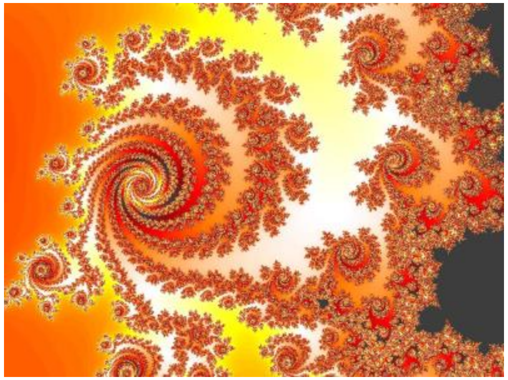
1. prednáška



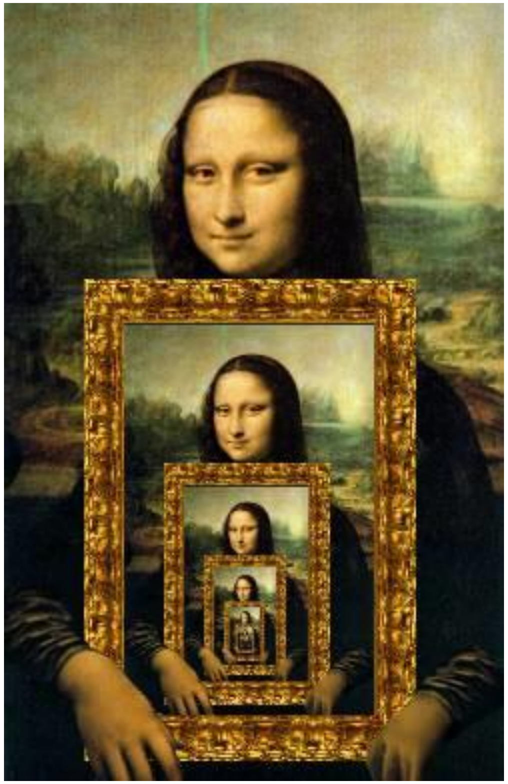
Rekurzia

alebo

viď rekurzia

Čo je rekurzia?

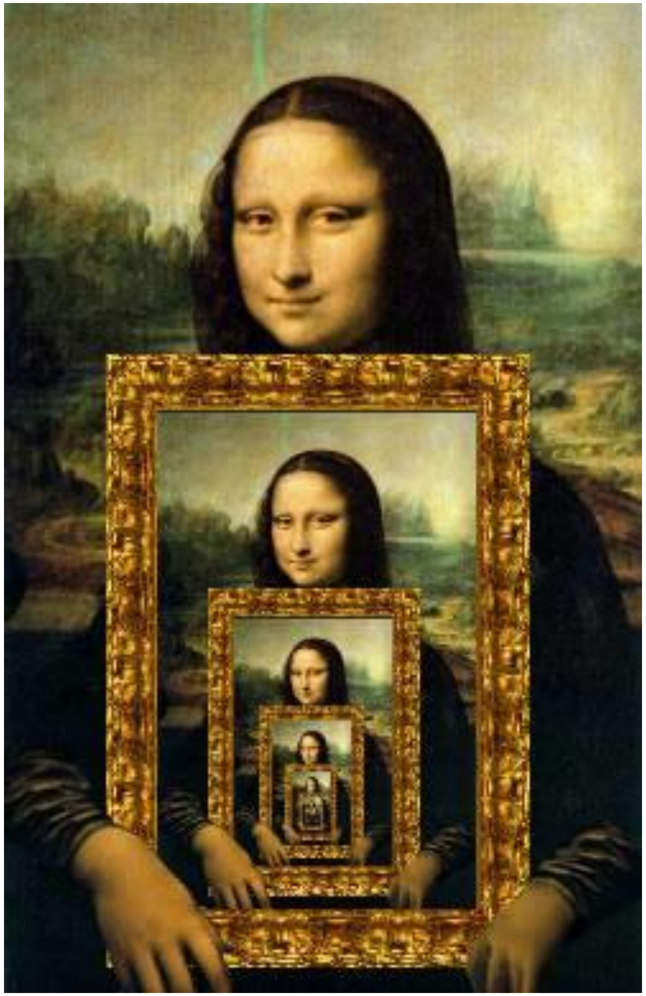
- **Rekurzia** = **definícia** niečoho pomocou **samého seba** (lat. *recurrere* = bežať naspäť)
- Rekurzia môže mať **mnoho podôb:**
  - zrkadlá postavené oproti sebe
  - fraktály
    - Kochova vločka
  - rekurzívne definované postupnosti
    - $a_n = a_{n-1} + a_{n-2}$, $a_0 = 0$, $a_1 = 1$
  - rekurzívne funkcie
    - faktoriál: $n! = n * (n-1)!$
      $0! = 1$



Rekurzia v obraze

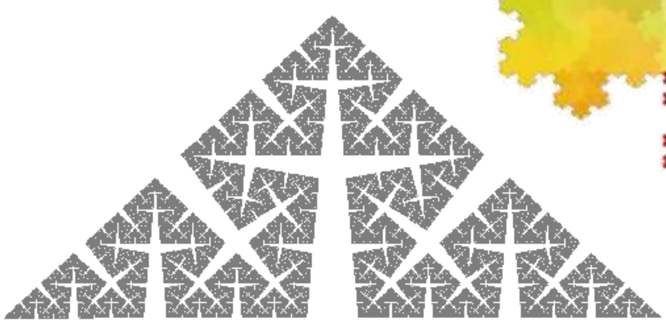
- Ako by sme opísali obraz „Mona Líza s obrazom“?
- Na obraze „Mona Líza s obrazom“ je namaľovaná Mona Líza, ktorá drží v rukách obraz „Mona Líza s obrazom“.
- Formálnejšie:

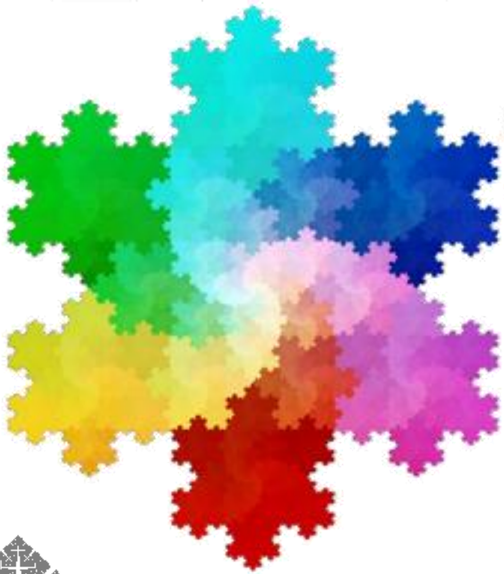
  OML je (ML + menší OML)



