Rekurze.

Princip a použití rekurze. Základní problémy řešitelné rekurzí.

Tomáš Bayer | bayertom@natur.cuni.cz

Katedra aplikované geoinformatiky a kartografie, Přírodovědecká fakulta UK.

Obsah přednášky

1. Rekurze a její vlastnosti
   - Výpočet faktoriálu a Fibonacciho čísel rekurzí
2. Příklady použití rekurze
   - Hledání maxima rekurzí
   - Binární hledání
   - Úloha Hanojských věží
   - Kochova vločka
3. Vhodnost/nevhodnost rekurze

1. Rekurze

Sebeopakování, realizováno bez použití cyklu.
Projevuje se u funkcí / datových struktur.
V informatice často používaná strategie.

Datová struktura/funkce v definice odkazuje na sebe nebo strukturu/funkci podobného typu.

Rekurzivní funkce f:
Volají samy sebe nebo podobné funkce; volání součástí její definice (umístěno v těle).
Nové volání funkce provedeno před ukončením jejího běhu.
Funkce musí obsahovat ukončovací podmínku (podobně jako cyklus).

Rekurze vhodná pro **specifický typ problémů:**
Dekomponovatelné úlohy, alternativa cyklů, rekurentní funkce generující posloupnost $\{a_1, ..., a_n\}$

$$a_n = f(a_{n-1}, ..., a_1).$$

Existuje “snadný” vztah mezi problémy řešenými v následujících krocích.

Rekurzivní algoritmy: využívají techniku rekurze.
Stručnější, avšak méně čitelný zápis.

Nevýhoda rekurze:
Při každém volání f dochází k vytvoření nové sady lokálních proměnných.
Za běhu existuje několik sad lokálních proměnných $\Rightarrow$ výpočetně náročné.

2. Vlastnosti rekurze

Dělení rekurze:

Přímá rekurze: funkce opakovaně volá sama sebe: $f \to f$.

Nepřímá (kruhová) rekurze: cyklické volání několik funkcí:

$$f_1 \to f_2 \to \ldots \to f_n \to f_1 .$$

Lineární rekurze: jedno rekurzivní volání.

Kaskádová rekurze: více než jedno rekurzivní volání.

Podmínka ukončení rekurze:

Rekurzivní algoritmus musí obsahovat ukončovací podmínku.

V opačném případě dojde k nekonečné rekurzi !!! $\Rightarrow$ Stack Overflow.

V určitém místě musí dojít k takovému dořešení problému, které nebude rekurzivní.

Přímý[1] vs. zpětný[2] chod:

[1] Opakované volání funkce před ukončením jejího běhu až do nerekurzivního dořešení.

[2] Postupné dosazování nerekurzivního dořešení, ukončování běhu volaných funkcí v opačném pořadí.

3. Vztah rekurze a iterace

Iterace:

Opakování bloku příkazů za stanovených podmínek.

Podmínka opakování bloku předchází (`for`, `while`).

```
while condition:        #Podminka
     do something       #Blok prikazu
```

Rekurze:

Opakování bloku příkazů.

Podmínka (neúplná / úplná) součástí bloku.

```
def f(param):
   if condition():          #Podminka
      do something          #Blok prikazu
      f(param)              #Rekurzivni volani funkce
   else:
      do something else
```

Mezi rekurzí a iterací úzký vztah.

Vzájemná převoditelnost, u kaskádové rekurze však netriviální.

Rekurze mnohdy přímočařejší než iterace, avšak výpočetně náročnější.

4. Přímá rekurzivní funkce

Obecný tvar rekurzivní funkce:

```
f(n):                #Rekurzivni funkce: prima rekurze
    Cn               #Blok prikazu, fce n
    if p(n):         #Podminka, zavisi na n
        D            #Nerekurzivni doreseni, nezavisi na n
    else:
        f(n - 1)     #Rekurzivni volani fce f
    En;              #Blok prikazu, fce n
```

nebo:

```
f(n) = {Cn; p(n) ? D : f(n - 1);En }
```

Posloupnost rekurzivních volání funkce F nad množinou N.

```
f(n)->f(n-1)->...f(1)
```

V praxi zjednodušené schéma:

```
f(n) = {p(n) ? D : f(n - 1)}
```

5. Posloupnost rekurzivních volání

Posloupnost rekurzivních volání:

```
#Rekurzivni volani: primy chod
fn -> C_n   f(n - 1)
fn -> C_n   f(n - 1)  C_n-1  f(n - 2)
fn -> C_n   f(n - 1)  C_n-1  f(n - 2)  C_n-2  f(n - 3)
...
fn -> C_n   f(n - 1)  C_n-1  f(n - 2)  ...   C_2  f(1)  C_1 #Konec rekurze
#Nerekurzivni doreseni a zpetny chod
fn -> C_n   f(n - 1)  C_n-1  f(n - 2)  ...   C_2  f(1)  C_1   D   E_1
fn -> C_n   f(n - 1)  C_n-1  f(n - 2)  ...   C_2  f(1)  C_1   D   E_1  E_2
...
fn -> C_n   f(n - 1)  C_n-1  f(n - 2)  ...   C_2  f(1)  C_1   D   E_1  E_2 ... E_n
```

Bloky C, E závislé na n volány n-krát.

Nerekurzivní dořešení D voláno pouze 1x.

Blok E volán až ve zpětném chodu.

Možno znázornit stromem.

Ukázka posloupnosti volání v Pythonu

```
def f(n):
    print("C(" + str(n)+")")                    #Blok Cn
    if (n == 1):
        print("D");                             #Blok D
    else:
        print("f(" + str(n-1) + ")")
        f(n - 1)                                #Rekurz. volani
    print("E(" + str(n) + ")")                  #Blok E
```

Pak:

```
C(4)
f(3)
    C(3)
    f(2)
        C(2)
        f(1)
            C(1)
            D               #Zpetny chod
            E(1)
        E(2)
    E(3)
E(4)
```

7. Výpočet faktoriálu s použitím rekurze

Faktoriál čísla n označujeme jako $n!$, definice

$$n! = \begin{cases} n(n-1)!, & \text{pro } n > 1, \\ 1, & \text{pro } n = 1. \end{cases}$$

Zápis s použitím rekurze (rekurentní)

