Rekurze.

Princip a použití rekurze. Základní problémy řešitelné rekurzí. Odhady složitostí. Převod rekurze na iteraci.

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
   - Nebezpečí rekurze
2. Příklady použití rekurze
   - Hledání maxima rekurzí
   - Binární hledání
   - Úloha Hanojských věží
   - Kartografická generalizace
   - Jednoduchá kresba s využitím rekurze
   - Konstrukce Kochovy vločky rekurzí
   - Vhodnost/nevhodnost rekurze
3. Převod rekurze na iteraci

1. Rekurze

Sebeopakování, realizováno bez použití cyklu.

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

Rekurzivní funkce f:

Volají samy sebe nebo podobné funkce.

Po každém kroku “zjednodušení” problému, jinak postup neefektivní.

Nalezení vztahu mezi problémy řešenými ve 2 následujících rekurzivních krocích.

Vhodné pro *rekurentní funkce*, generují posloupnost $\{a_n\}$

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

Při každém volání f dochází k vytvoření nové sady *lokálních proměnných*.

Za běhu existuje několik sad lokálních proměnných.

Hloubka rekurze h:

Počet rekurzivních volání funkce f.

2. Vlastnosti rekurze

Podmínka ukončení rekurze:

Rekurzivní algoritmus musí obsahovat ukončovací podmínku.

V opačném případě dojde k nekonečné rekurzi !!!.

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

Nevhodná konstrukce podmínky:

Nepoužívat složité a nepřehledné podmínky.

Vyhnout se kontradikcím a tautologiím (nemění se pravdivostní hodnota).

Dělení rekurze:

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

Nepřímá (kruhová) rekurze: cyklické volání několik funkcí:

$$f_1 \rightarrow f_2 \rightarrow ... \rightarrow f_n \rightarrow f_1.$$

3. Vztah rekurze a iterace

Iterace:

Opakování bloku příkazů za stanovených podmínek.

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

Podmínka opakován blok následuje (`do while`).

Rekurze:

Opakování bloku, podmínka součástí bloku.

Nejčastěji používána u funkcí.

Mezi rekurzí a iterací úzký vztah.

Vzájemná převoditelnost, u vícenásobné rekurze netriviální.

Rekurze mnohdy přímočařejší než iterace.

Avšak výpočetně náročnější.

Pokud je to možné, rekurzi se vyhýbáme.

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:

$f(n) = \{C_n;\ p(n)\ ?\ D\ :\ f(n - 1); E_n\ \}$

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 -> Cn    f(n - 1)
fn -> Cn    f(n - 1)  Cn-1  f(n - 2)
fn -> Cn    f(n - 1)  Cn-1  f(n - 2)  Cn-2  f(n - 3)
...
fn -> Cn    f(n - 1)  Cn-1  f(n - 2)  ...   C2  f(1)  C1 #Konec rekurze
#Nerekurzivni doreseni a zpetny chod
fn -> Cn    f(n - 1)  Cn-1  f(n - 2)  ...   C2  f(1)  C1  D  E1
fn -> Cn    f(n - 1)  Cn-1  f(n - 2)  ...   C2  f(1)  C1  D  E1  E2
...
fn -> Cn    f(n - 1)  Cn-1  f(n - 2)  ...   C2  f(1)  C1  D  E1  E2 ... En
```

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)

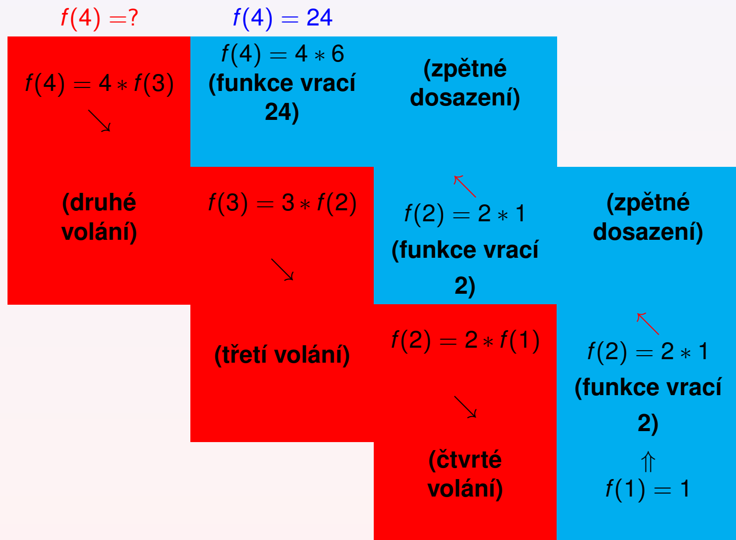
$$f(n) = \begin{cases} nf(n-1), & \text{pro } n > 1, \\ 1, & \text{pro } n = 1. \end{cases}$$

Rekurzivní volání f nad *zmenšující se* množinou prvků.

Zjednodušení problému (předpoklad efektivního použití rekurze).

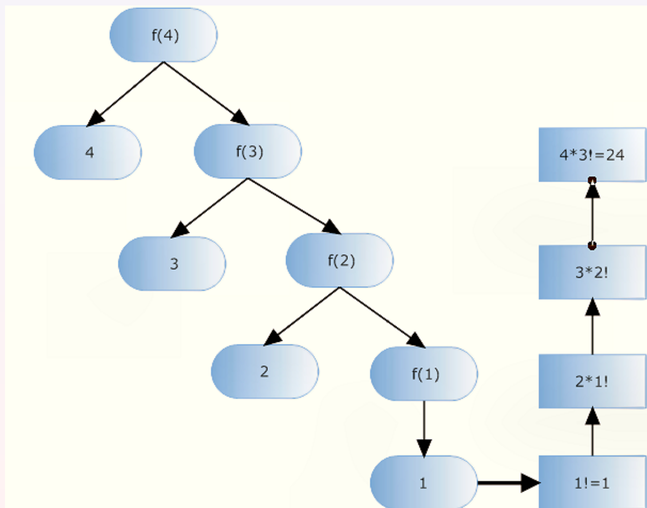
Splněny předpoklady.

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:

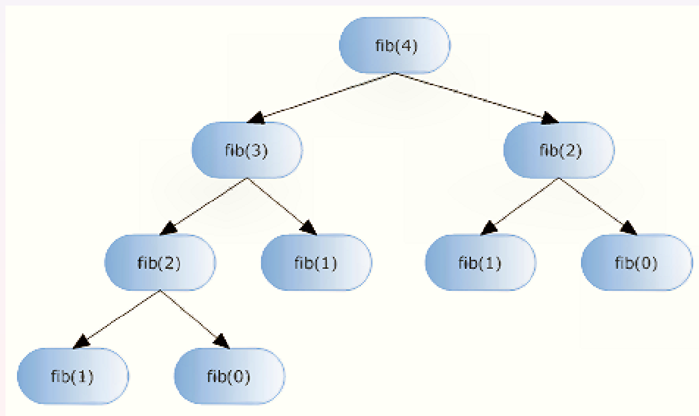
Pro $n = 4$: $fib(1)$ určován 3x, $fib(0)$ 2x.

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:

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

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. Růst množiny

Další krok nevede ke zjednodušení předchozího.

Důsledkem růst množiny, nad níž výpočet prováděn $\Rightarrow$ *divergence řešení*.

Vznik nekonečné rekurze.

Pro $n > 0$ je definována $\alpha(n)$ jako

$$\alpha(n) = \begin{cases} n \cdot \alpha(n+1), & \text{pro } n > 1, \\ 1, & \text{pro } n = 1. \end{cases}$$

Výpočet $\alpha(n)$ s použitím rekurze:

