{n-2}$
- nakon dijeljenja jednadžbe s a^{n-2} dobijemo: $a^2 = a + 1$,

  čija su rješenja: $a_{1,2} = \frac{1 \pm \sqrt{5}}{2}$, odnosno $a_1 = (1+\sqrt{5}) / 2 = \varphi \approx 1.618$
- matematičkom indukcijom možemo potvrditi $T(n) = \Theta(a^n)$ za $a = \varphi \approx 1.618$, tj. $\boldsymbol{T(n) = \Theta(\varphi^n)}$

Fibonaccijevi brojevi - vrijeme izvođenja rekurzivne funkcije (4)

- za $n > 1$: $T(n) = T(n\text{-}1) + T(n\text{-}2) + \Theta(1)$
  $= T(n\text{-}1) + T(n\text{-}2)$ **(*)**

Određivanja vremena izvođenja:

- sam izraz **(*)** za računanje $T(n)$ **predstavlja Fibonaccijev broj**
- vrijeme računanja n-tog člana Fibonaccijevog niza jednako je zbroju vremena potrebnih za računanje prethodna dva člana niz
- Fibonaccijev broj F_i može se napisati kao:

  $F_i = \left\lfloor \frac{\varphi^i}{\sqrt{5}} + \frac{1}{2} \right\rfloor$ gdje je $\varphi \approx 1.618$ (zlatni rez) (uz $F_1 = F_2 = 1$)
- zato vrijedi: $\boldsymbol{T(n) = F_n = \Theta(\varphi^n)}$

Fibonaccijevi brojevi - vrijeme izvođenja rekurzivne funkcije (5)

$$\boldsymbol{T(n)} = \begin{cases} \Theta(\mathbf{1}) & ako\ je\ \boldsymbol{n} \leq \mathbf{1} \\ \boldsymbol{T(n-1)} + \boldsymbol{T(n-2)} + \Theta(\mathbf{1}) & ako\ je\ \boldsymbol{n} > \mathbf{0} \end{cases}$$

- $T(n) = \Theta(\varphi^n)$
- slika na str. 18:
  - broj listova stabla je F_n
  - broj unutarnjih čvorova je $F_n - 1$
  - ukupan broj čvorova je $2F_n - 1$, uz vrijeme izvođenja akcija u pojedinom čvoru $\Theta(1)$

Rekurzija vs dinamičko programiranje

Fibonaccijevi brojevi – dinamičko programiranje

- Problem: ako želimo rekurzivno izračunati F_6, onda npr. F_2 trebamo izračunati 5 puta (str. 18)

- Kako ovakav rekurzivan problem riješiti učinkovitije?
- ako ne želimo svaki put iznova računati F_2, možemo ga pohraniti u polje
  - pri prvome pozivu F_2 ćemo izračunati
  - u svim ostalim pozivima, koristit ćemo vrijednost pohranjenu u polju
- umjesto ponovnog računanja istog potproblema rekurzijom, rezultat (rješenje) potproblema se pohranjuje u memoriji, npr. polju (eng. *memoization*)

  → **dinamičko programiranje**

Dinamičko programiranje

- Kada koristiti?
  - kada se potproblemi preklapaju (tj. kada se isti potproblem treba riješiti više puta)
  - kada se problem može riješiti korištenjem rješenja potproblema
- Dva pristupa:
  - *top-down*: koristi se rekurzija, ali tako da se sprema rezultat svakog potproblema (npr. u polje ili tablicu raspršenog adresiranja)
  - *bottom-up*: prvo se rješavaju najmanji problemi, a zatim se na temelju njih rješavaju veći

Dinamičko programiranje – *top-down*

Fibonacci.cpp

```
int *arrF;
RetValFibonacci FibonacciTopDown(int n) {
   if (arrF[n] > 0) return RetValFibonacci{arrF[n]};
   RetValFibonacci r{1};
   if (n > 1)
      r = FibonacciTopDown(n - 1) + FibonacciTopDown(n - 2);
   arrF[n] = r.Fn;
   return r;
}
RetValFibonacci F(int n) override { // public
    arrF = new int[n + 1];
    for (auto i = 0; i <= n; i++) arrF[i] = 0;
    RetValFibonacci r = FibonacciTopDown(n);
    delete[] arrF;
    return RetValFibonacci r;
};
```