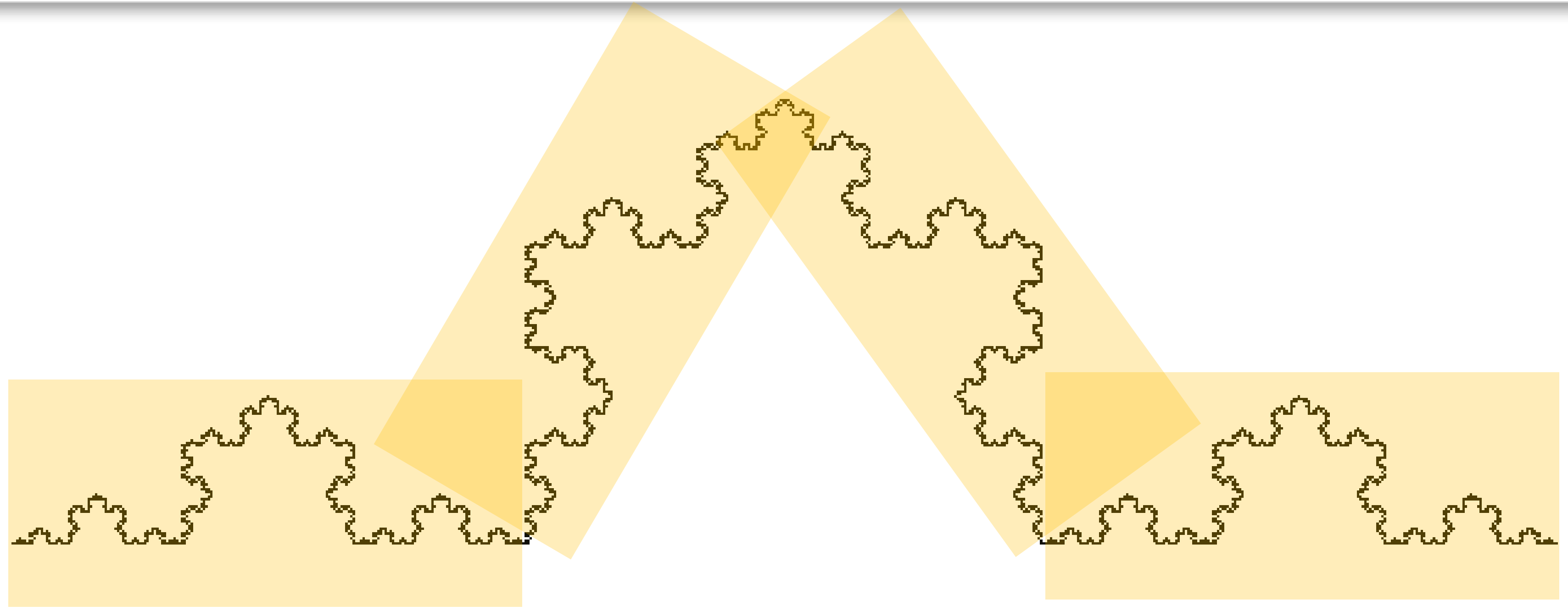
Fraktály

- **Fraktál** je **geometrický útvar**, ktorý možno rozdeliť **na časti**, z ktorých **každá je zmenšeninou** celého geometrického útvaru.





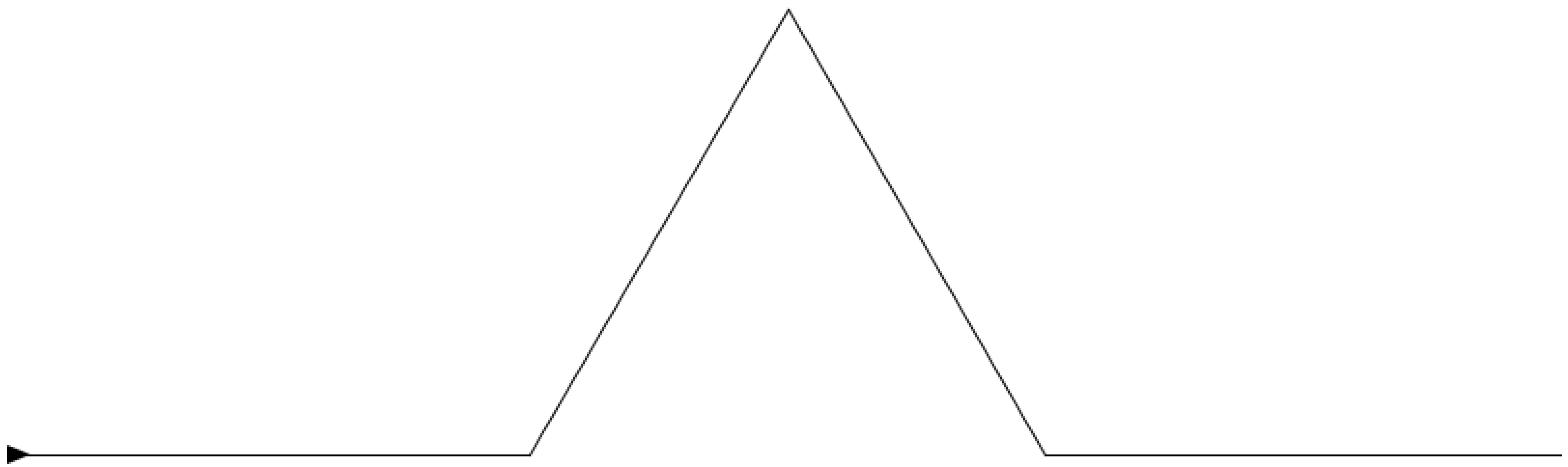
Kochova krivka (1)

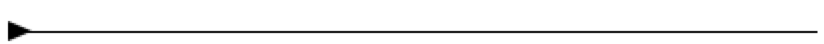


Kochova krivka sa zkladá zo 4 „menších" Kochových kriviek.

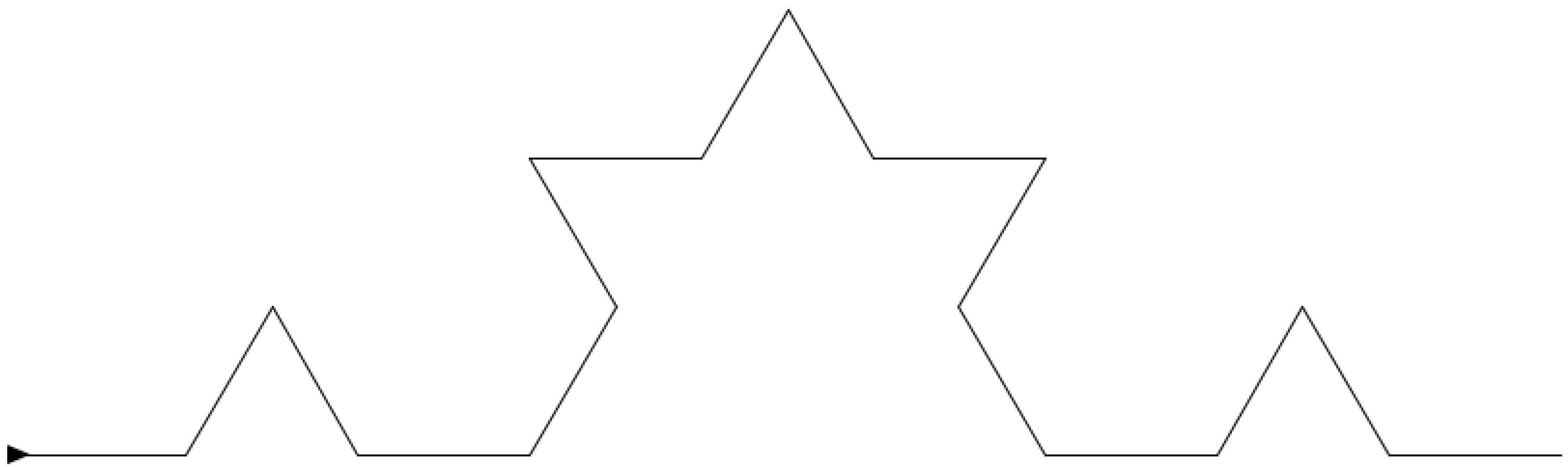
„menší" = menej čiar?, kratšie čiary?, ...

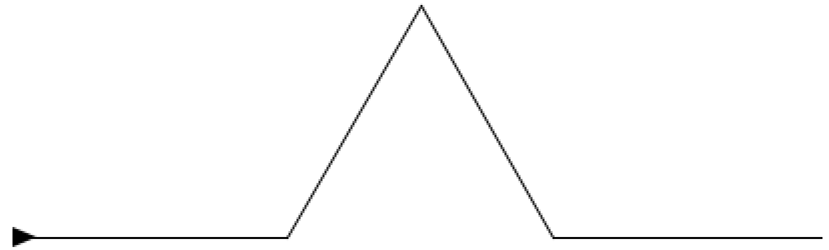
Kochova krivka (2)