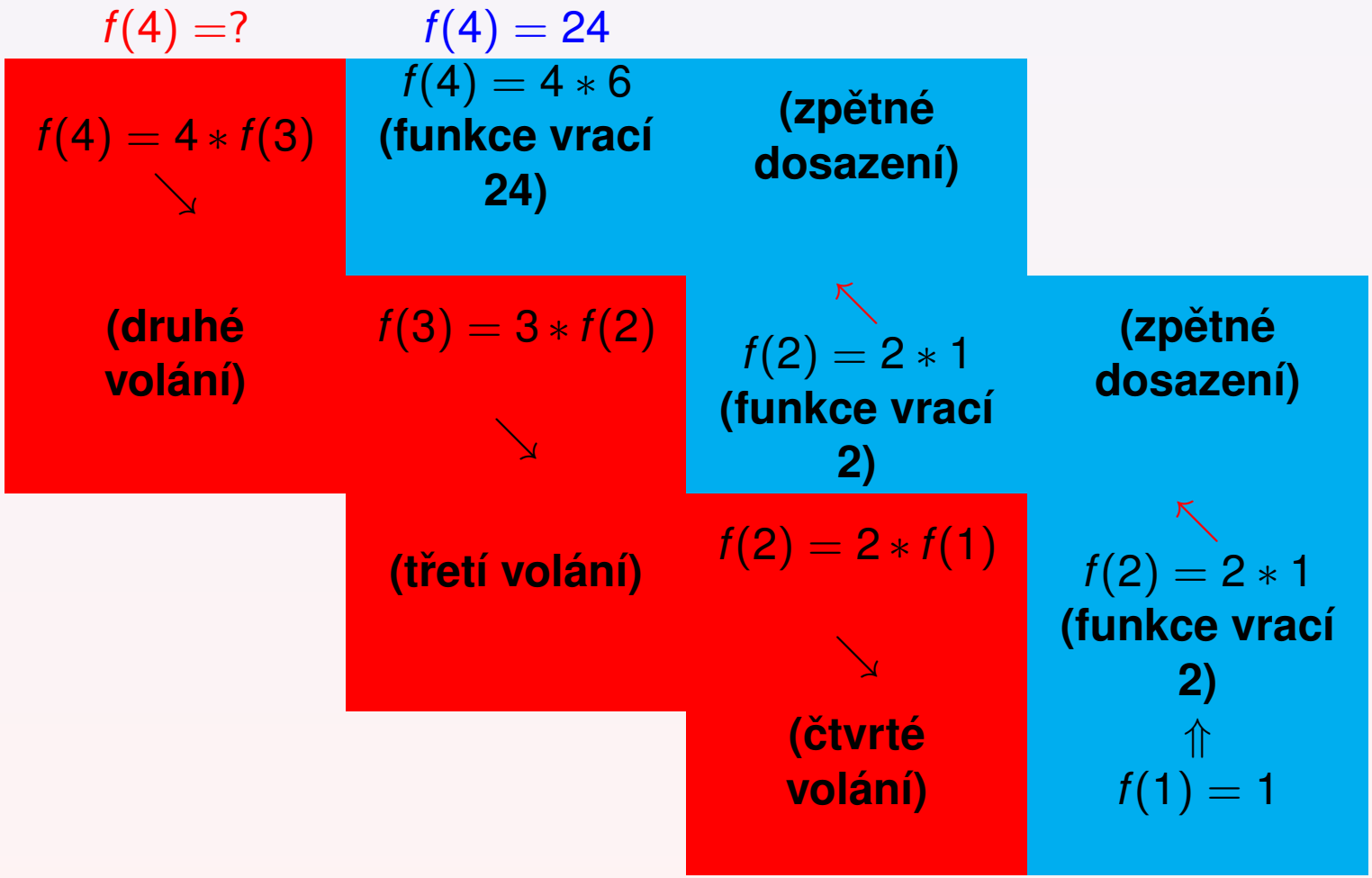
$$f(n) = \begin{cases} nf(n-1), & \text{pro } n > 1, \\ 1, & \text{pro } n = 1. \end{cases}$$

Rekurzivní volání f nad *zmenšující se* množinou prvků.

Zjednodušení problému (předpoklad efektivního použití rekurze).

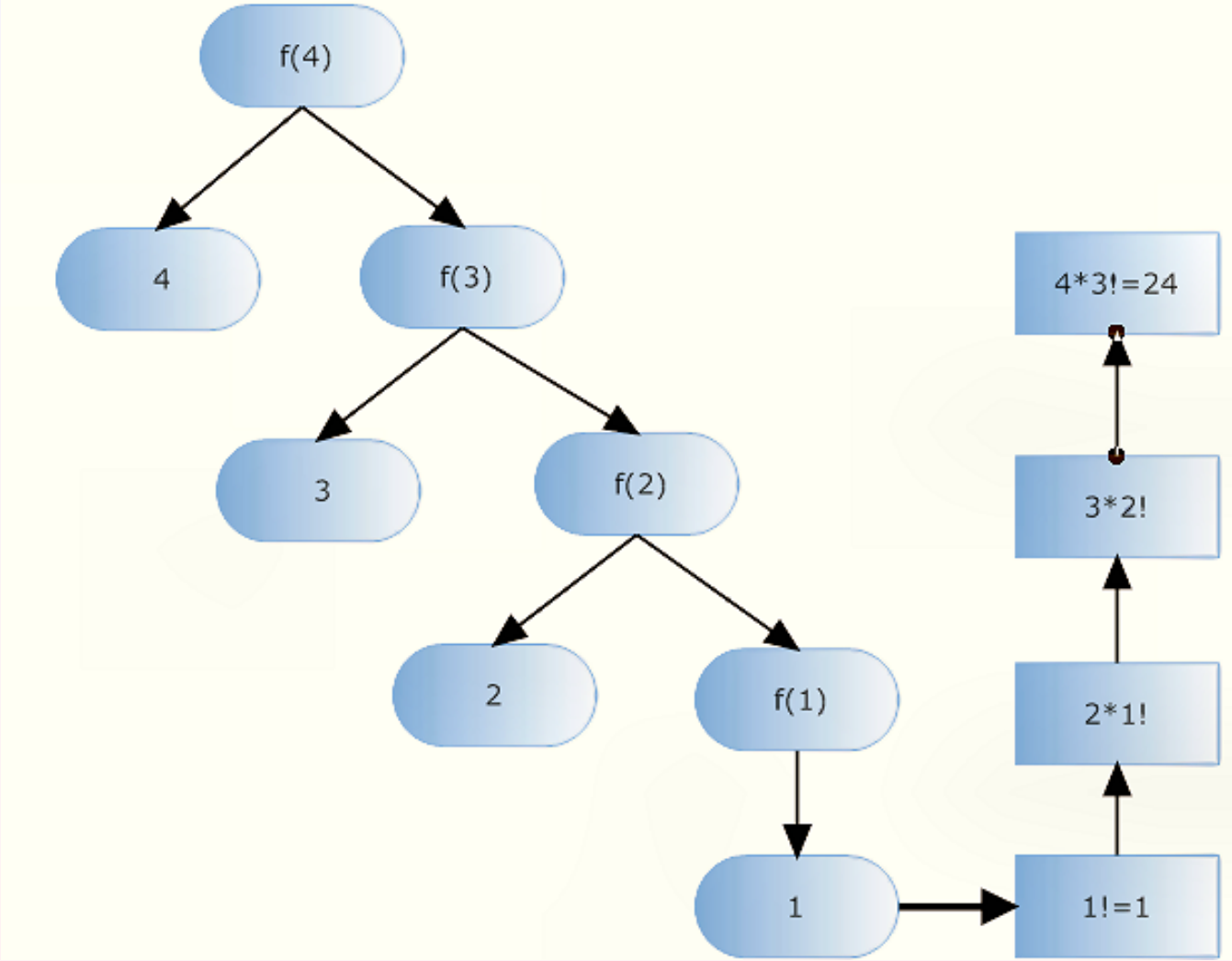
Splněny předpoklady.

8. Znázornění průběhu rekurze



9. Znázornění průběhu rekurze

Strom rekurzivního volání funkce *n*!