Vrijeme izvođenja: *O(n)*

Memorijska potrošnja: ***O(n)***

Dinamičko programiranje – *bottom-up* (1)

Fibonacci.cpp

```
RetValFibonacci F(int n) override {
	int *F = new int[std::max(2, n + 1)];
	F[0] = 1;
	F[1] = 1;
	for (int i = 2; i <= n; i++) {
		F[i] = F[i - 1] + F[i - 2];
	}
	int Fn = F[n];
	delete[] F;
	return RetValFibonacci{ Fn };
};
```

Vrijeme izvođenja: *O(n)*

Memorijska potrošnja: ***O(n)***

Dinamičko programiranje – *bottom-up* (2)

Fibonacci.cpp

```
RetValFibonacci F(int n) override {
	int F0 = 1, F1 = 1, Fn;
	if (n <= 1) return (Fn = 1);
	for (int i = 2; i <= n; i++) {
		Fn = F0 + F1;
		F0 = F1;
		F1 = Fn;
	}
	return RetValFibonacci{ Fn };
};
```

Vrijeme izvođenja: *O(n)*

Memorijska potrošnja: ***O(1)***

Primjeri rekurzija

Najveća zajednička mjera

EuclidGCD.cpp

- jedan od najstarijih algoritama je Euklidov postupak (~300. g. pr. Kr.) za pronalaženje najveće zajedničke mjere (*nzm*) (ili najvećeg zajedničkog djelitelja; *nzd*) dva nenegativna cijela broja:

```
ako je b = 0
      nzm = a
inače
      nzm = najveća zajednička mjera od b i
            ostatka dijeljenja a sa b
```

Najveća zajednička mjera – primjer i funkcija

- primjer:

```
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
nzm (21,0) = 21
nzm (0,21) = nzm (21,0) = 21
```

- rekurzivna funkcija:

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

Najveća zajednička mjera – složenost (1)

- rekurzivna funkcija:

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

- $T(a, 0) = \Theta(1)$
- $T(a, b) = 1 + T(b, r_0) = 2 + T(r_0, r_1) = \ldots = n + T(r_{n-2}, r_{n-1}) = n + 1$
- koliki je n?
- promatramo najlošiji slučaj: kada je ostatak u svakom pozivu rekurzije maksimalan

Najveća zajednička mjera – složenost (2)

- $T(a, 0) = \Theta(1)$
- $T(a, b) = 1 + T(b, r_0) = 2 + T(r_0, r_1) = \ldots$
- najlošiji slučaj: kada je ostatak u svakom pozivu rekurzije maksimalan
- neka a_i i b_i označavaju a i b u i-tom koraku i neka je $a > b$
- vrijedi: $a_{i+2} \leq a_i / 2$ (a je nakon svaka dva rekurzivna poziva manji barem dva puta)
  - ako je $b_i \leq a_i / 2$, onda tvrdnja vrijedi
  - inače, $b_{i+1} = a_i \% b_i \leq a_i / 2$;
    kako je $a_{i+2} = b_{i+1}$, onda slijedi $a_{i+2} \leq a_i / 2$
- n je najviše $2\log_2 a$, pa je vrijeme izvođenja: $O(\log a)$

Traženje člana polja

Searching.cpp

- Rekurzivni postupak za traženje indeksa zadnjeg člana jednodimenzionalnog polja od **n** članova koji ima vrijednost **item**.

```
RetValSearch search(const T A[], const size_t n,
                              const T& item) override {
      int index = n - 1;
      if (index < 0)
            return RetValSearch{ false, -1 };
      else if (A[index] == item)
            return RetValSearch{ true, index };
      return search(A, n - 1, item);
}
```

$$\boldsymbol{najlošiji\ slučaj}:\ \boldsymbol{T}(\boldsymbol{n}) = \begin{cases} \Theta(\mathbf{1}) & ako\ je\ \boldsymbol{n} \leq \mathbf{1} \\ \boldsymbol{T}(\boldsymbol{n}-\mathbf{1}) + \Theta(\mathbf{1}) & ako\ je\ \boldsymbol{n} > \mathbf{1} \end{cases} \Rightarrow \boldsymbol{T}(\boldsymbol{n}) = \Theta(\boldsymbol{n})$$