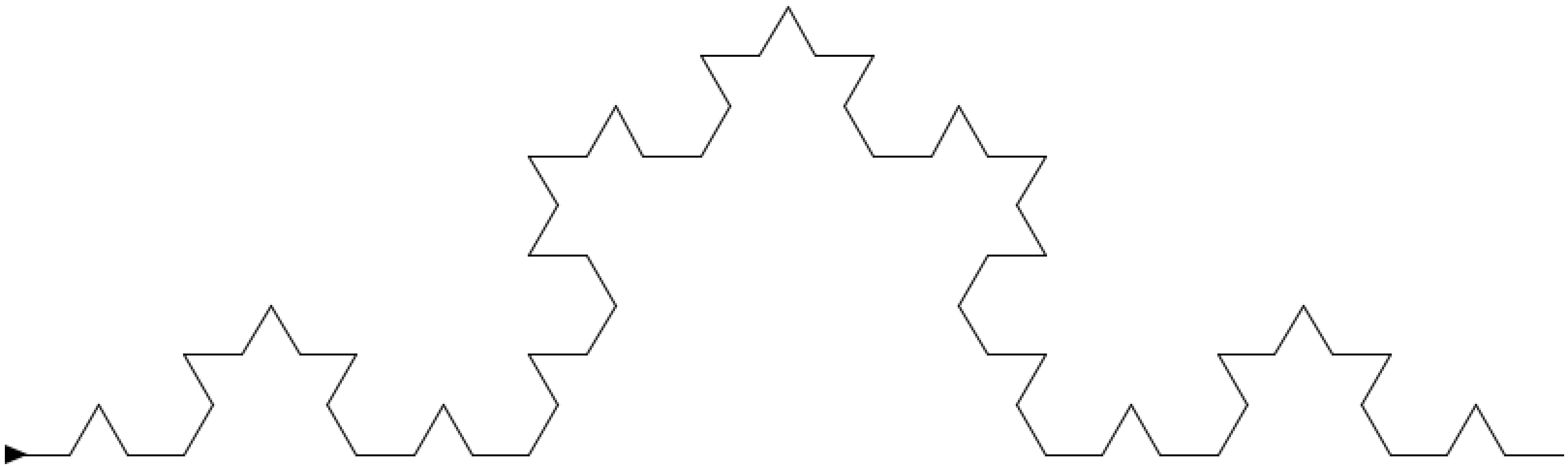


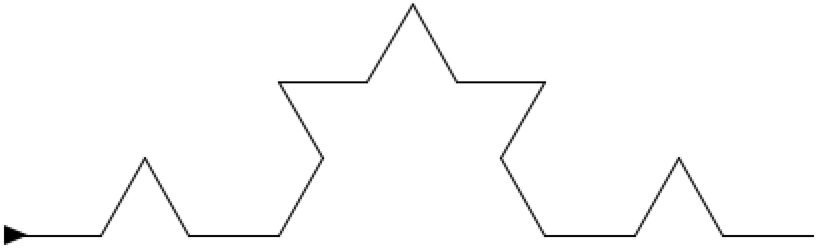
Kochova krivka (2)



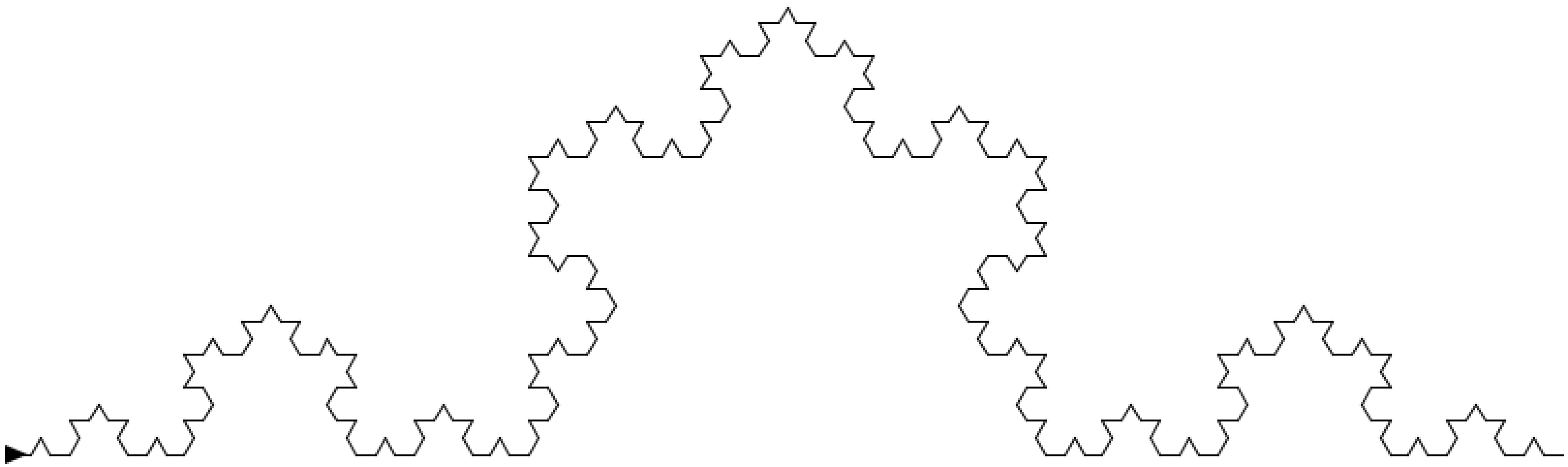


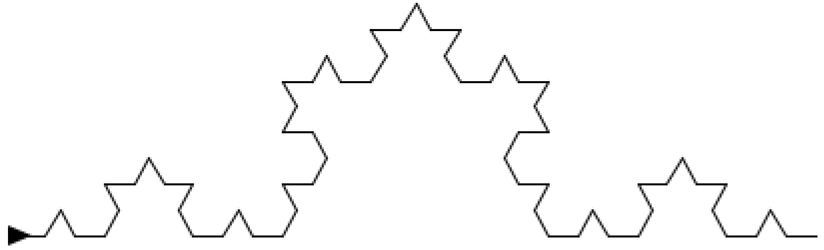
Kochova krivka (2)



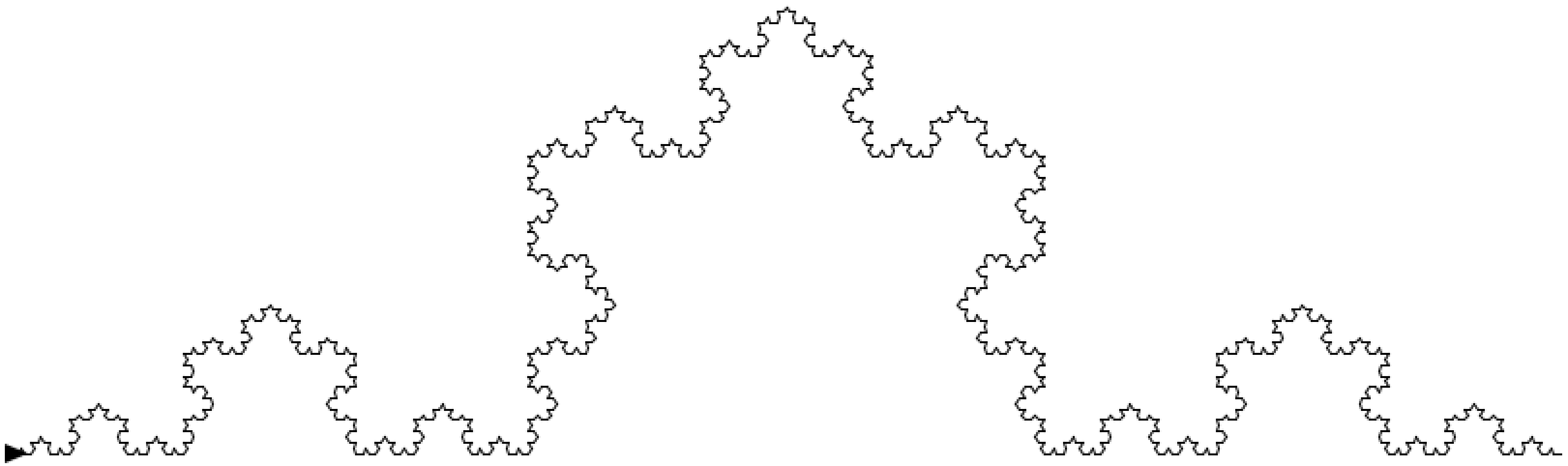


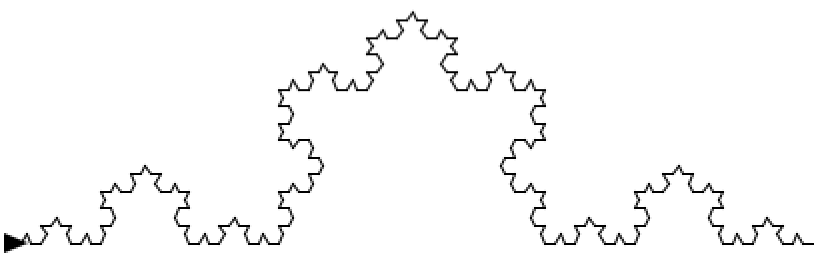
Kochova krivka (2)