10. Zápis zdrojového kódu

Výpočet faktoriálu s použitím rekurze:

```
def f(n):
    if n > 1:      #Ukoncovaci podminka
        return n * f(n-1)
    else:
        return 1;  #Nerekurzivni doreseni
```

V každém kroku se velikost n zmenší o 1.

Jaká je doba běhu algoritmu?

11*. Výpočet doby běhu

Časová funkce $T(n)$ v závislosti na velikosti vstupu n:

$$
\begin{aligned}
T(0) &= T(1) = 1, \\
T(n) &= T(n-1) + 3, \\
&= T(n-2) + 3 + 3, \\
&= T(n-3) + 3 + 3 + 3, \\
&\dots \\
&= T(n-i) + 3i, \\
&= T(n-n) + 3n, \\
&= T(0) + 3n, \\
&= 1 + 3n, \\
&\doteq 3n.
\end{aligned}
$$

Doba běhu lineární funkcí velikostí vstupu.
Horní odhad složitosti $O(n)$.

12. Fibonacciho posloupnost

Fibonacciho funkce

$$fib(n) = \begin{cases} fib(n-1) + fib(n-2), & \text{pro } n > 1, \\ 1, & \text{pro } n = 1, \\ 1, & \text{pro } n = 0. \end{cases} \quad (1)$$

Přirozeně rekurzivní definice, generuje Fibonacciho posloupnost:

$$\{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ..\}$$

Použití: zlatý řez.

Výpočet rekurzí velmi neefektivní:

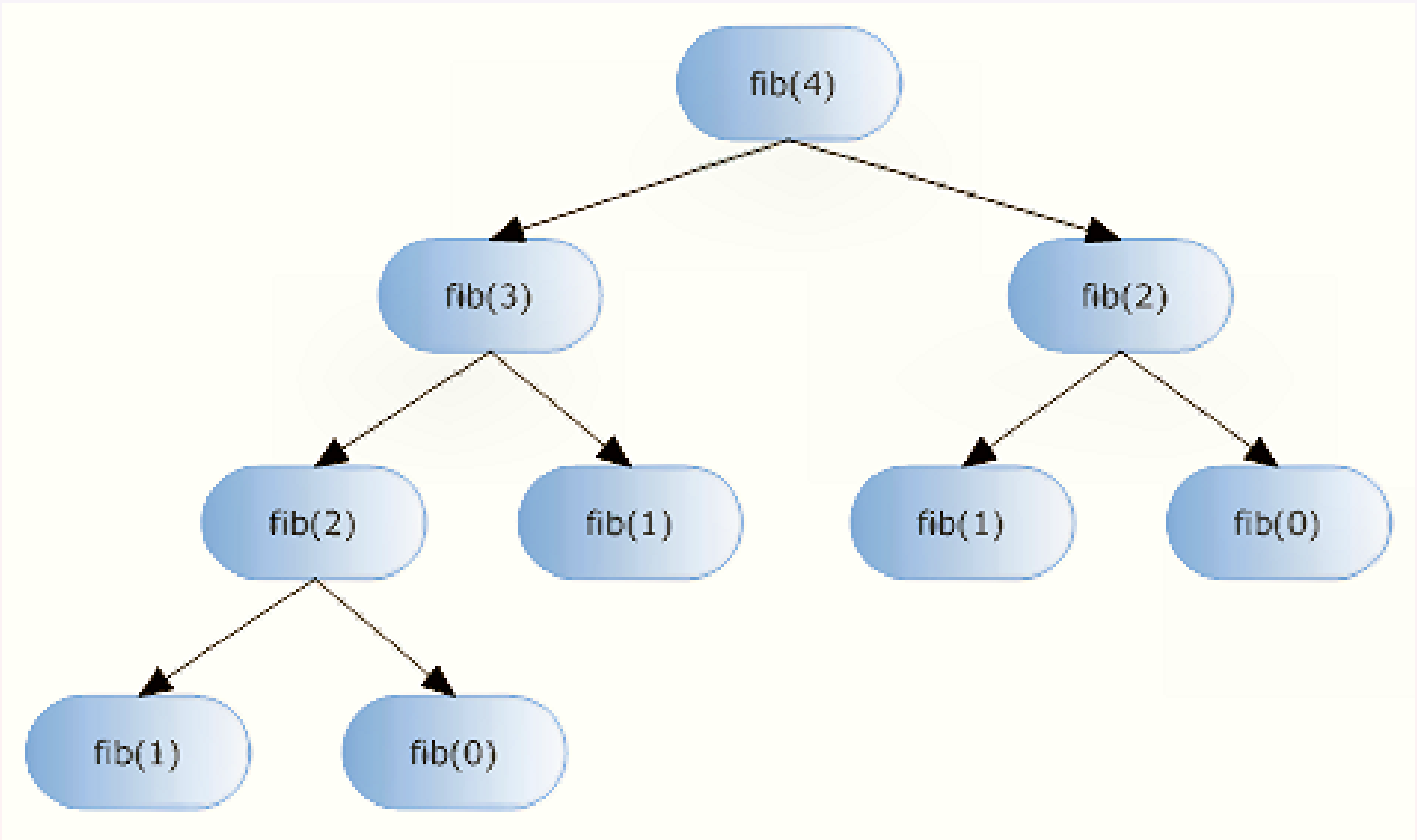
Pro $n = 4$: $fib(1)$ určován 3x, $fib(0)$ 2x.

Pro $n = 5$: $fib(1)$ určován 5x, $fib(0)$ 3x.

V praxi se řeší iterací.

13. Strom rekurzivního volání

Strom rekurzivního volání funkce *fib*(*n*).



14. Výpočet Fibonacciho čísel rekurzí

Zdrojový kód:

```
def fib(n):
    if (n>1):          #Podminka ukonceni rekurze
        return fib(n-1)+fib(n-2)
    else:
        return 1;      #Nerekurzivni doreseni
```

Je tento algoritmus efektivní?

15*. Výpočet doby běhu

Časová funkce $T(n)$ v závislosti na velikosti vstupu n:

$$
\begin{aligned}
T(0) &= T(1) = 1 \\
T(n) &= T(n-1) + T(n-2) + 3, \\
&< T(n-1) + T(n-1), \\
&= 2T(n-1), \\
&= 2[2T(n-2)] = 2^2 T(n-2), \\
&= 2\{2[2T(n-3)]\} = 2^3 T(n-3), \\
&= 2^i T(n-i), \\
&\ldots \\
&= 2^n T(n-n), \\
&= 2^n.
\end{aligned}
$$

Doba běhu exponenciální funkcí velikostí vstupu - neefektivní.
Horní odhad složitosti $O(2^n)$.

16*. Výpočet Fibonacciho čísel bez použití rekurze

Zdrojový kód:

```
def fib(n):
    fi = 1
    fii = 1
    i = 0
    while i < n:
        fi = fi + fii  #Vypocet predchoziho clenu
        fii = fi - fii #Vypocet nasledujiciho clenu
        i = i + 1
    return fi
```

Postup mnohem efektivnější než s použitím rekurze.

Doba běhu roste lineárně s velikostí n.

$$T(n) = kn, k > 0.$$

17. Špatná konstrukce rekurzivní podmínky

Špatná konstrukce rekurzivní podmínky $\Rightarrow$ nekonečná rekurze.

Chyba v podmínce nemusí být na první pohled viditelná.