```
def alfa(n):
  if n > 1:                      #Ukoncovaci podminka
      return n * alfa(n + 1)     #Mnozina roste
  else:
      return 1                   #Nikdy nedojde k ukonceni
```

18. Špatná konstrukce rekurzivní podmínky

Špatná konstrukce rekurzivní podmínky $\Rightarrow$ nekonečná rekurze

Pro $n > 0$ je definována $\beta(n)$ jako

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

Pro sudé n dojde k nekonečné rekurzi.

Výpočet $\beta(n)$ s použitím rekurze.

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

19. Další ukázky použití rekurze

- Nalezení maxima z množiny.
- Binární hledání.
- Hledání největšího společného dělitele.
- Hanojské věže.
- Kresba vepsaných čtverců, spirály.
- Fraktály: Kochova vločka.
- Digitální kartografie: generalizace Douglas & Peucker.

20. 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 $\overline{x} = \max(x_i)$.

21. 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;
```

22. Hledání maxima rekurzí

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

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

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

$$\max\{x_i\} = x_i, \qquad \max\{x_i, x_{i+1}\} = \begin{cases} x_i, & x_i \geq x_{i+1}, \\ x_{i+1}, & x_i < x_{i+1}. \end{cases}$$

Úloha je dekomponovatelná na úlohy stejné třídy:

$$\begin{aligned} \overline{x} &= \max\{x_1, \dots, x_n\}, \\ &= \max\{\max\{x_1, \dots, x_{(1+n)/2}\}, \max\{x_{(1+n)/2}, \dots, x_5\}\}, \\ &= \dots, \\ &= \max\{\max\{\max\{x_1, x_2\}, \max\{x_2, x_3\}\}, \dots, \max\{\max\{x_{n-2}, x_{n-1}\}, \max\{x_{n-1}, x_n\}\}\}. \end{aligned}$$

Následně zpětné dosazení do vztahů.

Výpočet středního indexu

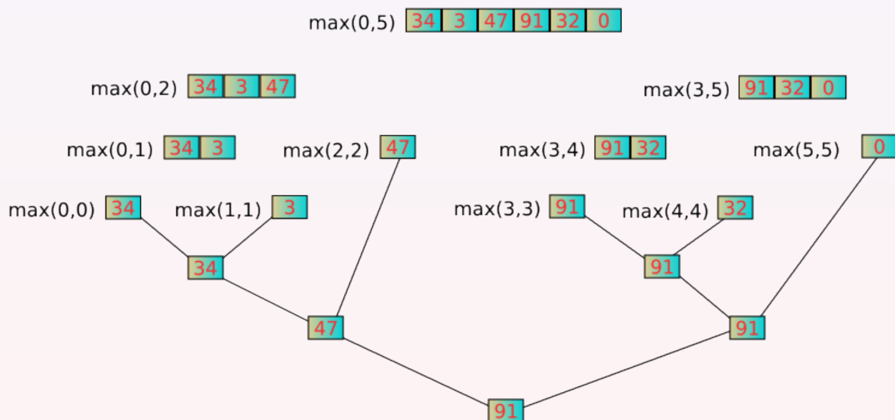
$$m = (l + r)//2.$$

Pak lze postup aplikovat rekurentně

$$\overline{x} = \max\{x_l, \dots, x_r\} = \max\{\max\{x_l, \dots, x_m\}, \max\{x_m, \dots, x_r\}\}.$$

