

# Rekurze

Jan Faigl

Katedra počítačů

Fakulta elektrotechnická

České vysoké učení technické v Praze

Přednáška 6

**A0B36PR1 – Programování 1**

# Část 1 – Rekurze

Faktoriál

Obrácený výpis

Karel

Hanojské věže

Rekurze

Fibonacciho posloupnost

## Část 2 – Příklady

Eratosthenovo síto

Řazení

# Část I

## Rekurze

# Výpočet faktoriálu

## ■ Iterace

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

```
int factorialI(int n) {  
    int f = 1;  
    for(; n > 1; --n) {  
        f *= n;  
    }  
    return f;  
}
```

## ■ Rekurze

$$n! = 1 \text{ pro } n \leq 1$$
$$n! = n(n-1)! \text{ pro } n > 1$$

```
int factorialR(int n) {  
    int f = 1;  
    if (n > 1) {  
        f = n * factorialR(n-1);  
    }  
    return f;  
}
```

lec05/DemoFactorial.java

*Připomínka*

## Příklad výpis posloupnosti 1/3

- Vytvořte program, který přečte posloupnost čísel a vypíše ji v opačném pořadí
- Rozklad problému:
  - Zavedeme abstraktní příkaz: „*obrat' posloupnost*”
  - Příkaz rozložíme do tří kroků:
    1. Přečti číslo  
**číslo uložíme pro pozdější „obracený” výpis**
    2. Pokud není detekován konec posloupnost „*obrat' posloupnost*”  
**pokračujeme ve čtení čísel**
    3. Vypiš číslo  
**vypíšeme uložené číslo**

## Příklad výpis posloupnosti 2/3

```
1 void reverse() {
2     int v = scan.nextInt();
3     if (scan.hasNext()) {
4         reverse();
5     }
6     System.out.printf("%3d ", v);
7 }
8
9 public void start() {
10    System.err.println("Enter a sequence of numbers (
11    use ctr+D for the end of the the sequence");
12    reverse();
13    System.out.println(""); //end of line
14 }
```

lec06/DemoRevertSequence.java

## Příklad výpis posloupnosti 3/3

- lec06/DemoRevertSequence.java

- Vytvoření posloupností

```
./generate_numbers.sh | tr '\n' ' ' | cat > numbers.txt
javac DemoRevertSequence.java
java DemoRevertSequence < numbers.txt 2>/dev/null > numbers-r.txt
java DemoRevertSequence <numbers-r.txt 2>/dev/null > numbers-rr.txt
```

- Příkaz pro výpis obsahu souborů

```
for i in numbers.txt numbers-r.txt numbers-rr.txt; do
    echo "$i"; cat $i; echo ""; done
```

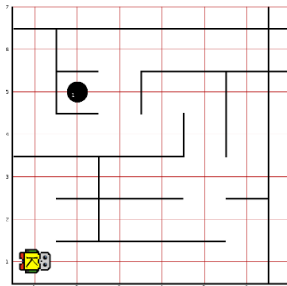
- Výpis obsahu souborů

```
numbers.txt
10 4 20 8 8 5 18 6 7 7
numbers-r.txt
7 7 6 18 5 8 8 20 4 10
numbers-rr.txt
10 4 20 8 8 5 18 6 7 7
```



## Robot Karel – hledání „beeperu“

- Robot Karel má za úkol najít „beeper“ v bludišti a **vrátit se s ním zpět na výchozí pozici**
- Karel se pohybuje v mřížkovém ortogonálním světě
- Karel umí:
  - jít rovně o jeden krok
  - otočit se doleva o  $90^\circ$
  - vzít / položit „beeper“
  - vypnout se
- Karel umí zjistit, zdali:
  - je před ním zeď
  - na políčku, na kterém stojí, je „beeper“



## Robot Karel – Iterační řešení 1/3

### ■ Návrh řešení

1. Jdi podél zdi (po pravé straně) dokud není na políčku „beeper“ a počítej počet kroků;
2. Seber „beeper“;
3. Otoč se a jdi podél zdi (po levé straně) a udělej stejný počet kroků jako při hledání „beeperu“;
4. Polož „beeper“ a vypni se.

### ■ Abstraktní příkazy pro nalezení „beeperu“ sledování zdi na pravé straně:

- Otoč se doprava: **turnRight**
- Zjistí, zda-li je na pravé straně zeď: **isWallOnRight**
- Jdi podél pravé zdi: **goByRightWall**

### ■ Analogické abstraktní příkazy pro pohyb podél zdi na levé straně:

- **isWallOnLeft** a **goByLeftWall**

## Robot Karel – Iterační řešení 2/3

```
1 void turnRight() {
2     for (int i = 0; i < 3; i++) {
3         turnLeft();
4     }
5 }
6
7 boolean isWallOnRight() {
8     turnRight();
9     boolean out = isInFrontOfWall();
10    turnLeft();
11    return out;
12 }
13
14 void goByRightWall() {
15     while (!isOnBeeper()) {
16         if (!wallOnRight()) {
17             turnRight();
18         }
19         while (isInFrontOfWall()) {
20             turnLeft();
21         }
22         step();
23     }
24 }
```

*Analogicky pro zed' po levé straně*

## Robot Karel – Iterační řešení 3/3

- Počet kroků je inkrementován v proceduře `step`
- Při návratu podél levé zdi Karel ujde stejný počet kroků

```
1  public void execute() {
2      goByRight();
3      pickBeeper();
4      turnAround();
5
6      if (stepsTaken > 0) {
7          goByLeft(stepsTaken);
8      }
9
10     putBeeper();
11     turnOff();
12 }
```

## Robot Karel – Rekurzivní řešení 1/3

- Řešení rekurzí založíme na opakovaném volání funkce, která posune robot pouze o jeden krok vpřed a následně jej vrátí zpět ve funkci **go**:
  1. Pokud jsi na „beeperu” seber jej a otoč o  $180^\circ$  a ukonči hledání
  2. Zapamatuj si svou orientaci a udělej jeden krok podél pravé zdi
  3. **go** (opakuj hledání o jeden krok)
  4. Udělej jeden krok zpět (vrať se o jeden krok zpět a natoč se do původního směru)

## Robot Karel – Rekurzivní řešení 2/3

```
1 void go() {
2     if (isOnBeeper()) {
3         pickBeeper();
4         turnAround(); //turn back
5         return;
6     }
7     int numberTurnRight = 0;
8     if (!isWallOnRight()) {
9         turnRight();
10        numberTurnRight++;
11    }
12
13    while (isInFrontOfWall()) {
14        turnLeft();
15        numberTurnRight--;
16    }
17    move(); //move forward
18    go();
19    move(); //move backward
20
21    numberTurnRight = (numberTurnRight + 4) % 4;
22    for (int i = 0; i < numberTurnRight; ++i) {
23        turnLeft(); //revert the turns
24    }
25 }
```

## Robot Karel – Rekurzivní řešení 3/3

- Hlavní procedura obsahuje pouze volání funkce **go**, která vrací robot na výchozí políčko

```
1 void execute() {  
2     go();  
3     putBeeper();  
4     turnOff();  
5 }
```

- Pro spuštění využijeme knihovnu `KarelJRobot.jar` a modifikovaný svět `maze.klwd`

```
javac -cp lib/KarelJRobot.jar DemoKarel.java  
java -cp lib/KarelJRobot.jar:. DemoKarel
```

- Robot se při zpáteční cestě vrací „rychleji“, protože již explicitně nekontroluje, zda-li je zed' na levé straně.