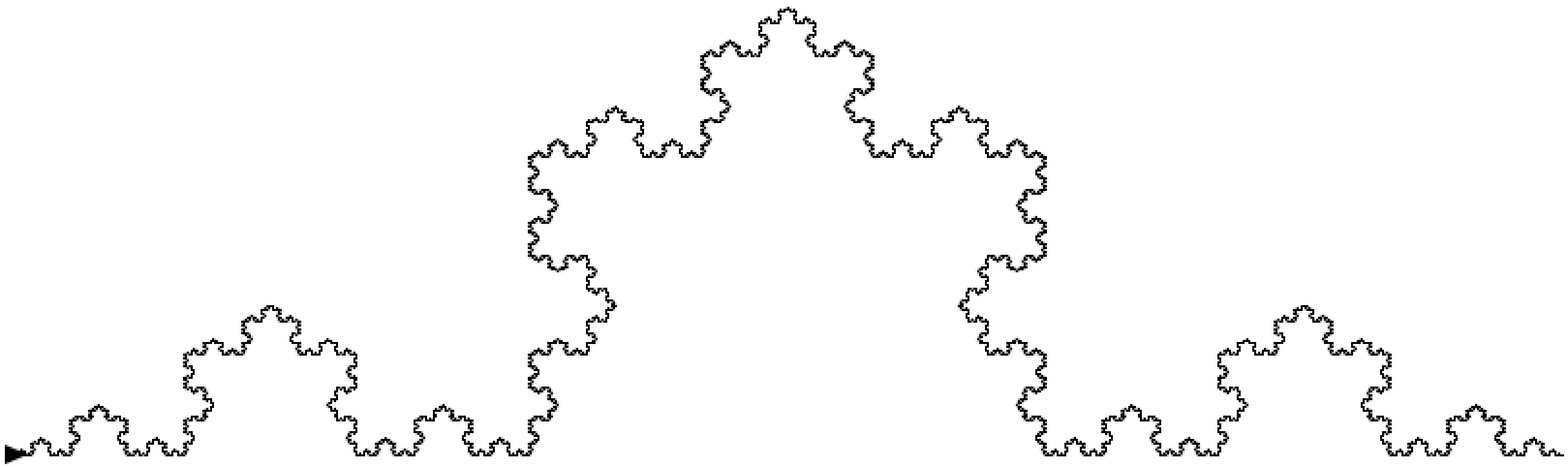


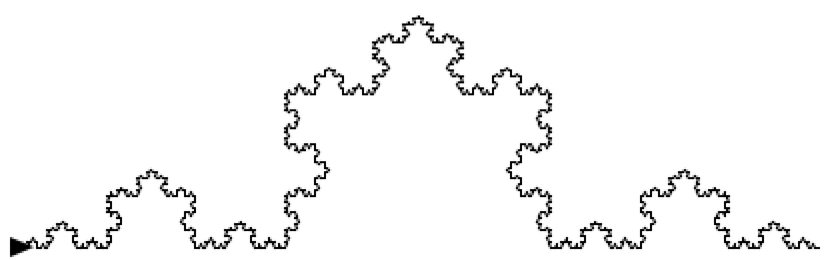
Kochova krivka (2)





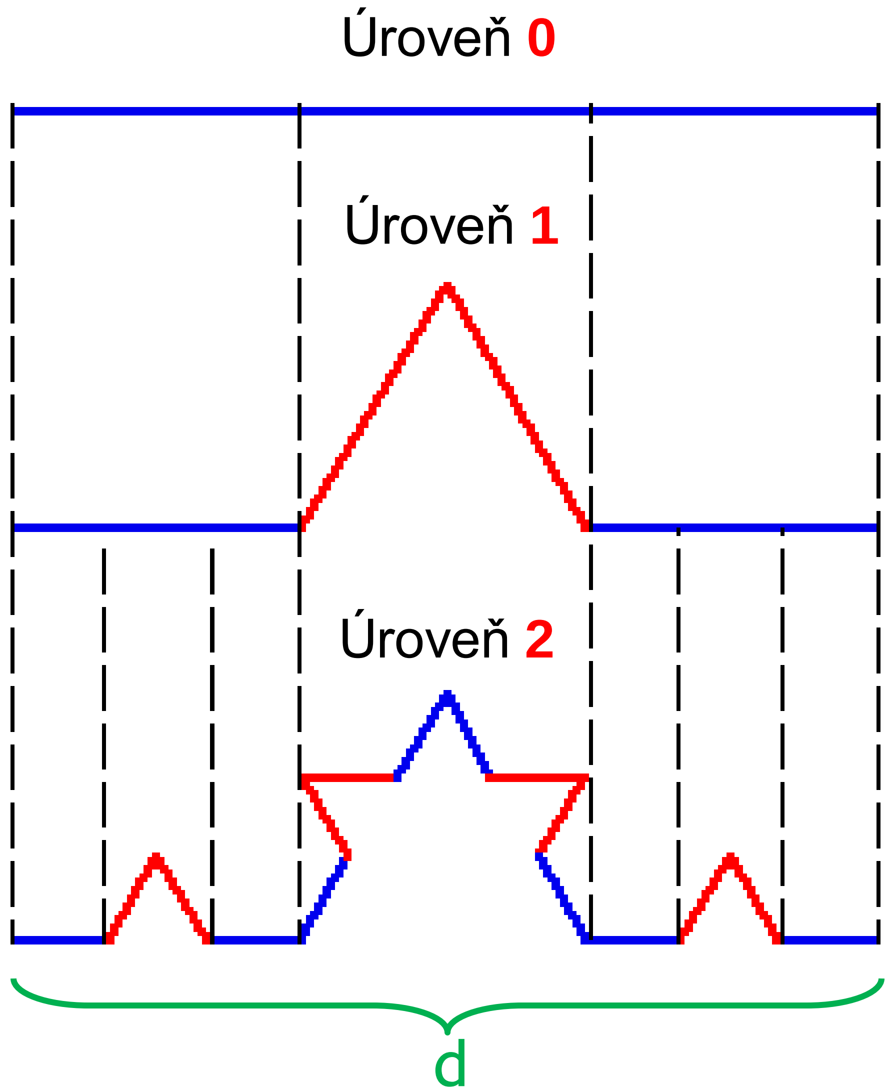
Kochova krivka (2)





Kochova krivka (3)

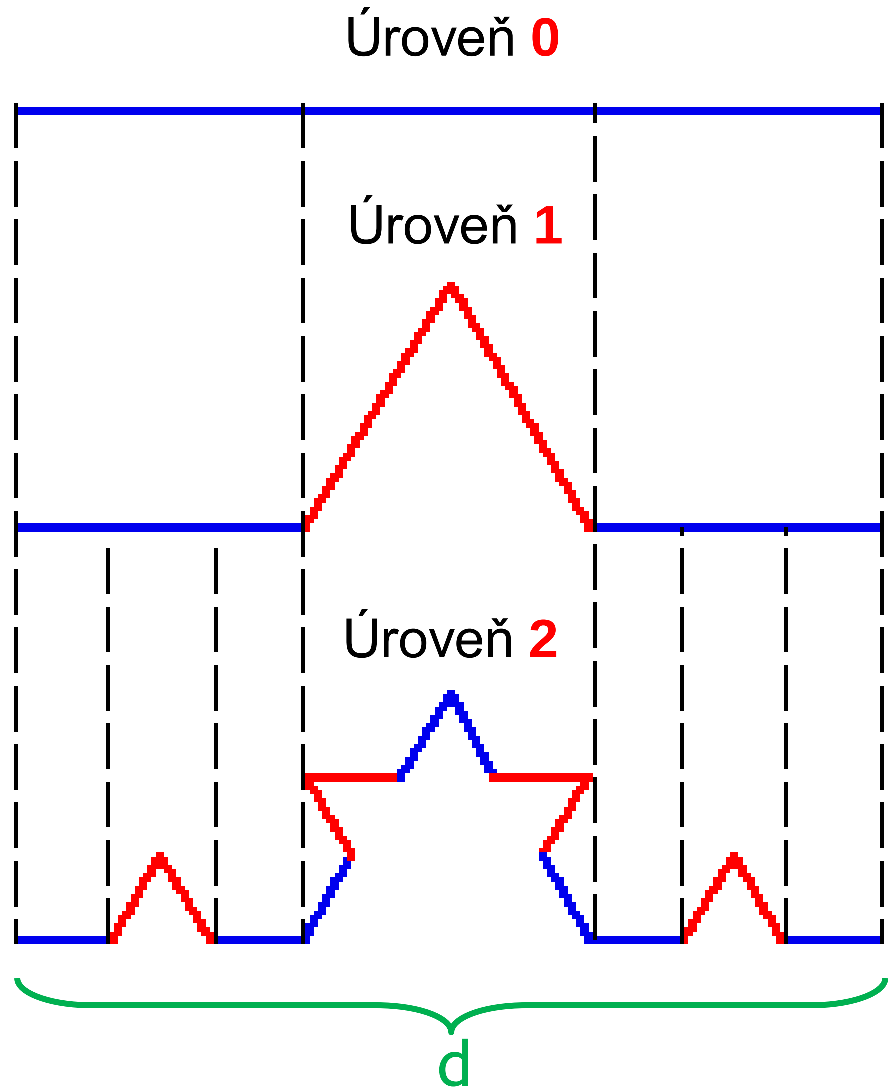
```
KK(2, d) {
    KK(1, d/3);
    turn(-60)
    KK(1, d/3);
    turn(120)
    KK(1, d/3);
    turn(-60);
    KK(1, d/3);
}
```