$$\boldsymbol{najbolji\ slučaj}:\ \Theta(\mathbf{1})\ \boldsymbol{uvijek}$$

Pretraživanje s ograničavačem

Searching.cpp

- pretraživanje je brže ako se prethodno u polje prošireno za jedan član stavi tzv. ograničivač (*sentinel*) `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) ...
```

$$\boldsymbol{najlošiji\ slučaj}: T(n) = \begin{cases} \Theta(1) & ako\ je\ n \le 1 \\ T(n-1) + \Theta(1) & ako\ je\ n > 1 \end{cases} \Rightarrow T(n) = \Theta(n)$$

$$\boldsymbol{najbolji\ slučaj}: \Theta(1)\ \boldsymbol{uvijek}$$

Traženje najvećeg člana polja

MaxElem.cpp

- 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;
}
```

$$\boldsymbol{T(n)} = \begin{cases} \Theta(\mathbf{1}) & ako\ je\ \boldsymbol{n} \leq \mathbf{1} \\ \boldsymbol{T(n-1)} + \Theta(\mathbf{1}) & ako\ je\ \boldsymbol{n} > \mathbf{1} \end{cases} \Rightarrow \boldsymbol{T(n)} = \Theta(\boldsymbol{n})$$

Primjer pogreške

IncorrectRecursion.cpp

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

- za vrijednost **n = 1** rekurzivni poziv je opet s argumentom **1**
  - nema napredovanja prema osnovnom slučaju
- program ne radi niti za druge vrijednosti argumenta:
  - npr. za **n = 4**, rekurzivno se poziva `failed` s argumentom **4/3 + 1 = 2**, zatim **2/3 + 1 = 1** i dalje stalno **1/3 + 1 = 1**

Rekurzija – složeniji primjeri

Kamate

Interest.cpp

- Zadana suma novaca oročena je u banci na zadani broj godina *n* uz zadanu godišnju kamatnu stopu *p*. Napisati program koji računa dobivenu sumu nakon isteka oročenja. Odredite *T(n)*.
- $g_n = g_0 \cdot \left(1 + \frac{p}{100}\right)^n$
- g_n – glavnica nakon *n* godina, g_0 – početna glavnica

```
float kamrac (float g, int n, float p) {
  // g - glavnica
  // n - trajanje oročenja u godinama
  // p - kamatna stopa u postotcima
  if (n <= 0) return g;
  else return (1 + p / 100) * kamrac(g, n - 1, p);
}
```