[lec06/DemoKarel.java](#)

## Příklad Hanojské věže



- Přemístit disky na druhou jehlu s použitím třetí (pomocné) jehly za dodržení pravidel:

1. V každém kroku můžeme přemístit pouze jeden disk a to vždy z jehly na jehlu

*Disky nelze odkládat mimo jehly*

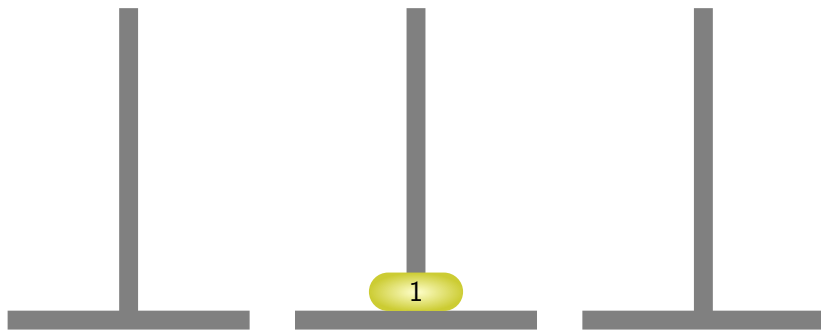
2. Položit větší disk na menší není dovoleno



## Hanojské věže – 1 disk

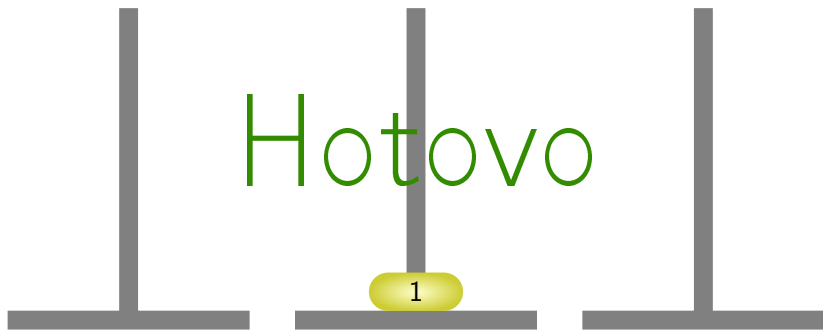


## Hanojské věže – 1 disk



Přesunutý disk z jehly 1 na jehlu 2.

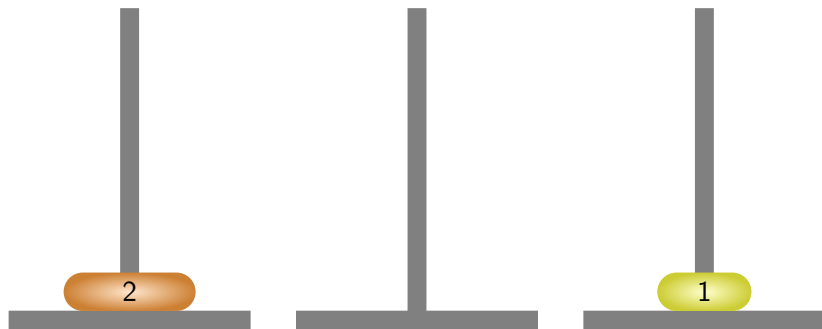
## Hanojské věže – 1 disk



## Hanojské věže – 2 disky

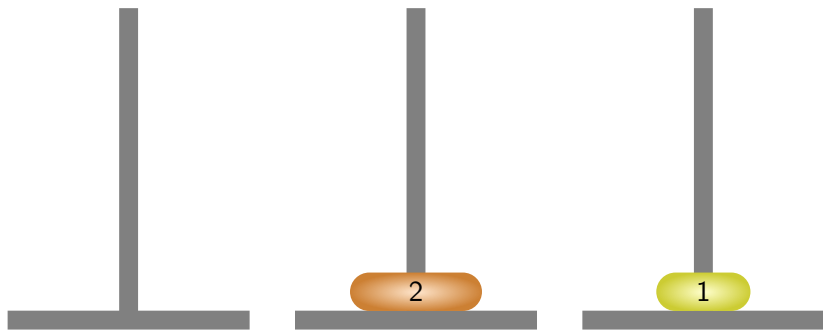


## Hanojské věže – 2 disky



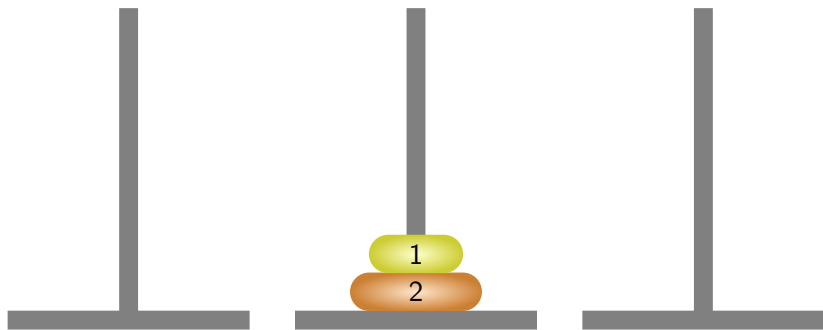
Přesunutý disk z jehly 1 na jehlu 3.

## Hanojské věže – 2 disky



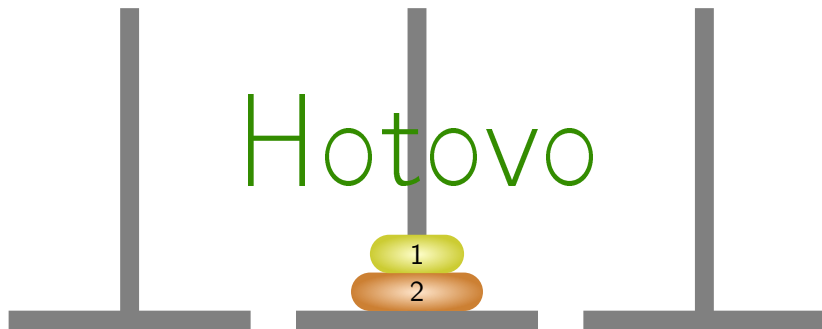
Přesunutý disk z jehly 1 na jehlu 2.

## Hanojské věže – 2 disky



Přesunutý disk z jehly 3 na jehlu 2.