Pro $n > 0$ je definována $f(n)$ jako

$$f(n) = \begin{cases} n \cdot f(n-2), & \text{pro } n \neq 1, \\ 1, & \text{pro } n = 1. \end{cases}$$

Pozor: pro sudé n dojde k nekonečné rekurzi:

```
f(5) = 5*f(3); f(3) = 3*f(1); f(1) = 1
f(4) = 4*f(2); f(2) = 2*f(0); f(0) = 0*f(-2); f(-2)=-2*f(-4);
...
```

Výpočet $f(n)$ s použitím rekurze:

```
def f(n):
  if n != 1:  #Nekonecna rekurze pro suda n
      return n * f(n - 2);
  else:
      return 1;
```

18. Nalezení maxima z množiny prvků

Dána neuspořádaná posloupnost prvků $X = \{x_i\}, i = 1, ..., n$.

Nalezněte hodnotu

$$\overline{x} = \max_{\forall x_i \in X}(x_i).$$

Varianty:

- Bez použití rekurze.
- S použitím rekurze.

Nalezení minima/maxima bez použití rekurze:

- Hodnotu $\overline{x}$ inicializujeme prvkem $\overline{x} = x[0]$.
- Porovnáváme $\overline{x}$ s ostatními členy posloupnosti.
- Pokud $\overline{x} > x[i]$, pak $\overline{x} = x[i]$.
- Po provedení porovnání se všemi prvky $\overline{x} = \max(x_i)$.

19. Hledání maxima “klasicky”

Zdrojový kód v Pythonu:

```
def maximum(x):
    max = x[0]                  #Inicializace maxima
    for xi in x:
        if xi > max:
            max = xi            #Aktualizace maxima
    return max;
```

20. Hledání maxima rekurzí

Hledání maxima $\overline{x}$ lze řešit rekurzí s dělením podmnožiny

$$\overline{x} = \max\{x_1, \ldots, x_n\}.$$

Přímým porovnáním umíme pouze pro $n = 1$, resp. $n = 2$:

$$\max\{x_1\} = x_1, \qquad \max\{x_1, x_2\} = \begin{cases} x_1, & x_1 \geq x_2, \\ x_2, & x_1 < x_2. \end{cases}$$

Úloha je dekomponovatelná na úlohy stejné třídy:

$$\begin{aligned} \overline{x} &= \max\{x_1, \ldots, x_n\}, \\ &= \max\{\max\{x_1, \ldots, x_{(1+n)/2}\}, \max\{x_{(3+n)/2}, \ldots, x_n\}\}, \\ &= \ldots, \\ &= \max\{\max\{\max\{x_1, x_2\}, \max\{x_3, x_4\}\}, \ldots, \max\{\max\{x_{n-3}, x_{n-2}\}, \max\{x_{n-1}, x_n\}\}\}. \end{aligned}$$

Maximum z levé a maximum z pravé poloviny, z obou částí maximum.
Následně zpětné dosazení do vztahů.

Výpočet středního indexu

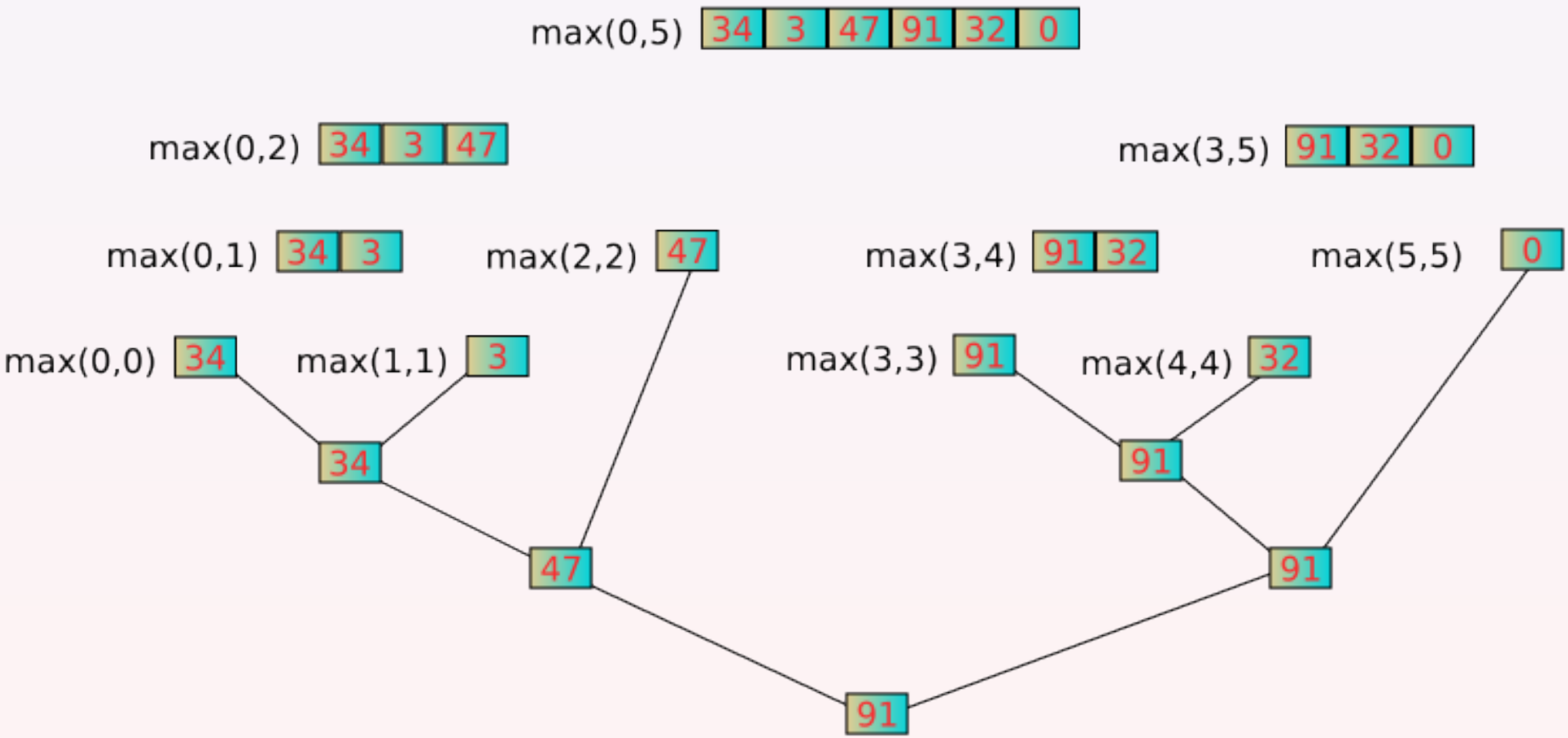
$$m = (l + r)//2.$$

Pak lze postup aplikovat rekurentně

$$\overline{x} = \max\{x_l, \ldots, x_r\} = \max\{\max\{x_l, \ldots, x_m\}, \max\{x_{m+1}, \ldots, x_r\}\}.$$

Složitost $O(n)$, nižší efektivita než u iterace.