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

23. Hledání maxima rekurzí, strom



24. 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)
```

25. Výpočet doby běhu

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

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

26. Prvočíslo (1/2)

Číslo $d \in \mathbb{N}$ dělí $c \in \mathbb{N}$, pokud existuje $k \in \mathbb{N}$ takové, že $c = kd$. Označení $d|c$.

Vlastnosti:

1) Pro všechna $c \in \mathbb{N}$ platí $1|c$ a $c|c$.
2) Pokud $k|c$ a $c|d$ pak $k|d$.

Číslo $c \in \mathbb{N}$ prvočíslem, pokud pro každá $a \in \mathbb{N}$ a $b \in \mathbb{N}$ platí

$$c|(a \cdot b) \Rightarrow c|a \vee c|b.$$

Alternativně

$$c : a|c \wedge b|c \Leftrightarrow a = c \wedge b = 1 \vee a = 1 \wedge b = c.$$

Číslo $c \in \mathbb{N}$ je složené, pokud není prvočíslo.

3) Pro $\forall a \in \mathbb{Z}$, $a > 0$, $\exists c$, $a < c \leq 2a$.
4) MFV: Pro $\forall a \in \mathbb{Z}$, $\exists c$, $0 < a < c$,

$$c|(a^c - a).$$

27. Prvočíslo (2/2)

Počet prvočísel $\pi(N)$ na intervalu $[2, N]$

Hustota prvočísel

$$\rho(N) = \frac{\pi(N)}{N} \approx \frac{N}{\ln N}.$$

$\pi(10) = 4$, $\pi(100) = 25$.

Složené číslo $c = a \cdot b$

$$c : a|c \wedge b|c \Leftrightarrow a \geq 2 \wedge b \leq c/2 \vee a \leq c/2 \wedge b \geq 2.$$

Pokud $a \leq b$

$$\overline{a} = \sqrt{c}.$$

Díky komutativitě symetrie: $c = ab = ba$.

Příklad:

36 = 2 * 13 = 4 * 9 = 6 * 6 = 9 *4 = 13 *2.

28. Největší společný dělitel (1/2)

Největší společný dělitel d, $d \in \mathbb{Z}$, čísel $a \in \mathbb{Z}$, $b \in \mathbb{Z}$

$$d = gcd(a, b) : d|a \wedge d|b.$$

Pokud a, b nesoudělná, pak $gcd(a, b) = 1$.
Dělení se zbytkem a beze zbytku

$$b|a = r \Leftrightarrow a = kb + r, \qquad b|a = 0 \Leftrightarrow a = kb.$$

Nechť $a \in \mathbb{Z}$, $b \in \mathbb{Z}$, pak

$$a = kd, \qquad b = ld.$$

Rozdíl Δ čísel je také dělitelný d

$$\Delta = a - b = d(k - l), \qquad a = b + \Delta,$$

platí

$$gcd(a, b) = gcd(b, \Delta).$$

29. Největší společný dělitel (2/2)

Výše uvedený princip použit u Eukleidova algoritmu

$$gcd(a-b, b) = gcd(a, b).$$

Předpoklad $a > b$, opakujeme, dokud $a > 0$.
Po k krocích

$$d = gcd(a, b) = gcd(0, d).$$

Výhodou je velmi jednoduchá implementace.

```
do
    if a < b
       swap a ↔ b
    a = a − b
while a > 0
d = b
```

30. Vylepšení algoritmu (1/2)

Pro velké $a, b \to \infty$ cyklus běží dlouho.

Vylepšení, místo

$$a = a - b,$$

použijeme

$$a = a - kb, \qquad k \in \mathbb{Z}.$$

Platí

$$a - kb = a\%b.$$

Upravený algoritmus:

```
do
    if a < b
        swap a ↔ b
    a = a%b
while a > 0
d = b
```

31. Vylepšení algoritmu (2/2)

Nemusíme prohazovat $a \leftrightarrow b$.

Předpoklad

$$b \geq a\%b.$$

Do a, b přiřadíme

$$a = \max(b, a\%b) = b,$$
$$b = \min(b, a\%b) = a\%b.$$

Upravený algoritmus:

```
do
    a = b
    b = a%b
while b > 0
```

Modifikace testovací podmínky: $a \leftrightarrow b$.

32. Rekurzivní řešení

```
def gcd (a, b):
    if (a==0 | b== 0 ):
        return -1
    if a%b > 0:
        return gcd (b, a % b);
    else:
        return b;
```

33. Výpočet doby běhu

Faktor zmenšení velikosti množiny

$$d = \frac{a}{a\%b}.$$

$$\begin{aligned} T(n) &= T(n/d) + k, \\ &= T(n/d^2) + 2k, \\ &= T(n/d^3) + 3k, \\ &= \ldots \\ &= T(n/d^i) + ik, \end{aligned}$$

Pokud $n/d^i = 1$, pak $n = d^i$, $i = \log_d n$

$$\begin{aligned} T(n) &= T(1) + k \log_d n, \\ &= 1 + k \log_d n, \\ &< \log_d n. \end{aligned}$$

34. Binární hledání

Předpokladem je uspořádaná vstupní množina $X = \{x_i\}, x_i \leq x_{i+1}, i = 1, ..., n-1$.
Dotaz, zda $a \in X$?

Úlohy $f(\{x_i\}, a)$ lze řešit přímo pro $n = 1$

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

Úloha je dekomponovatelná na úlohy stejné třídy:

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

Následně zpětné dosazení do vztahů.

Výpočet středního indexu

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

Pak lze postup aplikovat rekurentně

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

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

35. 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

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

36. 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?
         if a > x[m]:
             l = m + 1               #Inkrementuj levy index
         else:
             r = m - 1               #Dekrementuj pravy index
    return -1;                       #a neni v x
```