## Hanojské věže – 2 disky

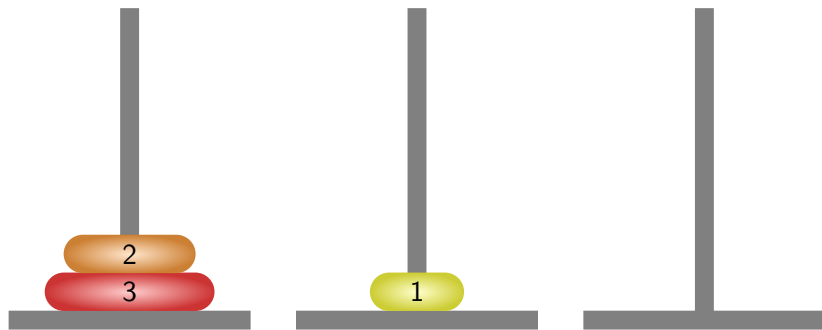




## Hanojské věže – 3 disky

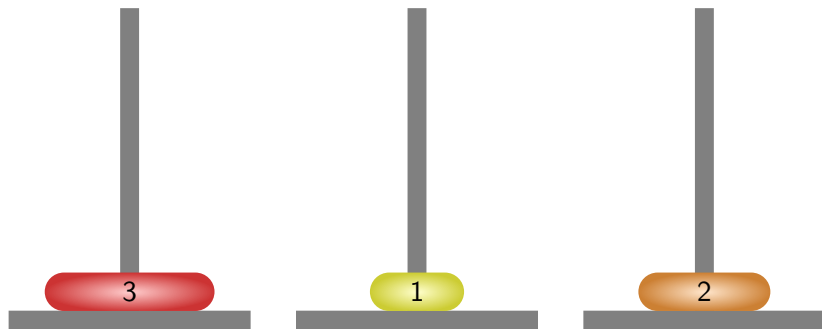


## Hanojské věže – 3 disky



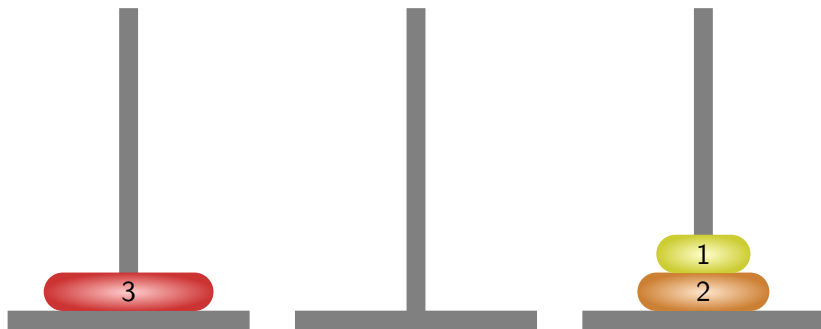
Přesunutý disk z jehly 1 na jehlu 2.

## Hanojské věže – 3 disky



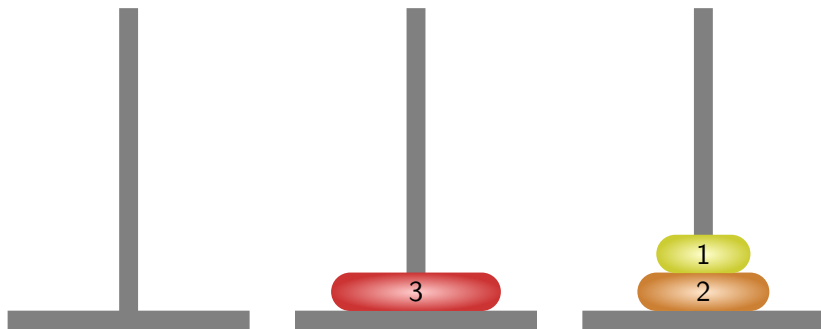
Přesunutý disk z jehly 1 na jehlu 3.

## Hanojské věže – 3 disky



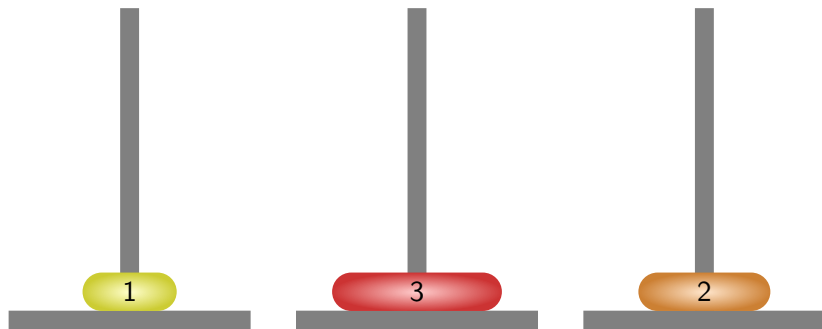
Přesunutý disk z jehly 2 na jehlu 3.

## Hanojské věže – 3 disky



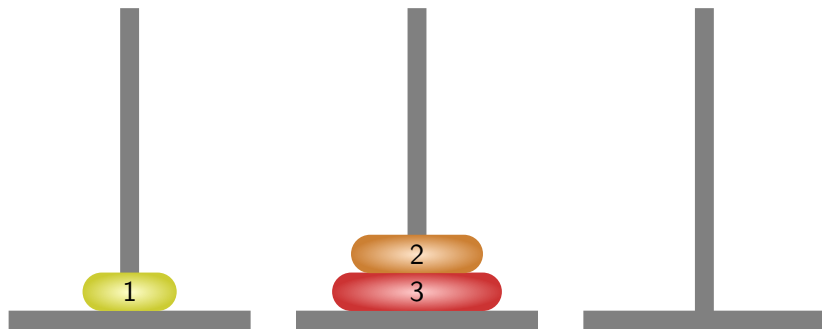
Přesunutý disk z jehly 1 na jehlu 2.

## Hanojské věže – 3 disky



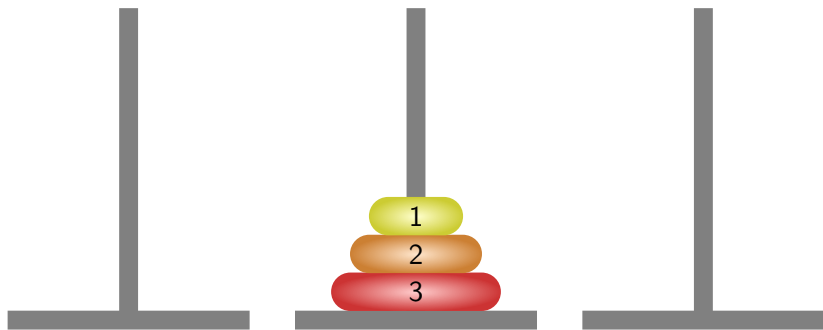
Přesunutý disk z jehly 3 na jehlu 1.

## Hanojské věže – 3 disky



Přesunutý disk z jehly 3 na jehlu 2.