21. Hledání maxima rekurzí, strom



22. Hledání maxima rekurzí

```
def maximum(x, l, r):
    if (r - l <= 1):                    # Maximum z 1-2 prvku umime
        return max(x[l], x[r])
    m = (l + r) // 2                    # Nalezeni prostredniho prvku
    lmax = maximum(x, l, m)             # Rekurze, L interval
    rmax = maximum(x, m + 1, r)         # Rekurze, R interval
    return max(lmax, rmax)
```

23*. Výpočet doby běhu

Časová funkce $T(n)$ v závislosti na velikosti vstupu n:

$$
\begin{aligned}
T(1) &= 1 \\
T(n) &= T(n/2) + T(n/2) + k, \\
&= 2T(n/2) + k, \\
&= 2[2T(n/4) + k] + k, \\
&= 4T(n/4) + 3k, \\
&= 2\{2[2T(n/8) + k] + k\} + k, \\
&= 8T(n/8) + 7k, \\
&= iT(n/i) + (i-1)k \\
&\dots \\
&= nT(1) + (n-1)k, \\
&\doteq nk + (n-1)k, \\
&= 2nk - k \\
&\doteq n
\end{aligned}
$$

24. Binární hledání

Předpokladem je setříděná vstupní množina $X = \{x_i\}$, $x_i \leq x_{i+1}$,$i = 1, \ldots, n-1$.

Dotaz, zda $a \in X$?

Problém $f(\{x_i\}, a)$

$$f(\{x_i\}, a) = \begin{cases} i, & x_i = a, \\ -1, & x_i \neq a. \end{cases}$$

Lze řešit přímo pro $n = 1$.

Úloha dekomponovatelná na úlohy stejné třídy:

$$\begin{aligned} f(\{x_1, \ldots, x_n\}, a) &= f(\{f(\{x_1, x_{(n-1)/2}\}, a), f(\{x_{(n+1)/2}\}, a), f(\{x_{(n+3)/2}, x_n\}, a)\}, a), \\ &= \ldots., \\ &= f(\{f(\{f(\{x_1\}, a), f(\{x_2\}, a)\}, a), \ldots, f(\{f(\{x_{n-1}\}, a), f(\{x_n\}, a)\}, a)\}, a). \end{aligned}$$

Leží v levé “polovině”, roven střednímu prvku, leží v pravé “polovině”?

Následně zpětné dosazení do vztahů.

Výpočet středního indexu

$$m = (l + r)/2.$$

Pak lze postup aplikovat rekurentně

$$f(\{x_l, \ldots, x_r\}, a) = f(\{f(\{x_l, \ldots, x_{m-1}\}, a), f(\{x_m\}, a), f(\{x_{m+1}, x_r\}, a)\}, a).$$

Efektivní algoritmus (pokud dělíme v mediánu), složitost $O(\log_2 n)$.

Nutnost předzpracování dat složitost $O(n \log_2 n)$.

25*. Výpočet doby běhu

Efektivní varianta:

Dělení ve středním prvku posloupnosti, sub-lineární

$$\begin{aligned} T(n) &= T(n/2) + k, \\ &= T(n/4) + 2k, \\ &= T(n/8) + 3k, \\ &= T(n/2^i) + ik, \\ &= T(1) + k \log_2 n, \\ &< \log_2 n. \end{aligned}$$

Neefektivní varianta:

Dělení v druhém/předposledním prvku posloupnosti, pouze lineární

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(n-1) + k, \\ &= T(n-2) + k + k, \\ &\ldots \\ &= T(n-n) + kn, \\ &= nk, \\ &< n. \end{aligned}$$

26*. Binární hledání, nerekurzivní řešení

```
def bSearch(a, x):
    l = 0
    r = len(x)- 1
    while l <= r:
        m = (l + r)//2              #Index prostredniho prvku
        if a == x[m]:
            return m                #Byl prvek nalezen?
        elif a > x[m]:
            l = m + 1               #Inkrementuj levy index
        else:
            r = m - 1               #Dekrementuj pravy index
    return -1;                      #a neni v x
```

27. Binární hledání, rekurzivní řešení

```
def bSearch(a, x, l, r):
    if (l > r):
       return -1                          #Prazdny interval
    m = (l + r)//2                        #Index prostredniho prvku
    if a == x[m]:
       return m                           #Cislo nalezeno v mnozine
    elif a < x[m]:
       return bSearch(a, x, l, m - 1)     #Leva podmnozina
    else:
       return bSearch(a, x, m + 1, r )    #Prava podmnozina
```

28. Problém Hanojských věží

Mniši v Tibetu řeší zajímavý hlavolam tvořený třemi tyčemi.
Na první tyči navlečeno 64 kroužků seřazených dle velikosti.

Každý den přesunou jeden kroužek z jedné tyče na druhou.
Větší kroužek nesmí být položen na menší.

Až přemístí všechny kroužky z první tyče na třetí, zanikne svět.

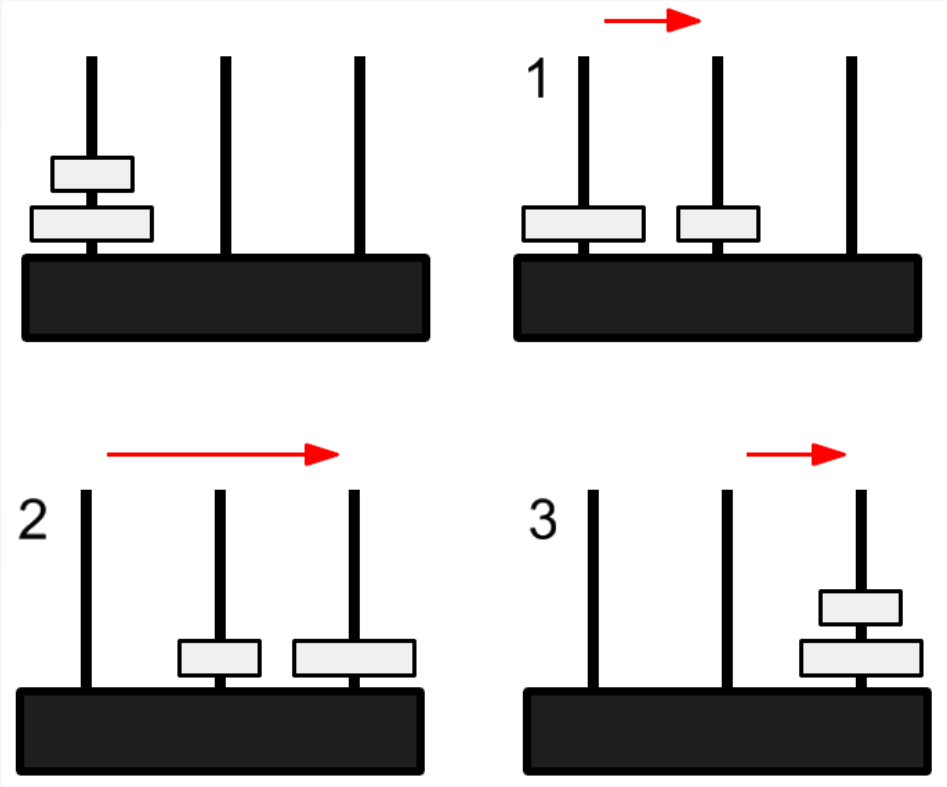
Nastane za

$$2^{64} - 1 = 18446744073709551615 \text{ dní},$$
$$\approx 505044327822301 20 \text{ let (51 biliard)}.$$

Plánovaná doba existence sluneční soustavy je miliardkrát kratší.
Netřeba mít strach :-).