37. 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
    if a < x[m]:
       return bSearch(a, x, l, m - 1)       #Leva podmnozina
    else:
       return bSearch(a, x, m + 1, r )      #Prava podmnozina
```

38. 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.

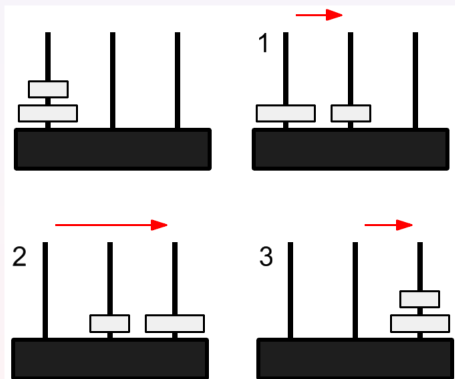
Nastane za

$$\begin{aligned} 2^{64} - 1 &= 18446744073709551615 \text{ dní}, \\ &\approx 50504432782230120 \text{ let (51 biliard)}. \end{aligned}$$

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

Problém lze řešit rekurzí.
Počet kroužků n.

39. Ukázka řešení pro $n = 2$



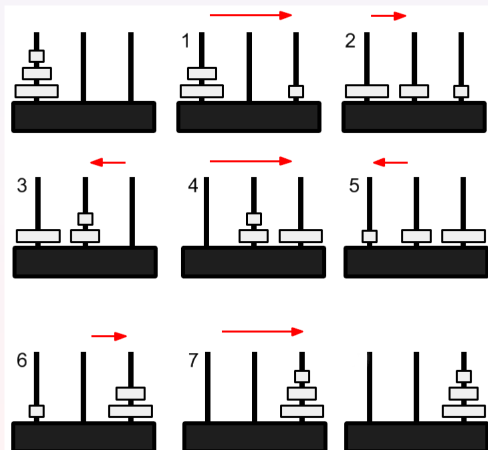
První kroužek přesuneme z 1. na 2. tyč.

Druhý kroužek přesuneme z 1. na 3. tyč.

První kroužek přesuneme z 2. na 3. tyč.

Celkem $2^2 - 1$ přesunů.

40. Ukázka řešení pro $n = 3$



Celkem $2^3 - 1$ přesunů.

41. 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

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

42. 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.

43. 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
```

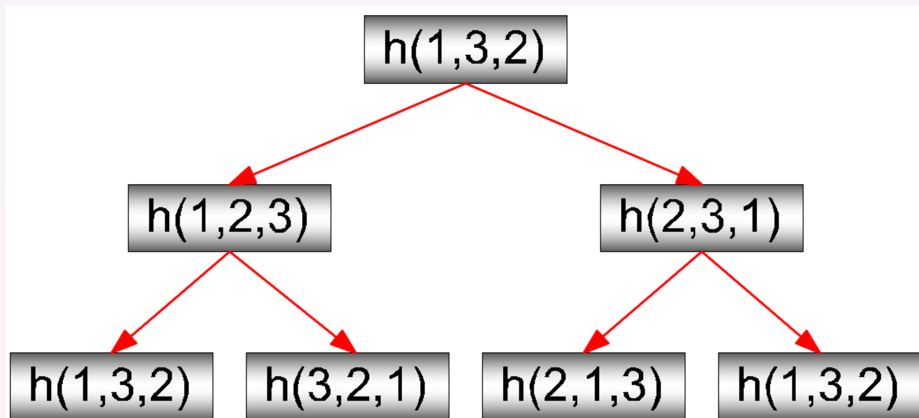
Volání:

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

Počet rekurzivních volání: $2^n - 1$.

Velmi neefektivní, max 20 kroužků.

46. Znázornění stromu rekurzivních volání



47. Analýza doby běhu

.

Dvojice rekurizivních volání:

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

Neefektivní, počet operací roste exponenciálně.

48. Douglas-Peucker algoritmus

Používá se pro generalizaci lomených čar.

Neodstraňuje z lomené čáry vrcholy nesplňující geometrickou podmínku.

Postupně přidává vrcholy splňující geometrickou podmínku.

Provádí opakované dělení vstupní množiny na menší podmnožiny.

V každém kroku dochází ke zjednodušení problému.

Lze implementovat rekurzí.

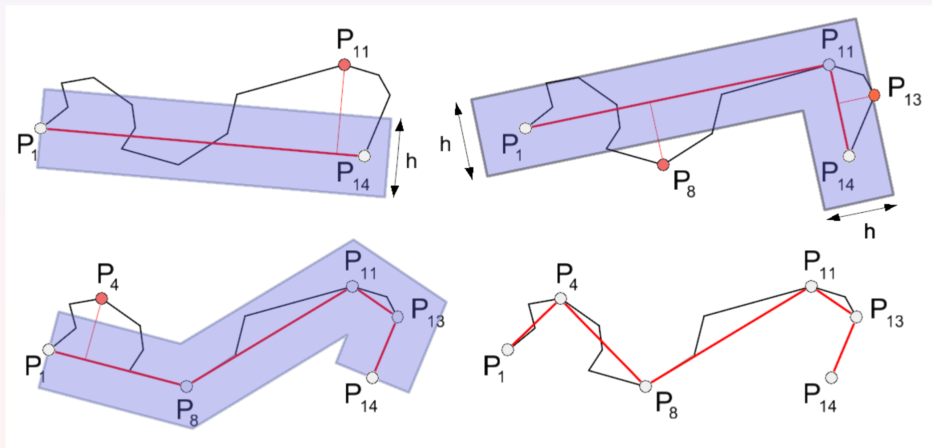
Lomená čára L, generalizovaná lomená čára LS, koridor o šířce h.

Startovní bod s, koncový bod e

1. Přidej s do LS
2. Nalezení bodu q nad segmentem (s, e)
   Nejvzdálenější od segmentu (s, e), vně koridoru.
3. Pro segment (s, q) jdi na 2)
4. Přidání q do LS.
5. Pro segment (q, e) jdi na 2).

Body 2-5 představují rekurzivní proceduru.