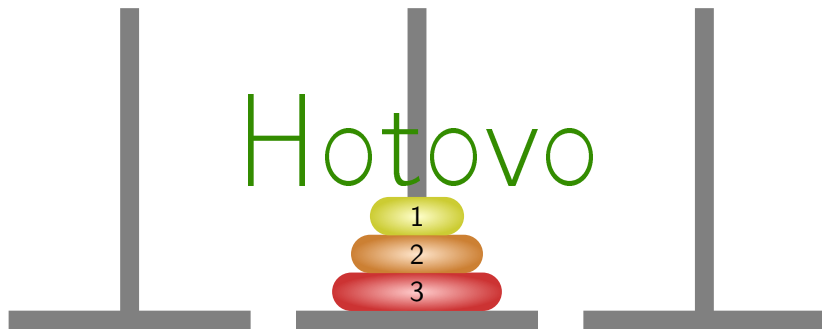
## Hanojské věže – 3 disky



Přesunutý disk z jehly 1 na jehlu 2.



## Hanojské věže – 3 disky



## Hanojské věže – 4 disky

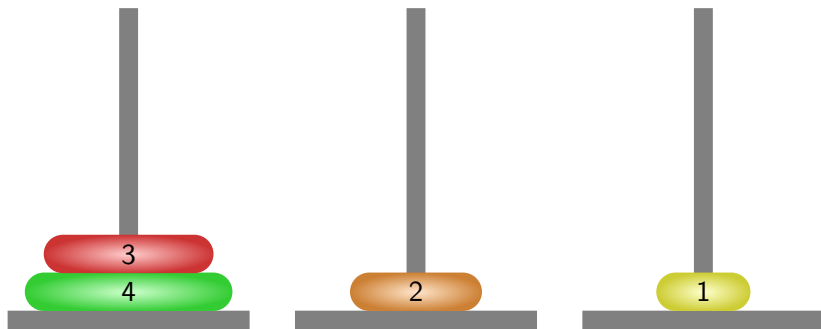


## Hanojské věže – 4 disky



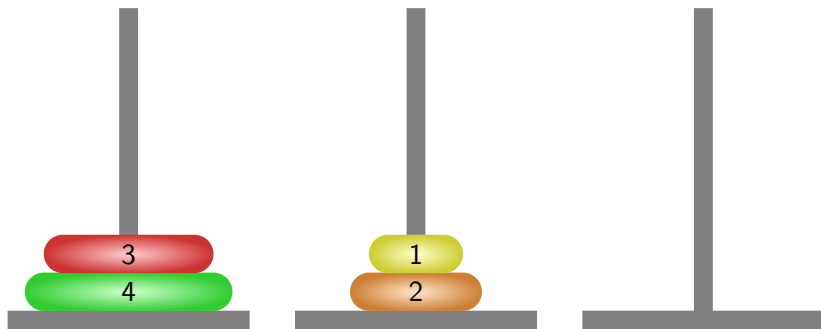
Přesunutý disk z jehly 1 na jehlu 3.

## Hanojské věže – 4 disky



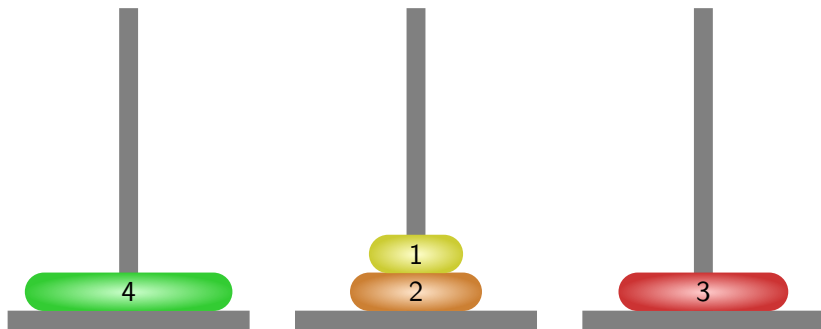
Přesunutý disk z jehly 1 na jehlu 2.

## Hanojské věže – 4 disky



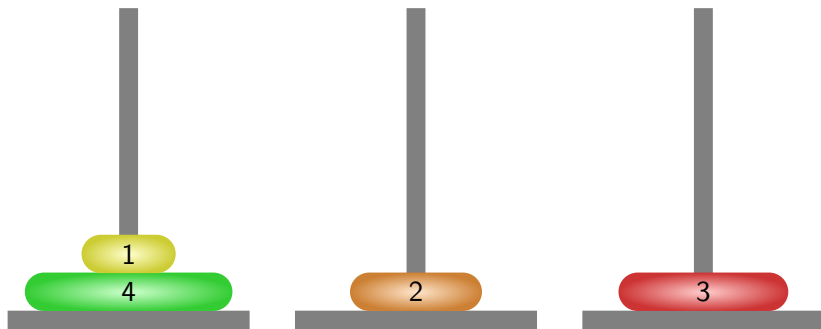
Přesunutý disk z jehly 3 na jehlu 2.

## Hanojské věže – 4 disky



Přesunutý disk z jehly 1 na jehlu 3.

## Hanojské věže – 4 disky



Přesunutý disk z jehly 2 na jehlu 1.

## Hanojské věže – 4 disky



Přesunutý disk z jehly 2 na jehlu 3.

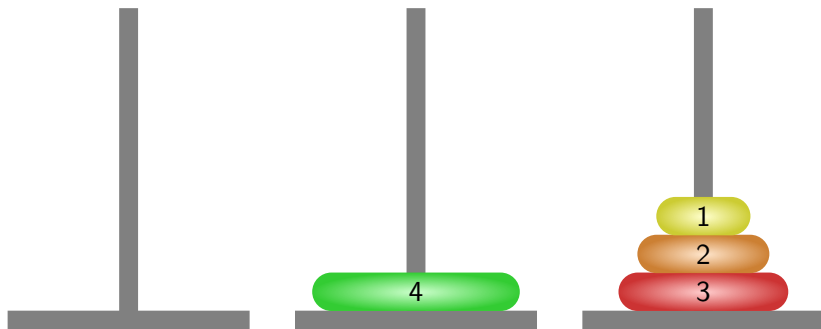


## Hanojské věže – 4 disky



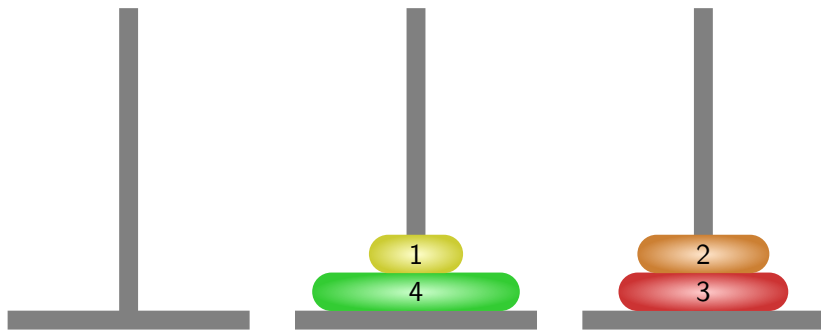
Přesunutý disk z jehly 1 na jehlu 3.