Kochova krivka (3)

Korytnačie príkazy na nakreslenie krivky úrovne u > 0 a dĺžky d:

```
KK(u, d) {
    KK(u-1, d/3);
    turn(-60)
    KK(u-1, d/3);
    turn(120)
    KK(u-1, d/3);
    turn(-60);
    KK(u-1, d/3);
}
```



Kochova krivka (4)

Korytnačie príkazy na nakreslenie krivky úrovne u > 0 a dĺžky d:

```
KK(u, d) {
    KK(u-1, d/3);
    turn(-60)
    KK(u-1, d/3);
    turn(120)
    KK(u-1, d/3);
    turn(-60);
    KK(u-1, d/3);
}
```

KK(u, d) je definovaná pomocou **KK(u-1, d/3)**, t.j. Kochova krivka je charakterizovaná Kochovou krivkou.

KK(0, d) je čiara dĺžky d.

Kochova krivka v Jave

```
public void kochovaKrivka(int u, double d) {
  if (u == 0) {
    this.step(d);
  } else {
    this.kochovaKrivka(u-1, d/3);
    this.turn(-60);
    this.kochovaKrivka(u-1, d/3);
    this.turn(120);
    this.kochovaKrivka(u-1, d/3);
    this.turn(-60);
    this.kochovaKrivka(u-1, d/3);
  }
}
```

Báza rekurzie (triviálny prípad)

Rekurzívne volania. **Metóda volá samu seba.**

Rekurzívne metódy

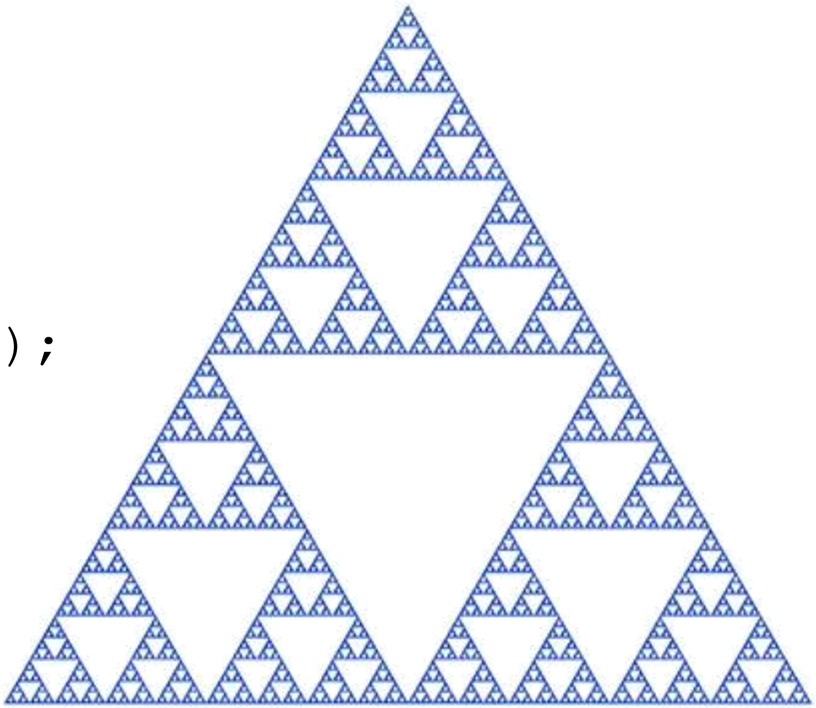
- **Rekurzívna metóda** je metóda, ktorá vo svojej implementácii (priamo alebo nepriamo) **volá samu seba**.
- **Nepriama rekurzia:**
  - X volá Y a Y volá X
  - X volá Y, Y volá Z a Z volá X, ...
- Typická **šablóna** rekurzívnej metódy:
  - otestovanie, či nie je **báza rekurzie/triviálny prípad**
    - ak je triviálny prípad, tak sa zrealizujú príslušné príkazy/akcie a vykonávanie metódy končí
  - realizácia **rekurzívnych volaní** „rekurzívny krok“

Ďalšie fraktály

- Sierpińskeho trojuholník:

```
public void sierpinski(int u, double d) {
  if (u == 0) {
    return;
  }

  for (int i=0; i<3; i++) {
      this.sierpinski(u-1, d/2);
      this.step(d);
      this.turn(120);
  }
}
```



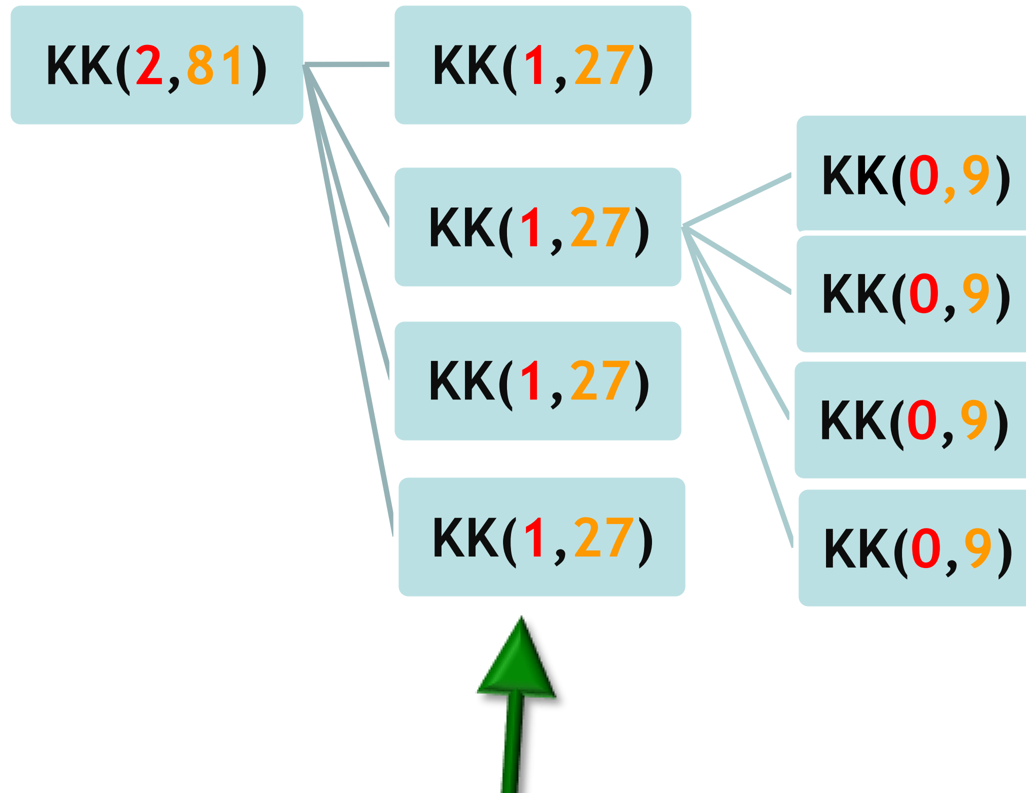
Aké parametre sú dobré?

- Testujme rôzne parametre ...

- Pozorovanie:
  - čas kreslenia **výrazne záleží** od parametra u – úroveň
  - parameter **d nevplýva** (pozorovateľne) na čas kreslenia

Štruktúra rekurzívnych volaní

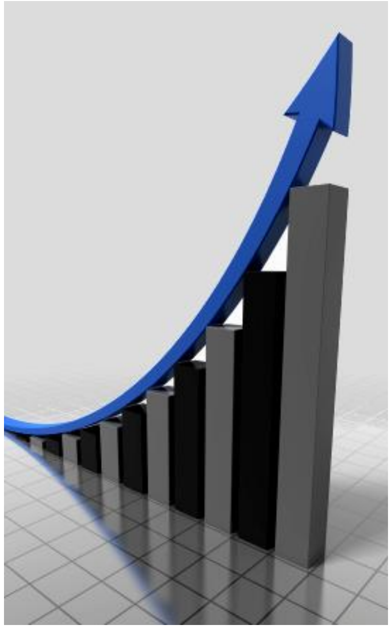
```
KK(u, d) {
    KK(u-1, d/3);
    turn(-60)
    KK(u-1, d/3);
    turn(120)
    KK(u-1, d/3);
    turn(-60);
    KK(u-1, d/3);
}
```



Každé KK(1, 27) má za následok
4 volania KK(0, 9)
celkom 4 x 4 = 16 volaní KK(0, 9)

Koľko čiar sa nakreslí?