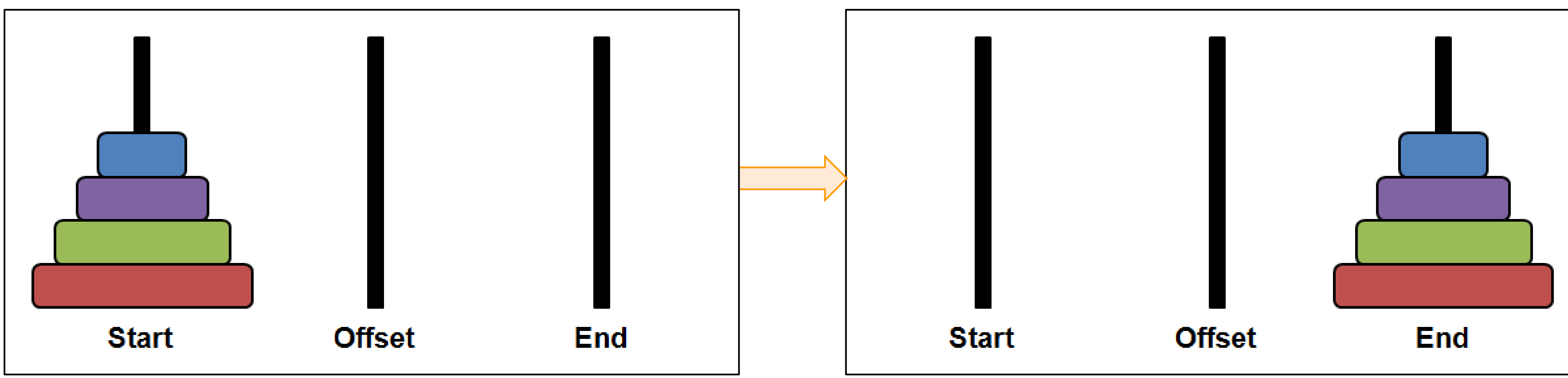
Obrtaljka

Palindrome.cpp

- Napišite program koji će rekurzivno provjeriti je li zadana riječ ili rečenica duljine *n* obrtaljka (palindrom). U ulaznom nizu podataka zanemarite razmak i sve znakove interpunkcije.
  - Primjeri:
    - UDOVICA BACI VODU
    - ON VIDI DIVNO
    - U RIMU UMIRU
    - ANA NABRA PAR BANANA
- Naputak: ako u palindromu izbacite prvo i posljednje slovo, preostali tekst također mora biti obrtaljka
- Odredite $T(n)$.

Hanojski tornjevi (1)

- Zadani su štapovi **I** (izvor), **O** (odredište), **P** (pomoćni).
- Na prvom štapu (**I**) ima **n** diskova različite veličine postavljenih tako da veći nikad ne dolazi iznad manjeg.
- Uz minimalni broj operacija preselite sve diskove na **O**, jedan po jedan. Disk se smije postaviti ili na prazan štap ili tako da je manji disk na većem.
  - Animacija: https://en.wikipedia.org/wiki/Tower_of_Hanoi#/media/File:Tower_of_Hanoi_4.gif



https://www.ocf.berkeley.edu/~shidi/cs61a/wiki/Towers_of_Hanoi

Hanojski tornjevi (2)

Hanoi.cpp

- Algoritam rješenja:
  - ignorirati donji (najveći) disk i riješiti problem za ***n*-1** diskova, ali tako da premještamo diskove sa štapa **I** na štap **P** koristeći **O** kao pomoćni
  - sada se najveći disk nalazi na **I**, a ostalih ***n*-1** na **P**
  - preseliti najveći disk sa **I** na **O**
  - preseliti ***n*-1** diskova s **P** na **O** koristeći **I** kao pomoćni (problem je već riješen za ***n*-1** diskova)

Hanojski tornjevi (3)

- Vrijeme izvođenja:
  - $\boldsymbol{T(1) = \Theta(1)}$
  - $\boldsymbol{T(n)} = T(n - 1) + \Theta(1) + T(n - 1)$

$$= 2 \cdot \boldsymbol{T(n - 1)} + \Theta(1)$$

$$= 2 \cdot \boldsymbol{(2 \cdot T(n - 2) + \Theta(1))} + \Theta(1)$$

$$= 2 \cdot \boldsymbol{(2 \cdot (2T(n - 3) + \Theta(1)) + \Theta(1))} + \Theta(1)$$

$$= 2^3 \cdot T(n - 3) + (2^2 + 2^1 + 2^0) \cdot \Theta(1)$$

…

$$= 2^{n-1} \cdot T(1) + (2^{n-2} + \ldots + 2^0) \cdot \Theta(1)$$

$$= (2^{n-1} + \ldots + 2^0) \cdot \Theta(1)$$

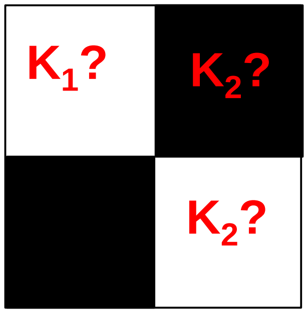
$$= (2^n - 1) / (2 - 1) \cdot \Theta(1)$$

$$= \boldsymbol{\Theta(2^n)}$$

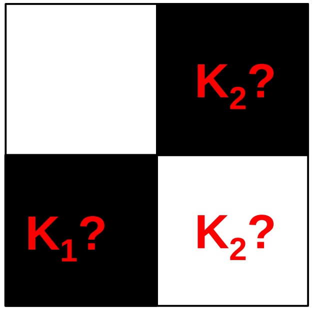
Problem postavljanja *n* kraljica na šahovsku ploču

- problem određivanja pozicija 8 kraljica na šahovskoj ploči, tako da se one međusobno ne napadaju
- općenitiji problem: postaviti ***n*** kraljica na šahovsku ploču ***n* x *n*** tako da se međusobno ne napadaju
  - rješenja postoje za $n \geq 4$ (za $n = 2$ i $n = 3$ ne postoje rješenja)
- zagonetku je postavio šahist Max Bezzel (1848.) i od tada su mnogi matematičari (uključujući Gaussa) radili na rješavanju toga problema
- Dijkstra je 1972. objavio detaljan opis algoritma za rješenje ovoga problema korištenjem postupka praćenja unatrag (engl. *depth first backtracking*)