## Hanojské věže – 4 disky



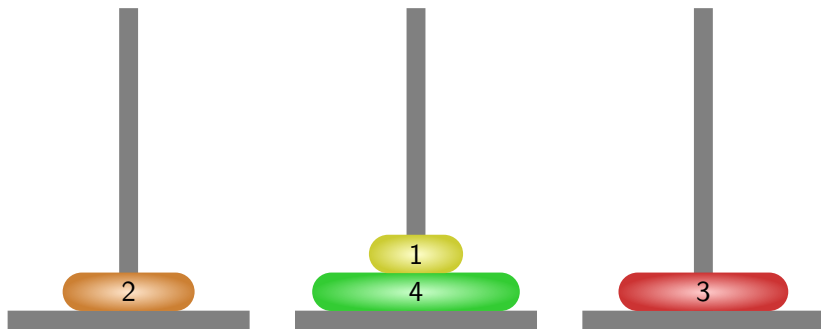
Přesunutý disk z jehly 1 na jehlu 2.

## Hanojské věže – 4 disky



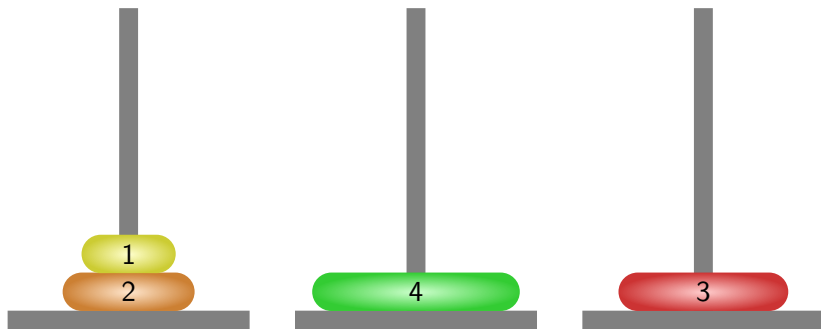
Přesunutý disk z jehly 3 na jehlu 2.

## Hanojské věže – 4 disky



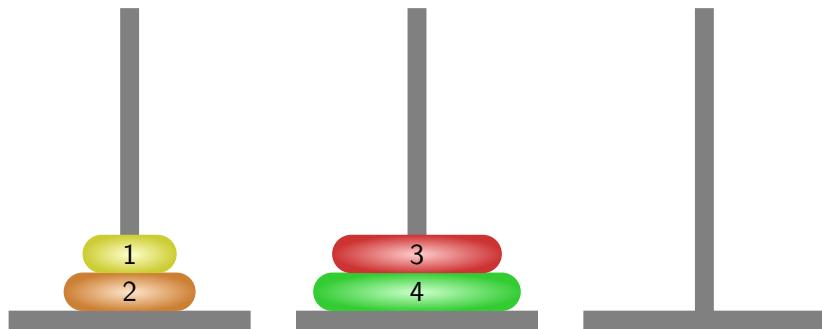
Přesunutý disk z jehly 3 na jehlu 1.

## Hanojské věže – 4 disky



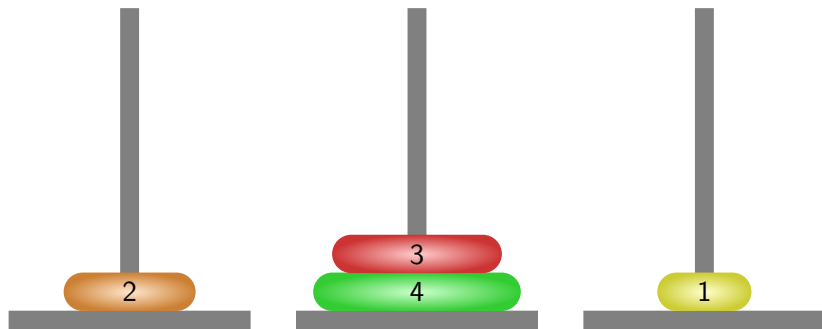
Přesunutý disk z jehly 2 na jehlu 1.

## Hanojské věže – 4 disky



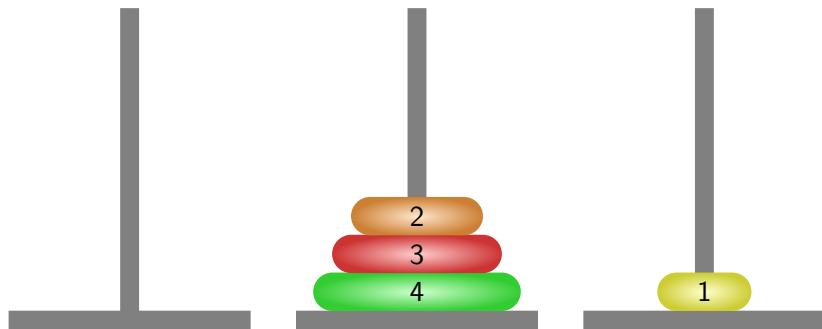
Přesunutý disk z jehly 3 na jehlu 2.

## Hanojské věže – 4 disky



Přesunutý disk z jehly 1 na jehlu 3.

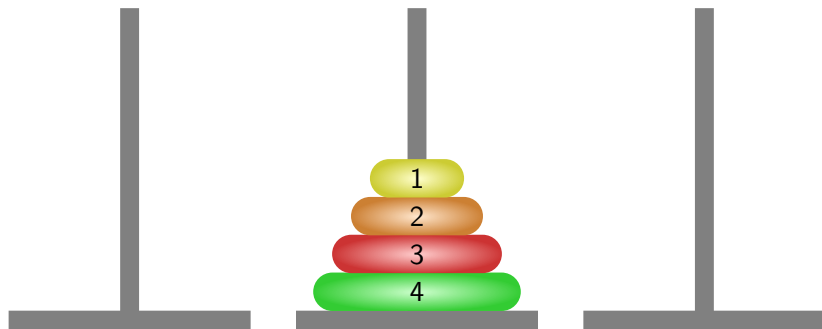
## Hanojské věže – 4 disky



Přesunutý disk z jehly 1 na jehlu 2.

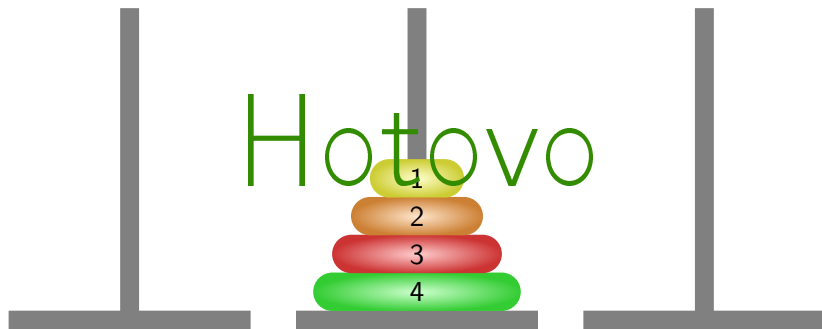


## Hanojské věže – 4 disky

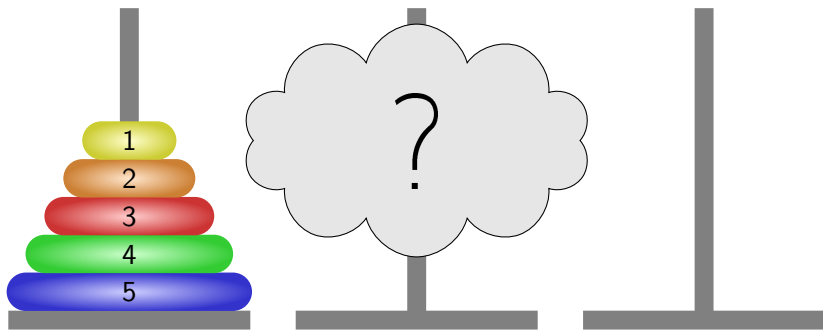


Přesunutý disk z jehly 3 na jehlu 2.