49. Ukázka Douglas-Peuckerova algoritmu



50. Implementace Douglas-Peuckerova algoritmu

```
def dp(L, LS, start, end, h):
    if (end == start + 1):
        return;                                  # Zadny mezilehly bod

    i_max = start + 1;                           # Inicializace
    d_max = distance(L[start], L[end], L[start + 1])
    for i in range(start + 2, end):              # Hledani nejvzdalenejsiho bodu
        di = distance(L[start], L[end], L[start + 1]);
        if (di > d_max):                         #Aktualizuj minimum
            i_max = i;
            d_max = di;
    if (d_max >= corridor):                      #Nalezen bod vne koridoru

        dp(L, LS, start, index_max, h)           #Rekurze, 1. segment
        LS.append(L[index_max])                  #Pridani bodu do LS
        dp(L, LS, index_max, end, h)             # Rekurze, 2. segment
```

51. Doba běhu Douglas-Peuckerova algoritmu

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

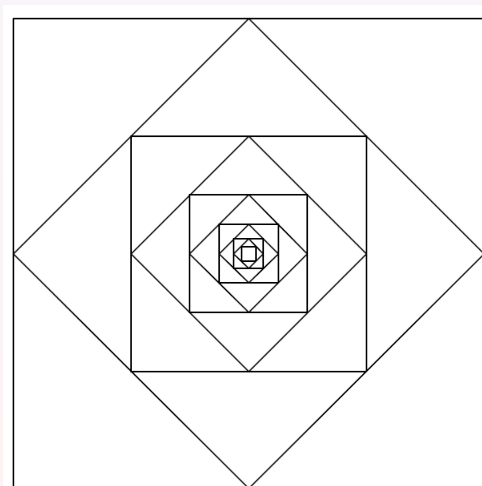
$$
\begin{aligned}
T(1) &= 1 \\
T(n) &= T(n/2) + T(n/2) + n, \\
&= 2T(n/2) + n, \\
&= 2[2T(n/4) + n] + n, \\
&= 4T(n/4) + 3n, \\
&= 2\{2[2T(n/8) + k] + k\} + k, \\
&= 8T(n/8) + 7n, \\
&= iT(n/i) + (i-1)n \\
&\ldots \\
&= nT(1) + (n-1)n, \\
&= n + (n-1)n, \\
&< n^2.
\end{aligned}
$$

Doba běhu kvadratickou funkcí velikostí vstupu!

52. Kresba vepsaných čtverců rekurzí

Nakreslete posloupnost vepsaných čtverců s použitím rekurze.

Délku počáteční hrany označme d.



53. Rozbor problému

Lze problém řešit prostřednictvím rekurze?

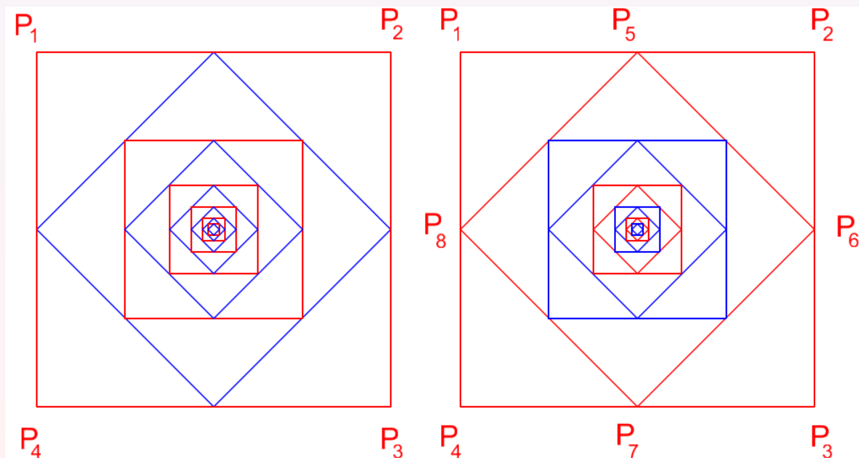
- Úloha je dekomponovatelná na úlohy stejné třídy: kresba čtverců daných vrcholy.
- V dalším kroku rekurze nedochází k zesložitění problému.
- Lze snadno stanovit podmínku ukončení rekurze: velikost strany menší než ε.

Existuje několik variant řešení problému:

- Konstrukce po čtvercích: v jednom kroku zkonstruován jeden čtverec.
- Konstrukce po dvojicích čtverců: v jednom kroku zkonstruována dvojice čtverců.

54. Znázornění konstrukce

Vlevo konstrukce po čtvercích, vpravo konstrukce po dvojicích čtverců. Čtverce konstruované v jednom kroku znázorněny stejnou barvou.



55. Konstrukce po čtvercích

Vrcholy čtverce v i-tém rekurzivním kroku: $P_1^{(i)}, P_2^{(i)}, P_3^{(i)}, P_4^{(i)}$.

$P_1^{(i)} = [x_1^{(i)}, y_1^{(i)}]$, $P_2^{(i)} = [x_2^{(i)}, y_2^{(i)}]$, $P_3^{(i)} = [x_3^{(i)}, y_3^{(i)}]$, $P_4^{(i)} = [x_4^{(i)}, y_4^{(i)}]$.

Vrcholy čtverce v $i+1$-tém rekurzivním kroku: $P_1^{(i+1)}, P_2^{(i+1)}, P_3^{(i+1)}, P_4^{(i+1)}$.

$P_1^{(i+1)} = [x_1^{(i+1)}, y_1^{(i+1)}]$, $P_2^{(i+1)} = [x_2^{(i+1)}, y_2^{(i+1)}]$, $P_3^{(i+1)} = [x_3^{(i+1)}, y_3^{(i+1)}]$,
$P_4^{(i+1)} = [x_4^{(i+1)}, y_4^{(i+1)}]$.