- Čiara sa kreslí, iba keď u = 0
- KK(2, 81)
  - 16 volaní KK(0, 9) = 16 čiar = 4^2 čiar
- KK(3, 243) = 4 volania KK(2, 81)
  - 4 x 16 volaní KK(0, 9) = 64 čiar = 4^3 čiar
- KK(4, 729) = 4 volania KK(3, 243)
  - 4 x 64 volaní K(0, 9) = 256 čiar = 4^4 čiar
- KK(u, d) = 4^u čiar



Exponenciálna závislosť od u

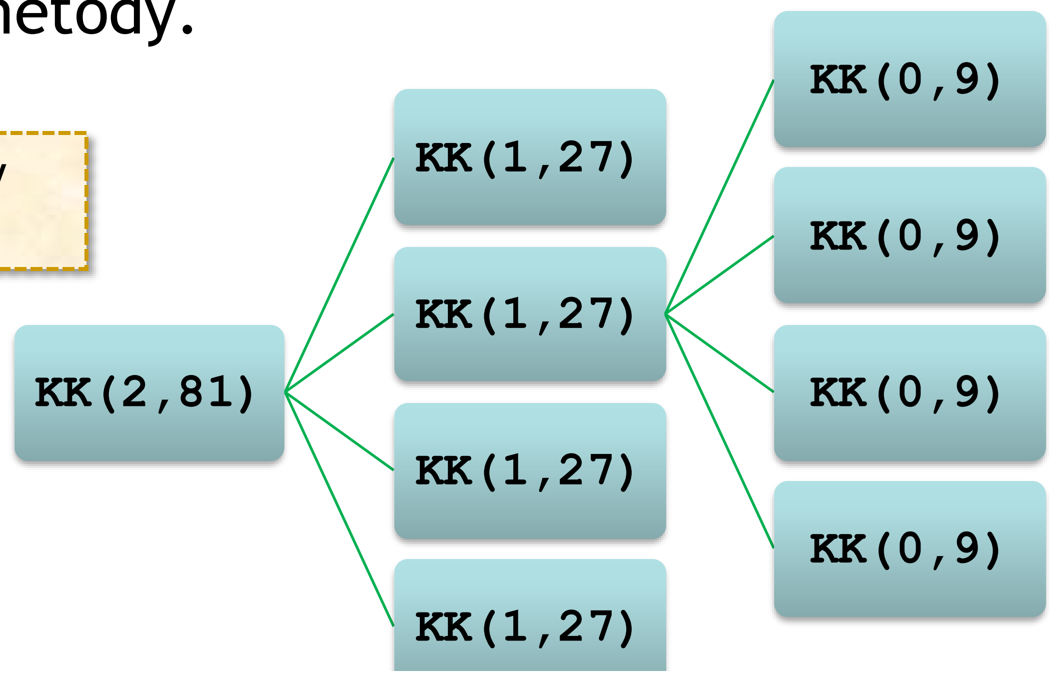
Aké úrovne sú dobré?

- KK(4, d) = 256 čiar
- KK(20, d) = 1099511627776 čiar
- KK(u, d) = 4^u čiar
  - na nakreslenie čiary treba aspoň jednu inštrukciu procesora
  - 10 GHz procesor < 10^{10} inštrukcií/čiar za sekundu
- KK(40, d) = 4^{40} čiar = 16^{20} čiar > 10^{20} čiar
  - 10^{20} čiar / 10^{10} čiar za sekundu = 10^{10} sekúnd
  - 1 rok = 31536000 sekúnd ~ tretina z 10^8 sekúnd
  - 10^{10} sekúnd je viac ako 300 rokov

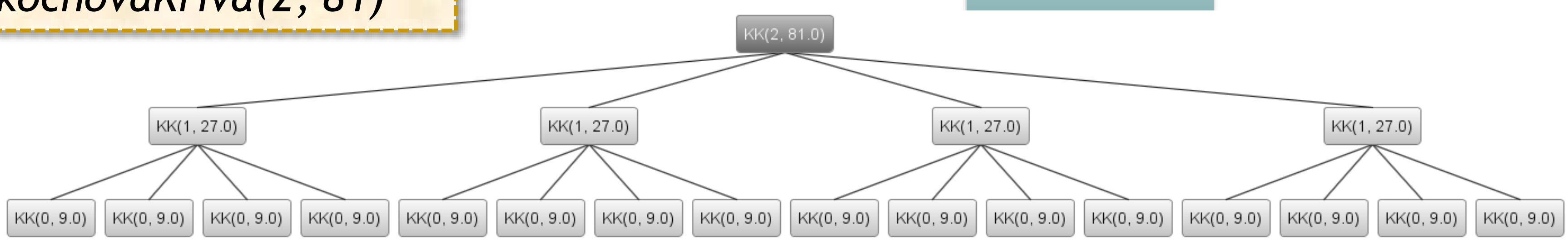
Strom volaní

- **Strom volaní** – zakreslenie toho, aké volania metód a s akými hodnotami parametrov sa realizujú pri nejakom konkrétnom volaní metódy.

Časť stromu volaní metódy *kochovaKriva(2, 81)*

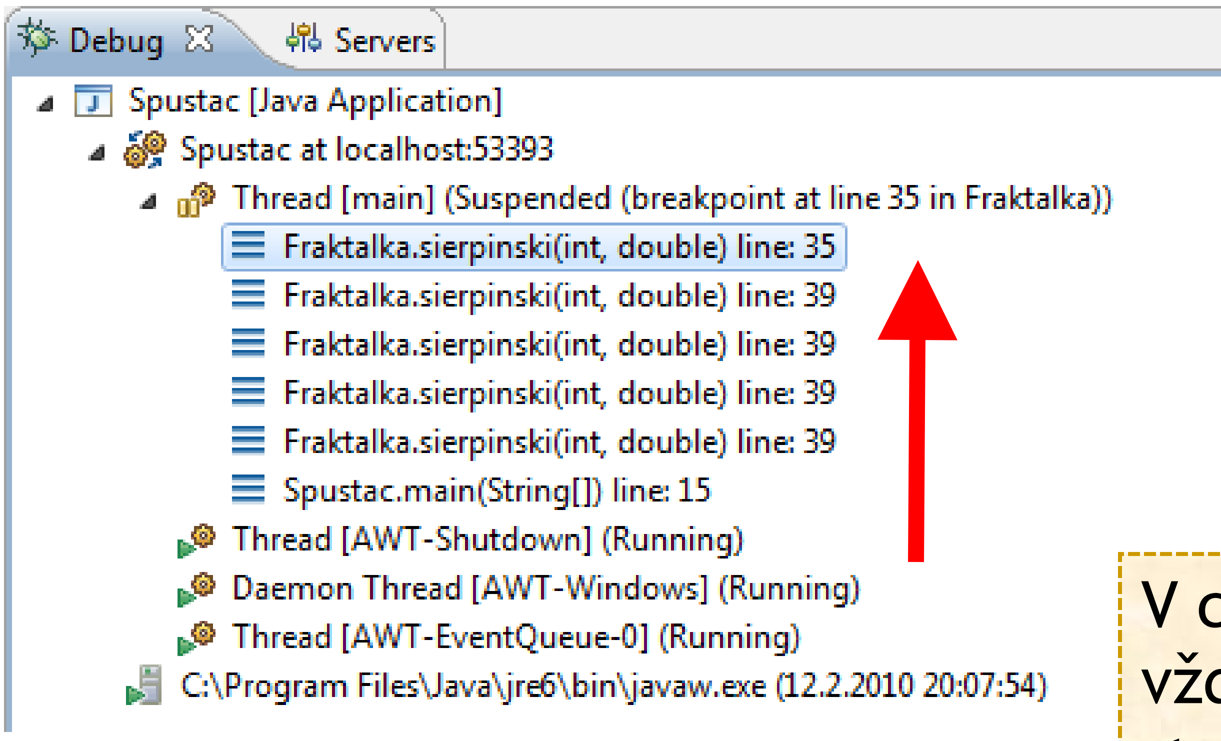


Strom volaní metódy *kochovaKriva(2, 81)*



Call stack

- **Call stack** – aktuálna postupnosť vzájomných volaní metód vo vykonávaní (kto koho volal).