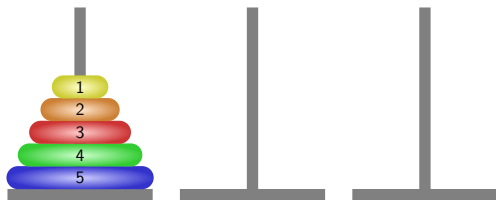
## Hanojské věže – 4 disky



## Hanojské věže – 5 disků



## Návrh řešení



- Zavedeme abstraktní příkaz **moveTower( $n$ , 1, 2, 3)** realizující přesun  $n$  disků z jehly 1 na jehlu 2 s použitím jehly 3.
- Pro  $n > 0$  můžeme příkaz rozložit na tři jednodušší příkazy
  1. **moveTower( $n-1$ , 1, 3, 2)**  
přesun  $n - 1$  disků z jehly 1 na jehlu 3
  2. „*přenes disk z jehly na jehlu 2*“  
přesun největšího disku na cílovou pozici  
*abstraktní příkaz*
  3. **moveTower( $n-1$ , 3, 2, 1)**  
přesun  $n - 1$  disků na cílovou pozici

## Příklad řešení

```
1 void moveTower(int n, int from, int to, int tmp) {
2     if (n > 0) {
3         moveTower(n - 1, from, tmp, to); //move to tmp
4         System.out.println("Move disc from " + from + "
5         to " + to);
6         moveTower(n - 1, tmp, to, from); //move from tmp
7     }
8 }
9 void start() {
10     int numberOfDiscs = 5;
11     moveTower(numberOfDiscs, 1, 2, 3);
12 }
```

lec06/DemoTowersOfHanoi.java

## Příklad výpisu

### ■ lec06/DemoTowersOfHanoi.java

```
java DemoTowersOfHanoi 3
Move disc from 1 to 2
Move disc from 1 to 3
Move disc from 2 to 3
Move disc from 1 to 2
Move disc from 3 to 1
Move disc from 3 to 2
Move disc from 1 to 2
```

```
java DemoTowersOfHanoi 4
Move disc from 1 to 3
Move disc from 1 to 2
Move disc from 3 to 2
Move disc from 1 to 3
Move disc from 2 to 1
Move disc from 2 to 3
Move disc from 1 to 3
Move disc from 1 to 2
Move disc from 3 to 2
Move disc from 3 to 1
Move disc from 2 to 1
Move disc from 3 to 2
Move disc from 1 to 3
Move disc from 1 to 2
Move disc from 3 to 2
```

# Rekurzivní algoritmy

- Rekurzivní funkce jsou přímou realizací rekurzivních algoritmů
- Rekurzivní algoritmus předepisuje výpočet „**shora dolů**”  
v závislosti na velikosti vstupních dat
  - Pro nejmenší (nejjednodušší) vstup je výpočet předepsán přímo
  - Pro obecný vstup je výpočet předepsán s využitím téhož algoritmu pro menší vstup
- Výhodou rekurzivních funkcí je jednoduchost a přehlednost

## Rekurzivní vs iteračními algoritmy

- Nevýhodou rekurzivních algoritmů může být časová náročnost způsobená např. zbytečným opakováním výpočtu
- Řadu rekurzivních algoritmů lze nahradit iteračními, které počítají výsledek „**zdola nahoru**“, tj. od menších (jednodušších) vstupních dat k větším (složitějším).
- Pokud algoritmus výpočtu „**zdola nahoru**“ nenajdeme, např. při řešení problému Hanojských věží, lze rekurzivitou odstranit pomocí zásobníku.

*Např. zásobník využijeme pro uložení stavu řešení problému.*



# Rekurze

*“To iterate is human, to recurse divine.”*

L. Peter Deutsch

<http://www.devtopics.com/101-great-computer-programming-quotes>

## Elegance vs obtížnost rekurze

*I've often heard people describe understanding recursion as one of those "got it" moments, when the universe opened its secret stores of knowledge and gifted the mind of a burgeoning developer with a very powerful tool. For me, recursion has always been hard. Each time I'm able to peer more into its murky depths, I am humbled to see how little I feel like I really appreciate and understand its power and elegance.*

Rick Winfrey, 2012

<http://selfless-singleton.rickwinfrey.com/2012/11/27/to-iterate-is-human-to-recurse-divine>

# Fibonacciho posloupnost

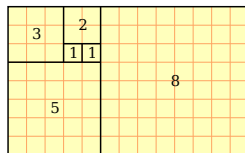
- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

*Nebo 0, 1, 1, 2, 3, 5, ...*

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

- pro  $F_1 = 1, F_2 = 1$

*Nebo  $F_1 = 0, F_2 = 1$*



- Nekonečná posloupnost přirozených čísel, kde každé číslo je součtem dvou předchozích.
- Limita poměru dvou následujících čísel Fibonacciho posloupnosti je rovna **zlatému řezu**.

- Sectio aurea – ideální poměr mezi různými délkami
- Rozdělení úsečky na dvě části tak, že poměr větší části ku menší je stejný jako poměr celé úsečky k větší části

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1,618\ 033\ 988\ 749\ 894\ 848 \dots$$

## Fibonacciho posloupnost – historie

- Indičtí matematici (450 nebo 200 BC)
- Leonardo Pisano (1175–1250) popis růstu populace králíků  
*italský matematik známý také jako Fibonacci*
  - $F_n$  – velikost populace po  $n$  měsících za předpokladu
    - První měsíc se narodí jediný pár.
    - Narozené páry jsou produktivní od 2. měsíce svého života.
    - Každý měsíc zplodí každý produktivní pár jeden další pár.
    - Králíci nikdy neumírají, nejsou nemocní atd.
- Henry E. Dudeney (1857–1930) – popis populace krav
  - „Jestliže každá kráva vyprodukuje své první tele (jalovici) za rok a poté každý rok jednu další jalovici, kolik budete mít krav za 12 let, jestliže žádná nezemře a na počátku budete mít jednu krávu?  
*Po 12 let je k dispozici jeden či více býků*

## Fibonacciho posloupnost – rekurzivně

- Platí:

$$f_0 = 1$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}, \text{ pro } n > 1$$

```
1 int fibonacci(int n) {  
2     return n < 2  
3         ? 1  
4         : fibonacci(n - 1) + fibonacci(n - 2);  
5 }
```

Zápis je elegantní, jak je však takový výpočet efektivní?