Problém lze řešit rekurzí.
Počet kroužků n.

29. Ukázka řešení pro $n = 2$



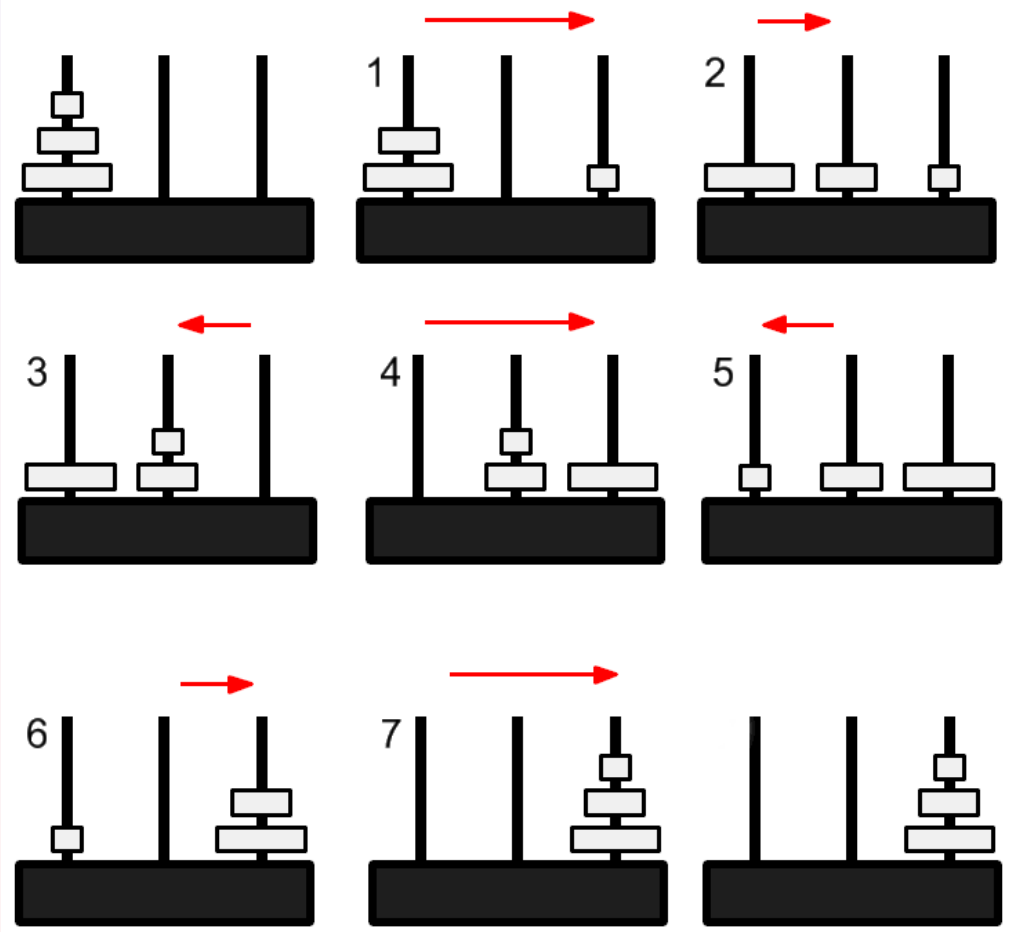
První kroužek přesuneme z 1. na 2. tyč.

Druhý kroužek přesuneme z 1. na 3. tyč.

První kroužek přesuneme z 2. na 3. tyč.

Celkem $2^2 - 1$ přesunů.

30. Ukázka řešení pro $n = 3$



Celkem $2^3 - 1$ přesunů.

31. Rozbor problému

Tah h s n kroužky z i na j přes k (pomocná)

$$h_n(i, j, k).$$

Dva kroužky, $n = 2$:

$$h_2(1, 3, 2) = \{h_1(1, 2, 3), h_1(1, 3, 2), h_1(2, 3, 1)\}.$$

Tři kroužky, $n = 3$:

$$\begin{aligned} h_3(1, 3, 2) &= \{h_1(1, 3, 2), h_1(1, 2, 3), h_1(3, 2, 1), h_1(1, 3, 2), h_1(2, 1, 3), \\ &\quad h_1(2, 3, 1), h_1(1, 3, 2)\}, \\ &= \{h_2(1, 2, 3), h_1(1, 3, 2), h_2(2, 3, 1)\}. \end{aligned}$$

Zobecnění

$$h_n(1, 3, 2) = \{h_{n-1}(1, 2, 3), h_1(1, 3, 2), h_{n-1}(2, 3, 1)\}.$$

32. Řešení rekurze pro *n* prvků

Základní myšlenka rekurzivního přístupu pro *n* prvků tvořena 3 kroky:

- přemístíme $n-1$ kroužků z 1. (zdrojové) na 2 .(pomocnou) -> rekurze.
- přemístíme 1 kroužek z 1. (zdrojové) na 3. (cílovou).
- přemístíme $n-1$ kroužků z 2. (pomocné) na 3. (cílovou) -> rekurze.

Kroky 1) a 3):

1. Řeší stejnou úlohu s jinými tyčemi a s menším množstvím prvků.
2. Lze je rekurzivně rozložit dle stejného schématu.

Úloha je dekomponovatelná na úlohy stejného typu.
V každém kroku se velikost množiny zmenšuje, lze využít rekurzi.
Avšak velmi neefektivní.

33. Zápis zdrojového kódu

```
def hanoi(n, o, k, p):
    if n==0:
        return                  #Ukonceni rekurze
    hanoi(n-1, o, p, k);  #Presun 1-2
    print("Z " + o +" na " + k);
    hanoi(n-1, p, k, o);  #Presun 2-3
```