Jaký je vztah mezi souřadnicemi vrcholů dvou čtverců v různých úrovních rekurze i a $i+1$:

$P_1^{(i+1)}$	$x_1^{(i+1)} = \frac{x_1^{(i)}+x_2^{(i)}}{2}$	$y_1^{(i+1)} = \frac{y_1^{(i)}+y_2^{(i)}}{2}$
$P_2^{(i+1)}$	$x_2^{(i+1)} = \frac{x_2^{(i)}+x_3^{(i)}}{2}$	$y_2^{(i+1)} = \frac{y_2^{(i)}+y_3^{(i)}}{2}$
$P_3^{(i+1)}$	$x_3^{(i+1)} = \frac{x_3^{(i)}+x_4^{(i)}}{2}$	$y_3^{(i+1)} = \frac{y_3^{(i)}+y_4^{(i)}}{2}$
$P_4^{(i+1)}$	$x_4^{(i+1)} = \frac{x_4^{(i)}+x_1^{(i)}}{2}$	$y_4^{(i+1)} = \frac{y_4^{(i)}+y_1^{(i)}}{2}$

Podmínka ukončením rekurze: délka strany nenatočenéjo čtverce menší než ε.

56. Zápis zdrojového kódu

```
def square(x1, y1, x2, y2, x3, y3, x4, y4, draw):
    if (abs(x2 - x1) > 0) | (abs(y2 - y1) > 0):  # Podminka ukonceni rekurze
        # Kresba ctverce
        draw.line((x1, y1, x2, y2), 'black')
        draw.line((x2, y2, x3, y3), 'black')
        draw.line((x3, y3, x4, y4), 'black')
        draw.line((x4, y4, x1, y1), 'black')
        # Kresli dalsi ctverec
        square((x1 + x2) // 2, (y1 + y2) // 2, (x2 + x3) // 2, (y2 + y3) // 2,
               (y3 + y4) // 2, (x4 + x1) // 2, (y4 + y1) // 2, draw);
```

Volání funkce:

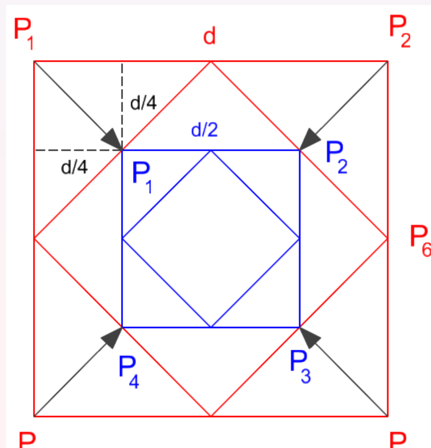
```
from PIL import Image, ImageDraw
im = Image.new('RGB', (800, 600), color='white')
draw = ImageDraw.Draw(im)
d=10
square (0,0,d,0,d,d,0,d);
```

57. Konstrukce po dvojicích čtverců

Vztah mezi vrcholy čtverců z kroků (i) a $(i + 1)$.

Počátek souřadnicového sytému v P_1.

Orientace os: $x \approx \overline{P_1, P_2}$, $y \approx \overline{P_1, P_4}$.



58. Konstrukce po dvojicích čtverců

Vrcholy čtverců v i-tém rekurzivním kroku: $P_1^{(i)}, P_2^{(i)}, P_3^{(i)}, P_4^{(i)}$ (vnější čtverec) a $P_5^{(i)}, P_6^{(i)}, P_7^{(i)}, P_8^{(i)}$ (vnitřní čtverec):

$P_1^{(i)}$	$x^{(i)}$	$y^{(i)}$
$P_2^{(i)}$	$x^{(i)} + d$	$y^{(i)}$
$P_3^{(i)}$	$x^{(i)} + d$	$y^{(i)} + d$
$P_4^{(i)}$	$x^{(i)}$	$y^{(i)} + d$

$P_5^{(i)}$	$x^{(i)} + \frac{d}{2}$	$y^{(i)}$
$P_6^{(i)}$	$x^{(i)} + d$	$y^{(i)} + \frac{d}{2}$
$P_7^{(i)}$	$x^{(i)} + \frac{d}{2}$	$y^{(i)} + d$
$P_8^{(i)}$	$x^{(i)}$	$y^{(i)} + \frac{d}{2}$

Vrcholy čtverců v $i+1$-tém rekurzivním kroku posunuty o hodnotu $\pm\frac{d}{4}$, velikost strany se zmenší na polovinu $d^{(i+1)} = \frac{d^{(i)}}{2}$.

$P_1^{(i+1)}$	$x_1^{(i)} + \frac{d}{4}$	$y_1^{(i)} + \frac{d}{4}$
$P_2^{(i+1)}$	$x_2^{(i)} - \frac{d}{4}$	$y_2^{(i)} + \frac{d}{4}$
$P_3^{(i+1)}$	$x_3^{(i)} - \frac{d}{4}$	$y_3^{(i)} - \frac{d}{4}$
$P_4^{(i+1)}$	$x_4^{(i)} + \frac{d}{4}$	$y_4^{(i)} - \frac{d}{4}$

$P_5^{(i+1)}$	$x_1^{(i+1)} + \frac{d}{2}$	$y_1^{(i+1)}$
$P_6^{(i+1)}$	$x_1^{(i+1)} + d$	$y_1^{(i+1)} + \frac{d}{2}$
$P_7^{(i+1)}$	$x_1^{(i+1)} + \frac{d}{2}$	$y_1^{(i+1)} + d$
$P_8^{(i+1)}$	$x_1^{(i+1)}$	$y_1^{(i+1)} + \frac{d}{2}$

59. Zápis zdrojového kódu

```
def square2 ( x, y, d, draw):
    if d>0:  #Podminka ukonceni rekurze
        #Kresba ctverce 1
        draw.line((x,y, x + d,y), 'black' );
        draw.line((x + d, y, x + d, y + d), 'black');
        draw.line((x + d, y + d, x, y + d), 'black');
        draw.line((x, y + d, x, y), 'black');
        #Kresba ctverce 2
        draw.line((x + d // 2, y, x + d, y + d//2), 'black');
        draw.line((x + d, y + d // 2, x + d // 2, y + d), 'black');
        draw.line((x + d // 2, y + d, x, y + d // 2), 'black');
        draw.line((x, y + d // 2, x + d // 2, y), 'black');
        #Rekurzivni volani funkce
        square2(x + d // 4, y + d // 4, d // 2, draw)
```