## Fibonacciho posloupnost – příklad 1/2

- Počet operací při výpočtu Fibonacciho čísla  $n$

```
1 long counter;
2
3 long fibonacciRecursive(int n) {
4     counter++;
5     return n < 2
6         ? 1
7         : fibonacciRecursive(n-1) + fibonacciRecursive(n-2);
8 }
9
10 long fibonacciIterative(int n) {
11     long fibM2 = 1l;
12     long fibM1 = 1l;
13     long fib = 1l;
14     for (int i = 2; i <= n; ++i) {
15         fibM2 = fibM1;
16         fibM1 = fib;
17         fib = fibM1 + fibM2;
18         counter += 3;
19     }
20     return fib;
21 }
```

lec06/DemoFibonacci.java

## Fibonacciho posloupnost – rekurzivně 2/2

```
1 public void start(String[] args) {
2     int n = args.length > 0 ? Integer.parseInt(args[0]) : 25;
3     counter = 0;
4     long fibR = fibonnaciR(n);
5     long counteR = counter;
6
7     counter = 0;
8     long fibI = fibonnaciI(n);
9     long counteI = counter;
10
11     System.out.println("Fibonacci number recursive: " + fibR);
12     System.out.println("Fibonacci number iteration: " + fibI);
13     System.out.println("Counter recursive: " + counteR);
14     System.out.println("Counter recursive: " + counteI);
15 }
```

lec06/DemoFibonacci.java

```
javac DemoFibonacci.java
java DemoFibonacci 30
```

```
Fibonacci number recursive: 1346269
Fibonacci number iteration: 1346269
Counter recursive: 2692537
Counter iteration: 8
```

## Fibonacciho posloupnost – rekurzivně vs iteračně

- Rekurzivní výpočet
  - Složitost roste exponenciálně s  $n \sim 2^n$
- Iterační algoritmus
  - Počet operací je proporcionální  $n \sim 3n$

`lec06/DemoFibonacciStats.java`, `lec06/fibonacci.sh`

- Skutečný počet operací závisí na konkrétní implementaci, programovacím jazyku, překladači a hardware
- Složitost algoritmů proto vyjadřujeme asymptoticky jako funkci velikosti vstupu
  - Například v tzv. „Big O” notaci
    - rekurzivní algoritmus výpočtu má složitost  $O(2^n)$
    - iterační algoritmus výpočtu má složitost  $O(n)$

*Efektivní algoritmy mají polynomiální složitost*



# Část II

## Příklady

## Pole reprezentující množinu prvočísel

- Vypsát všechna prvočísla menší nebo rovna zadané hodnotě  $\max$
- Algoritmus:
  1. Vytvoříme množinu obsahující všechna přirozená čísla od 2 do  $\max$
  2. Z množiny vypustíme násobky čísla 2
  3. Najdeme nejbližší číslo  $k$  tomu, jehož násobky jsme v předchozím kroku vypustili, a vypustíme všechny násobky tohoto čísla
  4. Opakujeme krok 3, dokud číslo, jehož násobky jsme vypustili, není větší než odmocnina z  $\max$
  5. Čísla, která v množině zůstanou, jsou hledaná prvočísla
- Pro reprezentaci množiny čísel použijeme pole prvků typu `boolean`, kde prvek pole `sieve[i]` udává, zda-li je celé číslo  $i$  v množině (`true`) nebo není (`false`), tj. zda-li je nebo není prvočíslem

# Eratosthenovo síto 1/2

```
1  boolean[] createSieve(int max) {
2      boolean[] sieve = new boolean[max + 1];
3      for (int i = 2; i <= max; ++i) {
4          sieve[i] = true;
5      }
6      int p = 2;
7      int pmax = (int)Math.sqrt(max);
8      do {
9          for(int i = p + p; i <= max; i += p) {
10             sieve[i] = false;
11         }
12         do {
13             p++;
14         } while (!sieve[p]);
15     } while (p <= pmax);
16     return sieve;
17 }
```

lec06/DemoSieveOfEratosthenes.java

## Eratosthenovo síto 2/2

```
1 void print(boolean[] sieve) {
2     for(int i = 2; i < sieve.length; ++i) {
3         if (sieve[i]) {
4             System.out.print(i + " ");
5         }
6     }
7 }
8
9 void start(int max) {
10    boolean[] sieve = createSieve(max);
11    System.out.print("Prime numbers from 2 to " + max + ":
12    ");
13    print(sieve);
14    System.out.println("");
15 }
```

```
javac DemoSieveOfEratosthenes.java
```

```
java DemoSieveOfEratosthenes
```

```
Prime numbers from 2 to 100: 2 3 5 7 11 13 17 19 23 29 31
37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

```
java DemoSieveOfEratosthenes 30
```

```
Prime numbers from 2 to 40: 2 3 5 7 11 13 17 19 23 29
```

[lec06/DemoSieveOfEratosthenes.java](#)

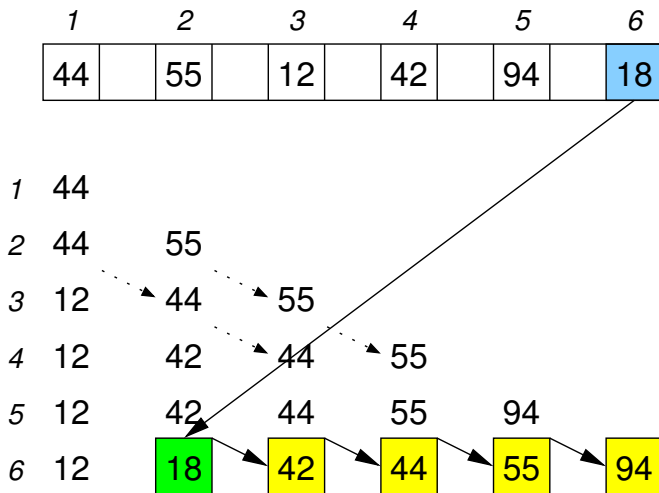
# Řazení pole

- Problém seřadit množinu prvků podle klíče (celého čísla)
- Posloupnost prvků  $q = \langle a_1, \dots, a_n \rangle$
- Délka posloupnosti  $q$  je  $|q| = n$
- Posloupnost  $q$  je seřazená, právě tehdy když
  - $|q| < 2$
  - $|q| \geq 2$  **klíč**( $a_1$ )  $\leq$  **klíč**( $a_2$ ) a posloupnost  $\langle a_2, \dots \rangle$  neobsahuje prvek  $a_1$ .
- Řazení polí je řazením na místě

*Pro jednoduchost v příkladech třídíme jen celá čísla, prakticky však zpravidla třídíme klíče datových položek.*

## Třídění přímým vkládáním – Insert Sort 1/2

## Příklad - Třídění přímým vkládáním



## Třídění přímým vkládáním – Insert Sort 2/2

**for**  $i \in \langle 2, n \rangle$

„vlož  $a_i$  na patřičné místo mezi  $a_1, \dots, a_i$ ”

### Příklad - Algoritmus