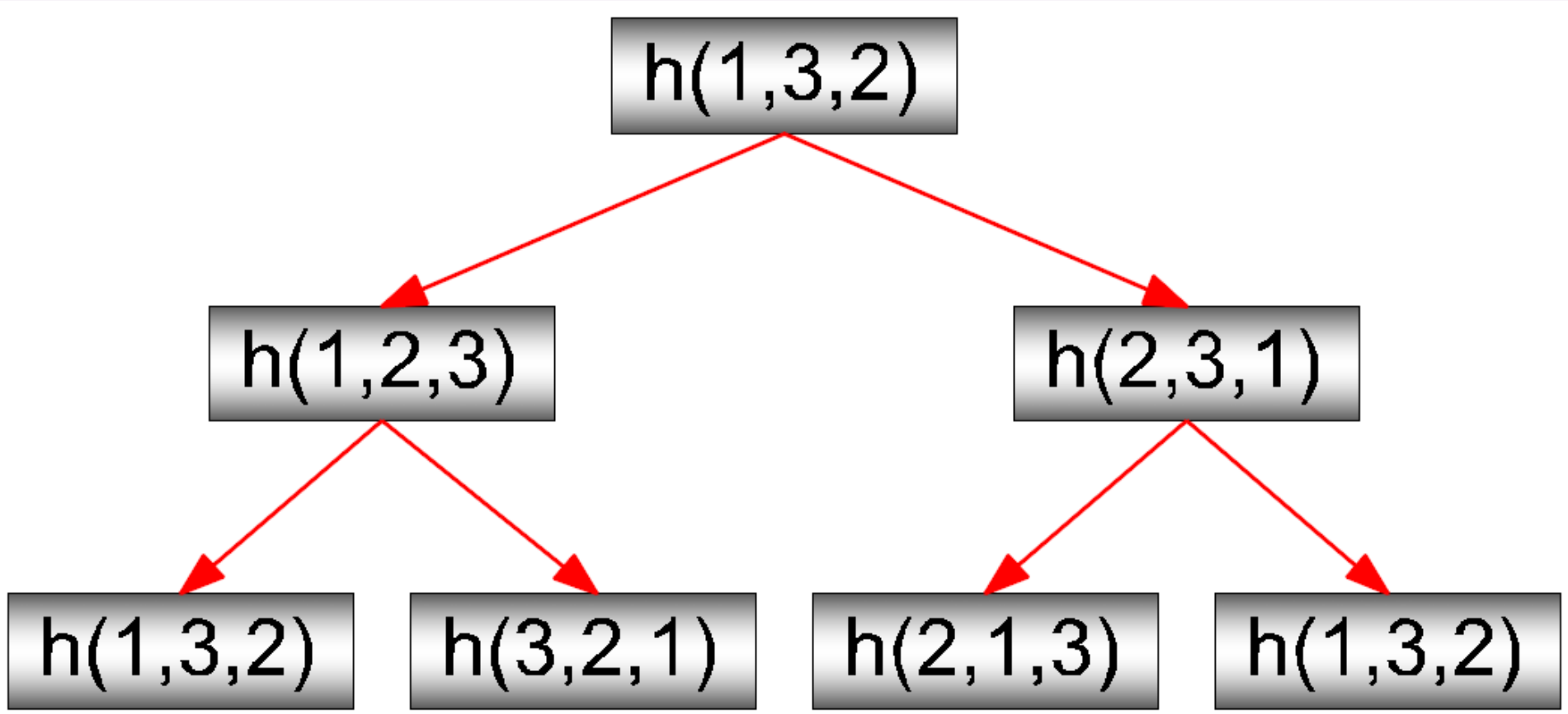
Volání:

```
hanoi(3,1,3,2);
```

Počet rekurzivních volání: $2^n - 1$.

Velmi neefektivní, max 20 kroužků.

34. Znázornění stromu rekurzivních volání



35*. Analýza doby běhu

.

Dvojice rekurizivních volání:

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-1) + 1 \\
&= 2T(n-1) + 1, \\
&= 2[2T(n-2) + 1] + 1, \\
&= 4T(n-2) + 3, \\
&= 2\{2[2T(n-3) + 1] + 1\} + 1, \\
&= 2^i T(n-i) + 2^0 + 2^1 + \ldots + 2^{i-1}, \\
&= \ldots \\
&= 2^{n-1} T(1) + 2^0 + 2^1 + \ldots + 2^{n-2}, \\
&= \frac{2^n - 1}{1} = 2^n - 1, \\
&\doteq 2^n.
\end{aligned}
$$

Neefektivní, počet operací roste exponenciálně.

36*. Kochova vločka

Autorem Niels Fabian Helge von Koch.

Patří mezi první objevené fraktální křivky.

Geometrický útvar vzniklý opakovaným rekurzivním dělením rovnostranného trojúhelníku.

Princip konstrukce:

- Každá ze stran trojúhelníku rozdělena na třetiny.
- Nad prostřední stranou vytvořen nový rovnostranný trojúhelník, jehož vrchol leží vně původního trojúhelníku.
- Z nového rovnostranného trojúhelníku odstraněna základna a pokračuje se bodem 1.

Kochova vločka vznikne opakováním tohoto postupu n krát.

Délka d Kochovy vločky se s každým krokem prodlouží o $\frac{1}{3}$, celková délka Kochovy vločky