Postavljanje kraljica za n = 2



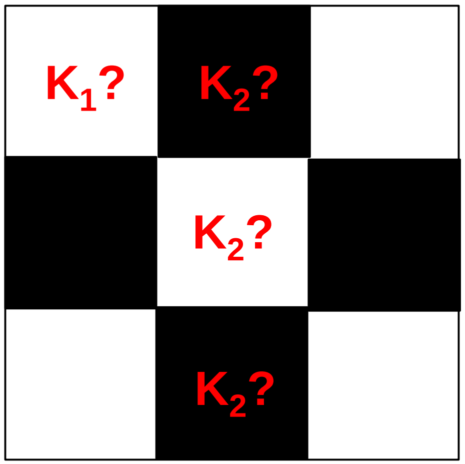
- Kraljicu K_1 postavimo u prvi stupac i prvi redak.
- Možemo li postaviti kraljicu K_2 u drugi stupac?
- Nije moguće, jer K_1 napada K_2 u bilo kojem retku drugog stupca.



- Kraljicu K_1 postavimo u prvi stupac i drugi redak.
- Možemo li postaviti kraljicu K_2 u drugi stupac?
- Nije moguće, jer K_1 napada K_2 u bilo kojem retku drugog stupca.

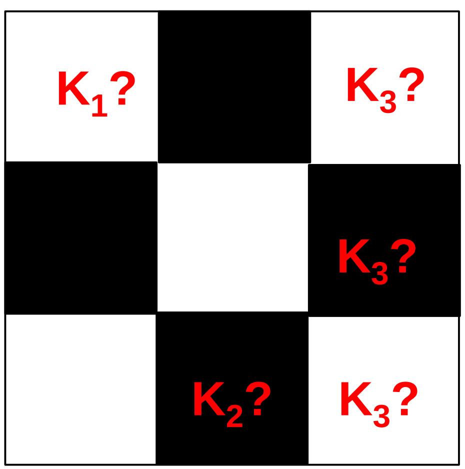
- **Zaključak:** ne postoji rješenje za n = 2.

Postavljanje kraljica za *n* = 3 (1)



- Kraljicu K_1 postavimo u prvi stupac i prvi redak.
- Možemo li postaviti kraljicu K_2 u drugi stupac?

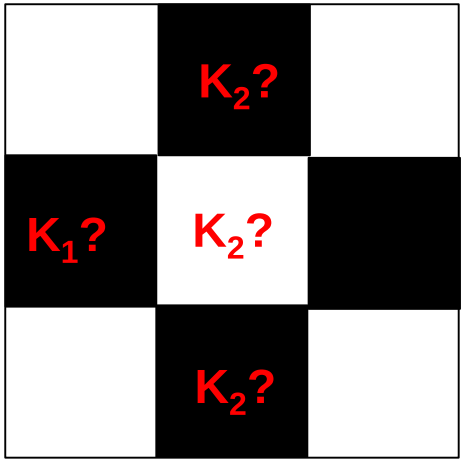
- K_2 možemo postaviti samo u treći redak u drugom stupcu.
- S obzirom da K_2 ne može biti u prvom ili drugom retku, onda uopće ne ispitujemo kombinacije s K_3 koje bi uključivale te pozicije K_2



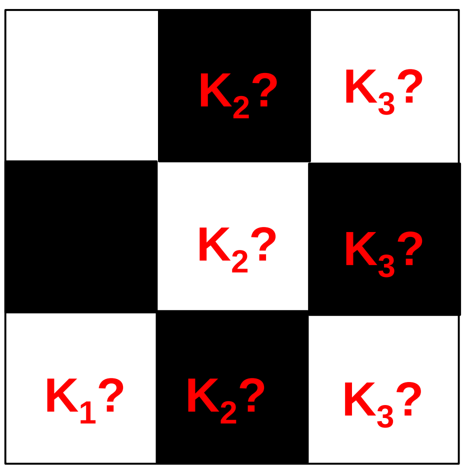
- Kraljicu K_1 postavimo u prvi stupac i prvi redak.
- Kraljicu K_2 postavimo u drugi stupac i treći redak.
- Možemo li postaviti kraljicu K_3 u treći stupac?