V call stack-u je vždy nejaká vetva v strome volaní.

Čo obsahuje Call Stack?

- Položkami *Call Stack*-u sú **záznamy o volaniach metód**, ktorých vykonávanie **začalo**, no ešte **nebolo ukončené** (`return`-om alebo `}`)

- Aj keď máme veľa volaní tej istej metódy, každé jedno volanie vytvára nové lokálne premenné
  - pozorujme pri debugovaní...
  - každé vykonávanie metódy má svoj vlastný „kontext“, ktorý obsahuje tieto lokálne premenné

Strom volaní a call stack

- Počet úrovní v strome volaní zodpovedá maximálnej výške call stacku.
- Každé **volanie** metódy **pridáva** položku **v call stacku**, návrat z metódy položku odoberá.
- Maximálna výška call stack-u a teda aj **počet rekurzívnych vnorení je obmedzený** pamäťou vyhradenou pre call stack
  - default 320-1024kB, ale záleží od OS a verzie Javy

```
Exception in thread "main"
   java.lang.StackOverflowError
at Fraktalka.sierpinski(Fraktalka.java:36)
at Fraktalka.sierpinski(Fraktalka.java:36)
```

Ukážka...

StackOverflowError

- Výnimka ***StackOverflowError*** nastáva vtedy, ak je v call stacku **priveľa otvorených volaní** metód
- V praxi to zvyčajne znamená, že máme rekurziu, ktorá sa „zacyklila“ (nekonečné vnáranie)
  - Neustálym rekurzívnym vnoreniam zabraňuje iba správne naprogramované obslúženie triviálneho prípadu („dna“ rekurzie)
- Typická chyba pri použití rekurzie je **zabudnuté** alebo **nesprávne** naprogramované **spracovanie bázy rekurzie** (triviálneho prípadu).

Fraktály trochu inak

- **Alternatíva na zastavenie** rekurzívnych volaní bez určenia maximálnej úrovne rekurzívneho vnorenia:

```
public void kochovaKrivka(double dlzka) {
  if (dlzka < 1) {
    this.step(dlzka);
    return;
  }

  this.kochovaKrivka(dlzka/3); this.turn(-60);
  this.kochovaKrivka(dlzka/3); this.turn(120);
  this.kochovaKrivka(dlzka/3); this.turn(-60);
  this.kochovaKrivka(dlzka/3);
}
```

Končíme, keď tvar je už príliš malý na ďalšie kreslenie.

Rekurzia v matematike

- Rekurzívne definície sú v matematike časté:
  - **rekurzívne definované** postupnosti
  - **rekurzívne definovateľné** matematické funkcie

- Prečo **matematici používajú rekurziu?**
  - jednoduchší zápis
  - lepšie zachytenie vlastností pojmu/funkcie
  - ideálne na dokazovanie matematickou indukciou

Statické metódy a premenné

- Statické metódy a premenné:
  - označujú sa kľúčovým slovom **static**
  - **static** = „patriaci triede“
  - statická premenná je premenná triedy, všetky objekty danej triedy ju „zdieľajú“
  - statická metóda je metóda triedy, **volá sa nad triedou**, nie nad objektom danej triedy
    - môže volať iné statické metódy a používať statické premenné
  - budeme ich používať v niektorých našich malých experimentálnych programčekoch

Faktoriál

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

```java
public static int faktorial(int n) {
   if (n == 0)
      return 1;
   else
      return n * faktorial(n-1);
}
```