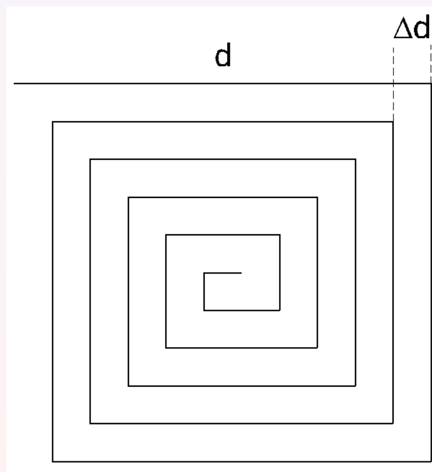
Volání funkce:

```
from PIL import Image, ImageDraw
im = Image.new('RGB', (800, 600), color='white')
draw = ImageDraw.Draw(im)
d=10
square2(0, 0, d, draw)
```

60. Konstrukce spirály rekurzí

Zkonstruujte za použití rekurze spirálu.

Délka spirály d, zkrácení spirály Δd.



61. Rozbor problému

Lze problém řešit prostřednictvím rekurze?

- Úloha je dekomponovatelná na úlohy stejné třídy: opakovaná konstrukce jednoho závitu spirály tvořeného 4 úsečkami.
- V dalším kroku rekurze nedochází k zesložitění problému.
- Podmínka ukončení rekurze: $d - \Delta d <= 0$.

Možné řešení problému:

Zkonstruujeme jeden závit spirály tvořený 4 segmenty.

Délky prvního a druhého segmentu d, délka třetího a čtvrtého segmentu $d - \Delta d$.

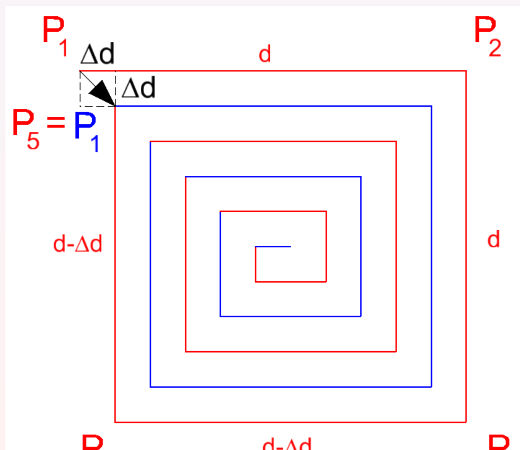
Stejným způsobem rekurzí zkonstruujeme další závity.

62. Znázornění spirály

Vztah mezi závity z i-tého a $i+1$ kroku.

Počátek souřadnicového sytému v P_1.

Orientace os: $x \approx \overline{P_1, P_2}$, $y \approx \overline{P_1, P_4}$.



63. Konstrukce spirály

Vrcholy závitu v *i*-tém rekurzivním kroku: $P_1^{(i)}, P_2^{(i)}, P_3^{(i)}, P_4^{(i)}$. Délka $d^{(i)}$.

$P_1^{(i)}$	$x_1^{(i)} + d^{(i)}$	$y_1^{(i)}$
$P_2^{(i)}$	$x_2^{(i)} + d^{(i)}$	$y_2^{(i)} + d^{(i)}$
$P_3^{(i)}$	$x_3^{(i)} + d^{(i)}$	$y_3^{(i)} + d^{(i)}$
$P_4^{(i)}$	$x_4^{(i)} + \Delta d$	$y_4^{(i)} + d^{(i)}$

Nová poloha vrcholu $P_1^{(i+1)} = P_1 + \Delta d$.

Změna délky segmentu závitu $d^{(i+1)} = d^{(i)} - 2\Delta d$.

Vrcholy závitu v *i*-tém rekurzivním kroku: $P_1^{(i+1)}, P_2^{(i+1)}, P_3^{(i+1)}, P_4^{(i+1)}$.

$P_1^{(i+1)}$	$x_1^{(i)} + \Delta d$	$y_1^{(i)} + \Delta d$
$P_2^{(i+1)}$	$x_2^{(i)} - \Delta d$	$y_2^{(i)} + \Delta d$
$P_3^{(i+1)}$	$x_3^{(i)} - \Delta d$	$y_3^{(i)} - \Delta d$
$P_4^{(i+1)}$	$x_4^{(i)} + \Delta d$	$y_4^{(i)} - \Delta d$

64. Zápis zdrojového kódu

```
def spiral(x, y, d, diff, draw):
    if d - diff > 0 : #Test ukonceni rekurze
        draw.line((x, y, x + d, y), 'black');
        draw.line((x + d, y, x + d, y + d), 'black');
        draw.line((x + d, y + d, x + diff, y + d), 'black');
        draw.line((x + diff, y + d, x + diff, y + diff), 'black
        #Rekurzivni volani spiraly
        spiral(x + diff, y+diff, d-2 * diff, diff, draw)
```

Volání funkce:

```
m = Image.new('RGB', (800, 600), color='white')
draw = ImageDraw.Draw(im)
d = 600
diff = 10
spiral(0,0, d, diff, draw);
im.show()
```