- K_3 ne možemo postaviti u treći stupac, pa zaključujemo da ne postoji rješenje gdje je K_1 u prvom retku prvoga stupca.

Postavljanje kraljica za *n* = 3 (2)



- S obzirom da ne postoji rješenje u kojemu bi **K_1** bila u prvom retku, vraćamo se natrag i postavljamo **K_1** u <u>drugi</u> redak, pa zatim ponavljamo postupak traženja mogućih pozicija za **K_2** i **K_3**.



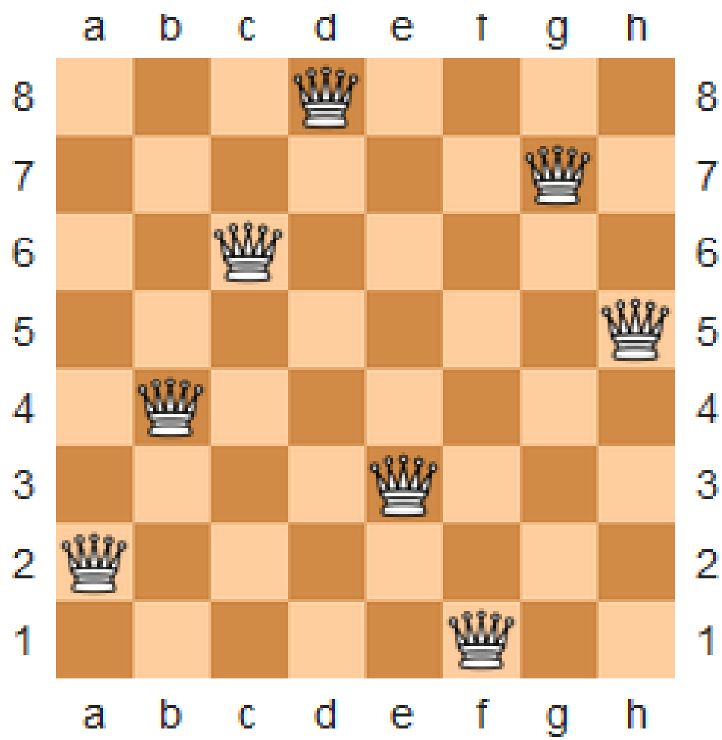
- Za **K_1** u drugome retku nema rješenja.
- Ponovit ćemo isti postupak i za **K_1** u trećem retku prvoga stupca.

Zaključak:

- S obzirom da **K_1** ne može biti niti u jednome retku (a da se pri tome kraljice međusobno ne napadaju), zaključujemo da ne postoji rješenje za $n = 3$.

n kraljica (1)

- Algoritam rješenja: **Queens.cpp**
  - promatramo stupce na šahovskoj ploči od prvoga prema zadnjemu
  - u svaki postavljamo jednu kraljicu
  - promatramo ploču u situaciji kada je već postavljeno ***i*** kraljica (u ***i*** različitih stupaca) koje se međusobno ne napadaju
  - želimo postaviti ***i* + 1** kraljicu tako da ona ne napada niti jednu od već postavljenih kraljica i da se ostale kraljice mogu postaviti uz uvjet nenapadanja



wiki/Eight_queens_puzzle

- napomena: postoje 92 različita rješenja za *n* = 8

n kraljica (2)

- koristimo princip *praćenja unatrag* (engl. *backtracking*)
  - poziciju kraljice zapisujemo kao par (stupac, redak)
  - prva se kraljica postavlja u prvi stupac i prvi redak
  - sljedeća se kraljica pokušava postaviti u prvi sljedeći stupac ***i***, a unutar ***i***-tog stupac u prvi mogući **redak *r*** tako da ne napada niti jednu već postavljenu kraljicu
  - ako takav stupac ne postoji, program se vraća u posljednje stanje koje je bilo dobro (*praćenje unatrag*) i onda pokušava postaviti kraljicu u sljedeći **redak (*r* + 1)** u *i*-tom stupcu
- <u>Napomena:</u> čim jedna kraljica ne može biti postavljena u neki redak, sve kombinacije drugih kraljica koje bi uključivale tu poziciju više se ne provjeravaju, jer je jasno da ne vode rješenju.