Báza rekurzie

Rekurzívne volanie

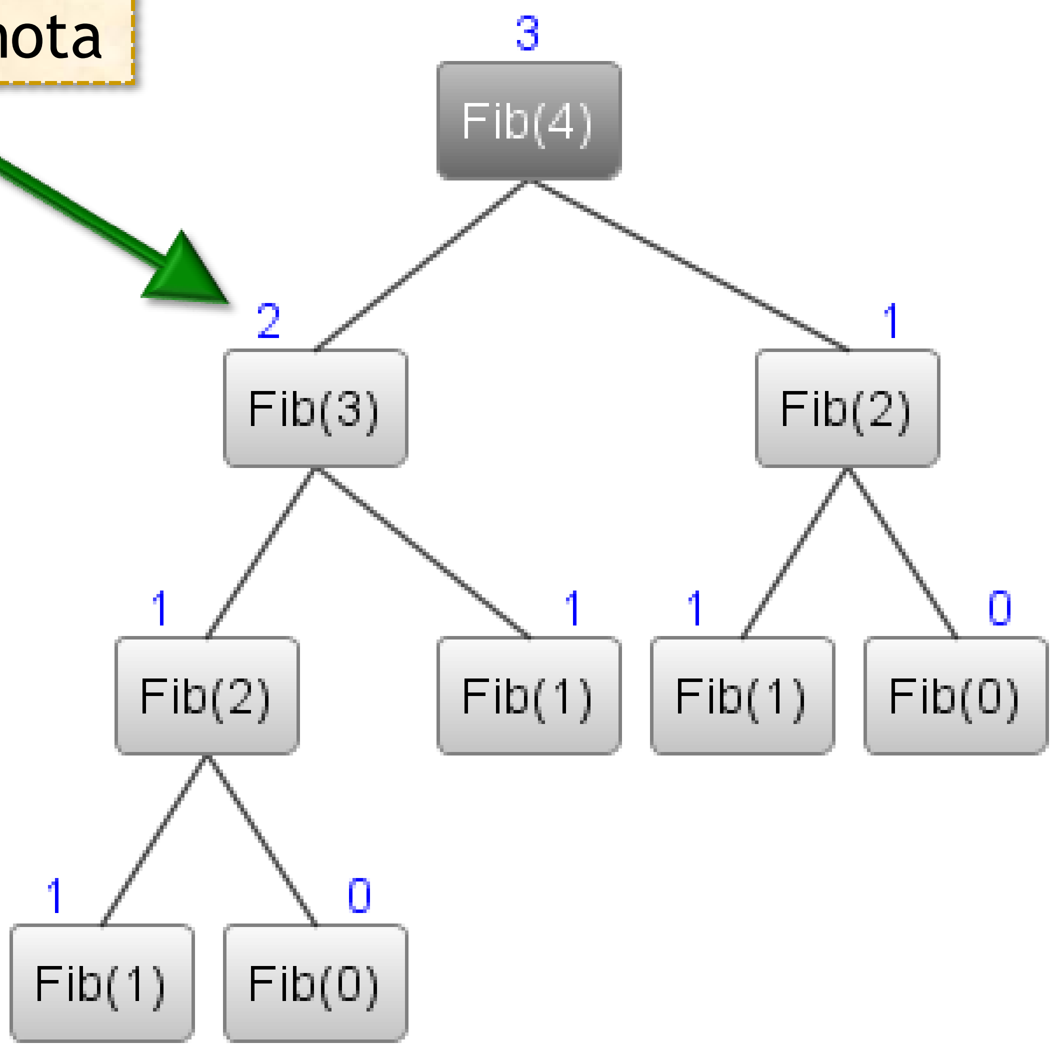
Fibonacciho postupnosť

- $F_0 = 0,\ F_1 = 1$
- $F_i = F_{i-1} + F_{i-2}$ *0, 1, 1, 2, 3, 5, 8,13,21, ...*

```java
public static int fibonacci(int n) {
  if (n == 0)
    return 0;
  if (n == 1)
    return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

Strom volaní Fibonacciho 4

Vrátená hodnota



Rekurzia v algoritmoch

- Rekurzívne riešenie je **základný stavebný kameň** väčšiny algoritmov.
- Idea:
  - Problém „úrovne **u**“ chceme **zredukovať** na niekoľko menších problémov „menšej úrovne“
  - zvyčajne „úroveň“ = veľkosť vstupu

Rekurzívny prístup k nájdeniu efektívneho algoritmu neznamená ešte rekurzívnu implementáciu !!!

Maximum v podpoli

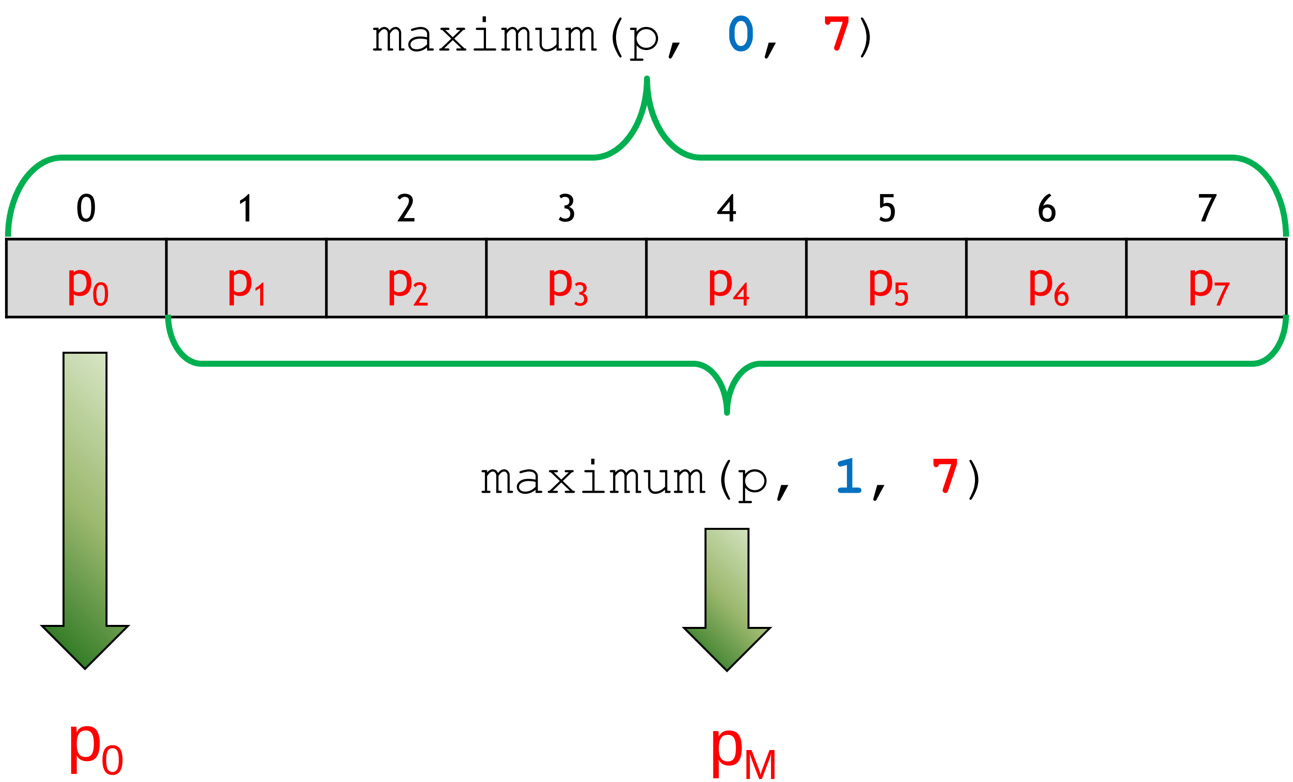
```
public static int maximum(int[] p,
   int odIdx, int poIdx)
```

- Funkcia má vrátiť maximálnu hodnotu v poli ***p*** na políčkach s indexami od ***odIdx*** po ***poIdx***

- Maximum z prvých 5 hodnôt v poli:
  - `maximum(p, 0, 4)`
- Maximum v celom poli:
  - `maximum(p, 0, p.length-1)`

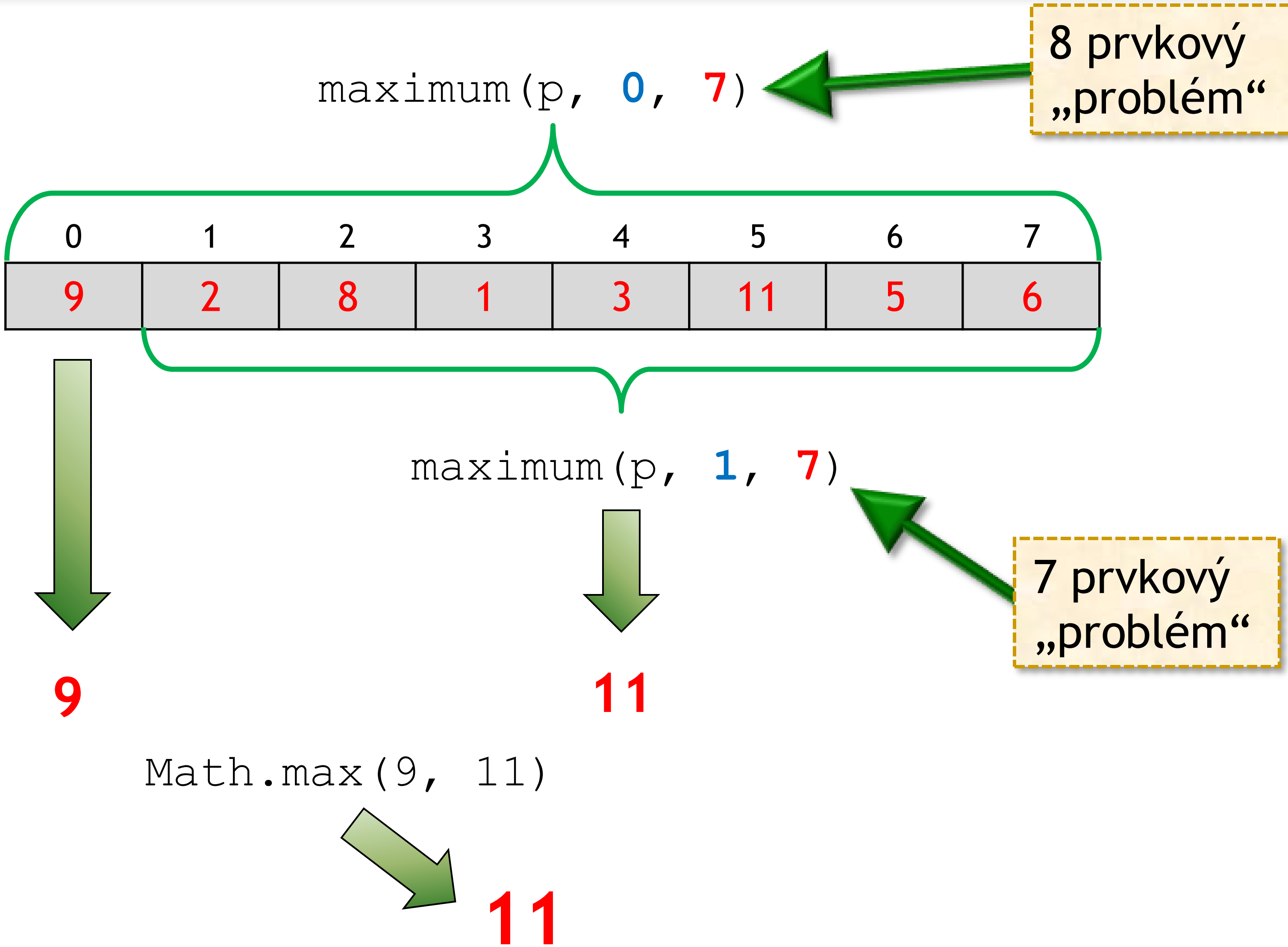
Myšlienková výzva: Ide to spraviť bez použitia cyklov?

Maximum v poli inak



`Math.max(`p_0`, `p_M`)`

Maximum v poli inak



Maximum v poli inak

```
public static int maximum(int[] p,
   int odIdx, int poIdx)
```

- Ak **odIdx = poIdx**, tak maximum je p[odIdx]
- Ak **odIdx < poIdx**, tak:
  - zabudneme na prvý prvok p_0 podpoľa na indexe *odIdx*
  - nájdeme maximum p_M vo zvyšnom podpoli **Rekurzia**
  - ako definitívnu odpoveď vrátime väčšie z čísel p_0 a p_M

Maximum v poli inak

```
public static int maximum(int[] p, int odIdx,
   int poIdx) {
   if (odIdx == poIdx)
      return p[odIdx];
   else
      return Math.max(p[odIdx],
                  maximum(p, odIdx+1, poIdx));
}
```

Báza rekurzie

Zmenšenie problému

Rekurzívne volanie

Na cvičeniach rozanalyzujete toto riešenie a ukážete si efektívnejšie riešenie založené na delení problému na „polovice“

Čo si treba pamätať

- Pri rekurzii nie je **opatrnosti** nikdy dosť (pretečenie call stack-u)
- Rekurzia môže byť **výpočtovo** veľmi **drahá**
  - „toto v **reálnych programoch neskúšajte**, pokiaľ nemusíte“
  - neskôr sa naučíme, kedy je OK a kedy nie
- Rekurzívne riešenie = **rozklad na podproblémy**
- Úvahy o rekurzívnom riešení zvyčajne vedú k efektívnemu nerekurzívnemu riešeniu ...