65. 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

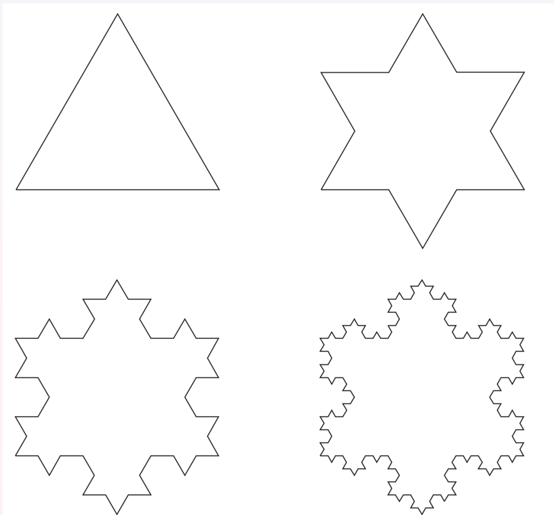
$$d = (\frac{4}{3})^n.$$

Pro n dělení platí

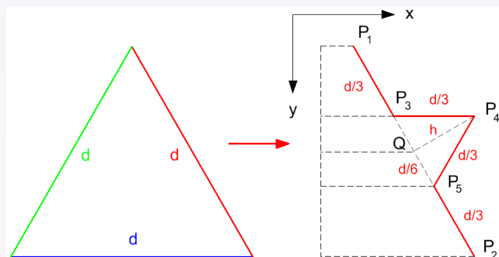
$$\lim_{n \to \infty} (d) = \infty.$$

66. Kochova vločka

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



67. 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 $\vec{u}$, normálový vektor $\vec{n}$, $\vec{n} \perp \vec{u}$.

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

68. 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

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

Bod P_4

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

69. 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()
```

70. Vhodnost/nevhodnost rekurze

- Při práci s rozsáhlými daty jsou rekurzivní algoritmy neefektivní
  Značné nároky na paměť.
- Lze ji 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).
- Nevhodná změna hodnoty proměnné ovlivňující ukončení rekurze způsobí nekonečnou rekurzi.
- Pro rozsáhlé datové soubory může “dojít paměť” dříve, než nalezeno řešení.
- Stručný, ale nepříliš přehledný kód
  Někdy není jasné, co algoritmus dělá.
- Používá se u řešení problémů, kde není známo nerekurzivní řešení.

71. Převod rekurze na iteraci

Cílem zefektivnění algoritmu, zejména snížení nároků na paměť.

Každý rekurzivní algoritmus lze přepsat na iterační.

U některých jazyků jediná možnost k provádění iterace představuje rekurze (nepoužívají cykly).

Možnosti převodu rekurze na iteraci:

- *Jednoduchá rekurze*
  Náhrada jedním cyklem (preference) či využití zásobníku popř. kombinace.

- *Vícenásobná rekurze*
  Každé rekurzivní volání reprezentováno samostatným vnořeným cyklem nebo využití zásobníku.
  Zásobník zpravidla používán pro složitější problémy, kde použití pouze cyklů nepřehledné.

Pro běžné typy problémů řešených rekurzí existují tzv. *rekurzivní schémata:* řešení problémů bez rekurze.

72. Odstranění jednoduché rekurze zásobníkem

Kombinace zásobníku (Stacku) + cyklu.

Dva kroky:

- *Převod lokálních proměnných na globální.*
  Lokální proměnné procedury/funkce převedeny na globální vzhledem k rekurzivní funkci.
- *Převod rekurze bez lokálních proměnných na iteraci.*
  Rekurzivní podmínka převedena na řídící podmínku cyklu
  Proměnné a mezivýsledky přidávány do zásobníku a následně dopočítávány jako při zpětné rekurzi.

Nevýhodou velká režie spojená se zásobníkem, hardwarově náročné.

Preferujeme cykly, pokud je to možné.

73. Jednoduchá rekurzivní funkce & Stack

```
f (n):                        #Rekurzivni funkce: prima rekurze
    S = [];
    S.append(D(n));           #Hodnota D pro n
    i = n;                    #Ridici promenna cyklu
    while i == podminka:
        C_i;                  #Blok prikazu, fce i, sestupne
        res_old = S.pop();    #Vyjme predchozireseni
        res = D(i,res_old)    #Aktualizace reseni
        S.append(res)         #Pridani noveho reseni
        E_{n-i};               #Blok prikazu, fce n vzestupne
        i--;                  #Dekrementace i
```

74. Fibonacciho čísla zásobníkem

```
fib (n):
    S=[];
    S.append (0);
    S.append (1);
    while n > 1:
        fi = S.pop();           #i clen
        fi_1= S.pop();          #i-1 clen
        fii = fi + fi_1;        #i+1 clen
        S.push(fi);             #Pridani jako i-1 clenu
        S.push(fii);            #Pridani jako i clenu
        n--;
```

Opět neefektivní, v každé iteraci dvojí přidávání a výběr ze zásobníku.

75. Rekurzivní schémata

Řada návrhů algoritmů vede k jejich rekurzivní implementaci.

Základní rekurzivní implementace většiny algoritmů jsou podobné, vedou ke vzniku rekurzivních schémat.

Nejčastější rekurzivní schémata:

- schéma if-else: F = {G, if P then F }
- schéma while-do: F = {G, while P do F }

V obou případech nemá smysl rekurzi používat, lze ji snadno nahradit iterací.

76. Rekurzivní schéma if-else

Často používané rekurzivní schéma s jedním voláním.

Náhrada úplné podmínky dvojicí cyklů.

Nejprve vyřešena kladná část podmínky, v samostatném cyklu větev else.

```
C ():
    if podminka(x):
        A(x)
        C()
        B(x)
    else:
        D(x)
```

```
C():
    n = 0
    while podminka(x):
        A(x)
        n+=1
    D(x)
    while (n !=0)
        B(x);
        n-=1
```

77. Rekurzivní schéma while-do

Rekurzivní schéma s jedním voláním opakováno v cyklu, dokud splněna podmínka.

Náhrada rekurze dvojicí cyklů + záměna cyklu while za do-while.

```
C():
    while podminka(x):
        A(x)
        C()
        B(x)
    D(x)
```

```
C():
    n = 0;
    do:
        while podminka(x):
            A(x)
            n+=1
        D(x)
        if (n==0) return;
        n-=1
        B(x)
    while (n!=0)
```