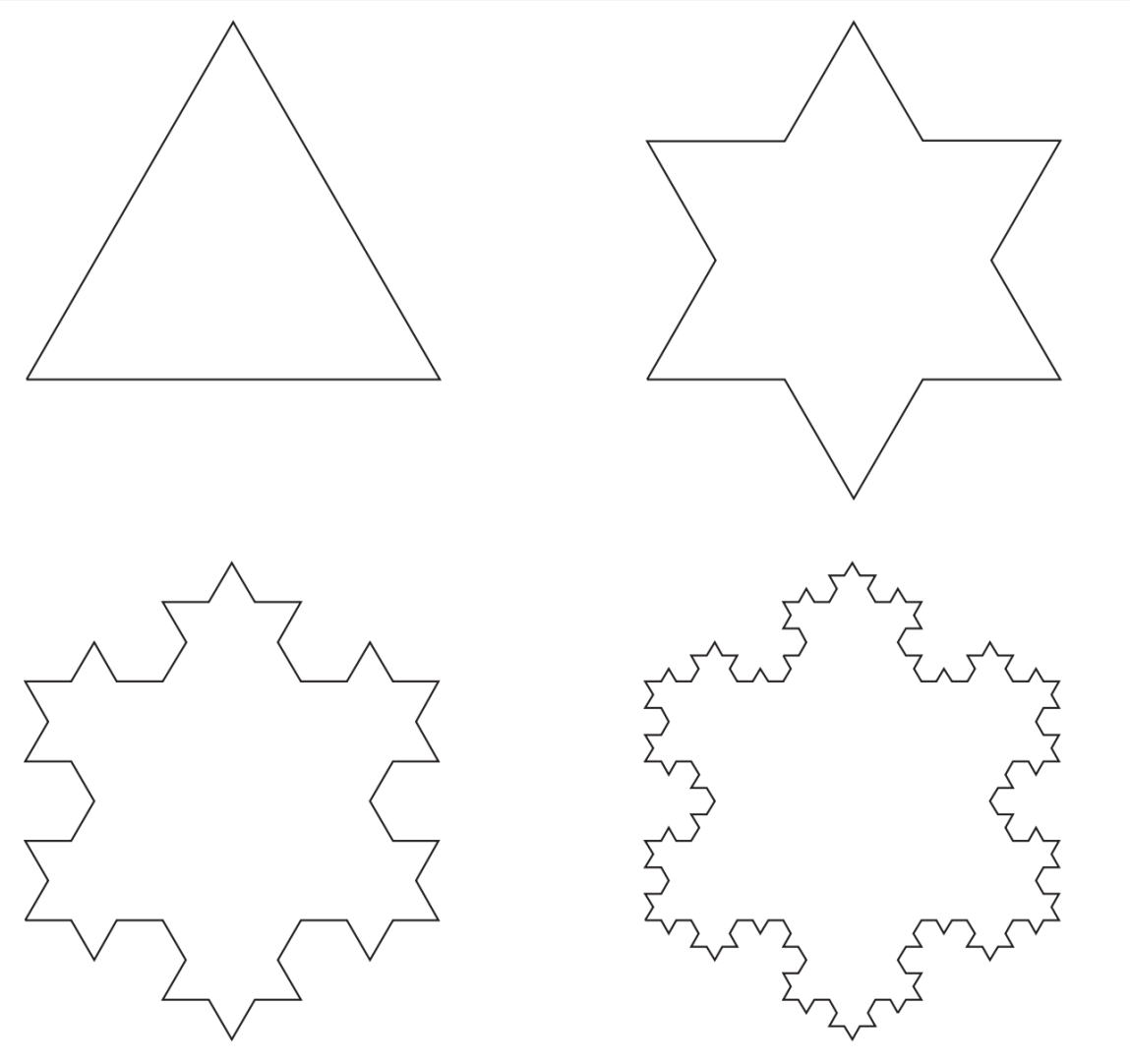
$$d = (\frac{4}{3})^n.$$

Pro n dělení platí

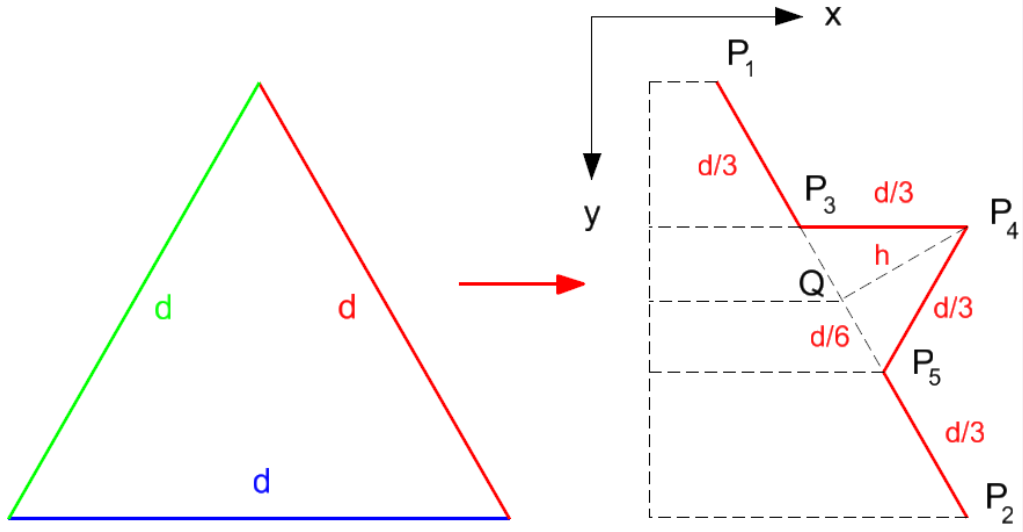
$$\lim_{n \to \infty} (d) = \infty.$$

37*. Kochova vločka, ukázka

Kochova vločka pro $n = 1$, $n = 2$, $n = 3$, $n = 4$.



38*. Postup konstrukce Kochovy vločky



Rekurzivní dělení prováděno nad každou ze tří stran trojúhelníku.
Známy souřadnice koncových bodů $P_1 = [x_1, y_1]$, $P_2 = [x_2, y_2]$ strany trojúhelníku.
Určujeme souřadnice bodů P_3, P_4, P_5 tvořící koncové body nových segmentů.

V každém kroku vznikají *čtyři* nové segmenty, výsledkem strom čtvrtého stupně.

Směrový vektor $\overrightarrow{u}$, normálový vektor $\overrightarrow{n}$, $\overrightarrow{n} \perp \overrightarrow{u}$.

$$\overrightarrow{u} = \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix} \quad \overrightarrow{n} = \begin{pmatrix} y_2 - y_1 \\ x_1 - x_2 \end{pmatrix}$$

39*. Výpočet souřadnic vrcholů

Souřadnice bodů P_1, P_2, P_3

$$\begin{pmatrix} x_3 \\ y_3 \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + \frac{1}{3}\begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix}, \quad \begin{pmatrix} x_5 \\ y_5 \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + \frac{2}{3}\begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix},$$
$$\begin{pmatrix} x_4 \\ y_4 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} x_1 + x_2 \\ y_1 + y_2 \end{pmatrix} + \alpha\begin{pmatrix} y_2 - y_1 \\ x_1 - x_2 \end{pmatrix}.$$

Výška h v $\triangle(P_4, P_5, Q)$

$$h = \sqrt{\frac{d^2}{9} - \frac{d^2}{36}} = \frac{d}{2\sqrt{3}}.$$

Parametr α

$$\alpha = \frac{h}{\|\overrightarrow{n}\|} = \frac{d}{2\sqrt{3}d} = \frac{1}{2\sqrt{3}}.$$

Bod P_4

$$\begin{pmatrix} x_4 \\ y_4 \end{pmatrix} = \frac{1}{2}\left[\begin{pmatrix} x_1 + x_2 \\ y_1 + y_2 \end{pmatrix} + \frac{1}{\sqrt{3}}\begin{pmatrix} y_2 - y_1 \\ x_1 - x_2 \end{pmatrix}\right].$$

40*. Zápis zdrojového kódu

```
def koch(x1, y1, x2, y2, n, draw):
    if n>1:  #Podminka ukonceni rekurze
        #Strana 1
        koch(x1, y1, x1 + (x2 - x1) / 3, y1 + (y2 - y1) / 3, n - 1, draw)
        #Strana2
        koch(x1 + (x2 - x1) / 3, y1 + (y2 - y1) / 3, (x1 + x2) / 2 + (y2 - y1) / (2 * sqrt(3)), \
            (y1 + y2) / 2 + (x1 - x2) / (2 * sqrt(3)), n - 1, draw)
        #Strana 3
        koch((x1 + x2) / 2 + (y2 - y1) / (2 * sqrt(3)), (y1 + y2) / 2 + (x1 - x2) / (2 * sqrt(3)), \
              x1 + (x2 - x1) * 2 / 3, y1 + (y2 - y1) * 2 / 3,  n - 1, draw);
        #Strana 4
        koch(x1 + 2 *(x2 - x1) / 3, y1 + 2 *(y2 - y1) / 3, x2, y2, n - 1, draw)
    else: #Vykresli segment
        draw.line((x1, y1, x2, y2), 'black')
```

Volání funkce:

```
m = Image.new('RGB', (800, 600), color='white')
draw = ImageDraw.Draw(im)
d = 600
h = 2
koch(100, 100, 100 + d, 100, h, draw);
koch(100 + d, 100, 100 + 0.5 * d, 100 + d * sqrt(3) / 2, h, draw);
koch(100 + 0.5 * d, 100 + d * sqrt(3) / 2, 100, 100, h, draw);
im.show()
```

41. Vhodnost/nevhodnost rekurze

- Rekurzi lze použít, pokud počet rekurzivních volání roste lineárně
  Funkce by měla obsahovat pouze jedno rekurzivní volání sama sebe.
- Pro každé rekurzivní volání vytvoření nové sady lokálních proměnných, jejich uložení do zásobníku a následné vyjmutí (značná hw. režie).
- Při práci s rozsáhlými daty rekurzivní algoritmy neefektivní
  Značné nároky na paměť.
- Pro rozsáhlé datové soubory může “dojít paměť” dříve, než nalezeno řešení.
- Nevhodná změna hodnoty proměnné ovlivňující ukončení rekurze způsobí nekonečnou rekurzi.
- Stručný, ale nepříliš přehledný kód
  Někdy není jasné, co algoritmus dělá.
- Používá se u řešení problémů, kde není známo nerekurzivní řešení.