```
1 void insertSort(int[] array) {
2     int x;
3     int j;
4     for (int i = 1; i < array.length; i++) {
5         x = array[i];
6         j = i - 1;
7         while (j >= 0 && x < array[j]) {
8             array[j + 1] = array[j];
9             j--;
10        }
11        array[j + 1] = x;
12    }
13 }
```

- Třídění binárním vkládáním - vkládání provádíme do již zatříděného pole  $\langle a_1, \dots, a_{i-1} \rangle$ .

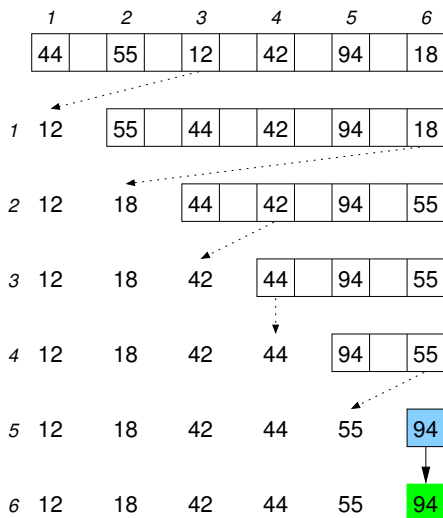
# Řazení přímým výběr – Select Sort 1/3

```
for  $i \in \langle 1, n \rangle$  {  
    „najdi index  $k$  nejmenšího prvku v  $\langle a_i, \dots, a_n \rangle$ ,  
     $a_k = \min\langle a_i, \dots, a_n \rangle$ ,  
    zaměň prvky  $a_i, a_k$ ”  
}
```



# Řazení přímým výběr – Select Sort 2/3

## Příklad - Řazení přímým výběrem Select Sort



# Řazení přímým výběr – Select Sort 3/3

## Příklad implementace

```
1 void selectSort(int[] array) {
2     for (int i = 0; i < array.length-1; ++i) {
3         int minIDX = i;
4         for (int j = i + 1; j < array.length; ++j) {
5             if (array[minIDX] > array[j]) {
6                 minIDX = j;
7             }
8         }
9         swap(minIDX, i, array);
10    }
11 }
```

*Zde pro jednoduchost třídíme jen celá čísla, prakticky však zpravidla třídíme klíče datových položek.*

- Složitost  $O(n^2)$ , kde  $n$  je počet položek

## Řazení rozdělováním – Quick Sort

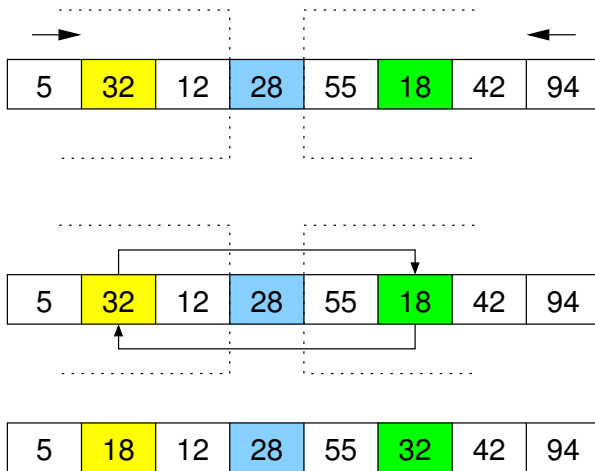
„Nejvýznamnější výměny jsou ty na velké vzdálenosti.”

- Pole rozdělíme nějakým prvkem  $x$  (pivot).
- Nalevo od prvku  $x$  umístíme všechny prvky menší než  $x$ .
- Napravo od prvku  $x$  umístíme všechny prvky větší než  $x$ .
- Rozdělení opakujeme pro pole tvořené prvky nalevo od  $x$  a pro pole napravo od  $x$ .
- Postup opakujeme dokud nerozdělujeme jednoprvkové úseky.
- Strategie „rozděl a panuj.”

*Tuto strategii můžeme implementovat rekurzí*

## Rozdělení pole

## Příklad - rozdělení



## Řazení rozdělením – Quick Sort 1/2

```
1 void qsortPart(int l, int h, int[] array) {
2     int i = l; //lower index
3     int j = h; //higher index
4     int pivot = array[l + (h - 1) / 2];
5     while (i <= j) {
6         while(array[i] < pivot) {
7             i++;
8         }
9         while(array[j] > pivot) {
10            j--;
11        }
12        if (i <= j) {
13            swap(i++, j--, array);
14        }
15    }
16    if (l < j) {
17        qsortPart(l, j, array);
18    }
19    if (i < h) {
20        qsortPart(i, h, array);
21    }
22 }
```

## Řazení rozdělením – Quick Sort 2/2

```
1 void quickSort(int[] array) {  
2     qsortPart(0, array.length-1, array);  
3 }
```

- Složitost pro nejnepríznivější případ je  $O(n^2)$
- Vhodně zvolený pivot výrazně snižuje časovou náročnost
- Randomizovaná varianta (vstupní pole permutováno a pivot je volen náhodně) –  $O(n \log_2 n)$

## Řazení Select Sort vs Quick Sort

### ■ Příklad spuštění Select Sort vs Quick Sort

```
javac DemoSort.java && java DemoSort 10
Values: 10 24 41 17 20 31 42 5 24 46
Values Select Sort: 5 10 17 20 24 24 31 41 42 46
Values Quick Sort: 5 10 17 20 24 24 31 41 42 46

java DemoSort 40000
Select Sort required time 3415 ms
Quick Sort required time 10 ms
```

[lec06/DemoSort.java](#)

### ■ Jednoduchý test výpočetní náročnosti můžeme udělat porovnáním výpočetních časů

*Např. využitím `System.currentTimeMillis`*

### ■ Skutečné výpočetní nároky se mohou lišit podle vstupních dat

*Zkuste zjistit počet operací pro vstupní pole setříděné sestupně*

### ■ Asymptotická složitost udává očekávané výpočetní nároky pro velké vstupy

*Reálné testování v tzv. „mikro benchmarks“ může být zvláště v Javě zavádějící (HotSpot)*

# Zápis algoritmu Quick Sort

## Implementace Quick Sort v programovacím jazyku Haskell

```
1 qsort []      = []
2 qsort (x:xs) = qsort (filter (< x) xs) ++ [x] ++ qsort
   (filter (>= x) xs)
```

*Pro zajímavost*



## Shrnutí přednášky

## Diskutovaná témata

- Rekurze a rekurzivní algoritmy
- Příklady rekurzivních algoritmů:
  - Výpočet faktoriálu
  - Výpis posloupnosti čísel
  - Hanojské věže
  - Fibonacciho posloupnost
- Reprezentace množiny polem
  - Příklad implementace v algoritmu Eratosthenovo síto
- Rozklad na prvočinitele
- Příklad rekurze v řazení
- **Příště: Objektově orientované programování v Javě**