# Rekurzivní generování

úlohy typu „zkoušení všech možností“ nebo „generování všech možností“

**Vypsat všechna *K*-ciferná čísla v poziční soustavě o základu *n***

Pokud *k* je předem pevně dáno, např pro *k* = 4:

```
for i1 in range(n):
    for i2 in range(n):
        for i3 in range(n):
            for i4 in range(n):
                print(f"{i1} {i2} {i3} {i4}")
```

Je-li *k* vstupním údajem, nemůžeme toto zapsat pomocí vnořených cyklů – nevíme předem, kolik jich máme v programu napsat.

*Řešení:* rekurzivní funkce obsahující jeden takový cyklus, rekurzivní zanoření jde vždy do hloubky *k*

```
k = 4              # počet cifer
n = 3              # číselná soustava
c = [0] * k        # vytvářené číslo

def cislo(p):
    """vypíše všechna k-ciferná čísla
       v poziční soustavě o základu "n",
       "p" je pořadové číslo vybírané cifry
    """
    for i in range(n):
        c[p] = i
        if p < k-1:
            cislo(p+1)
        else:
            print(c)

cislo(0)
```

*Jiné řešení:* rekurzivní zanoření do hloubky *k*+1

```
k = 4              # počet cifer
n = 3              # číselná soustava
c = [0] * k        # vytvářené číslo

def cislo(p):
    """vypíše všechna k-ciferná čísla
       v poziční soustavě o základu "n",
       "p" je pořadové číslo vybírané cifry
    """
    if p == k:
        print(c)
    else:
        for i in range(n):
            c[p] = i
            cislo(p+1)

cislo(0)
```

**Variace s opakováním**
*k*-prvkové z *n*-prvkové množiny {1, 2, …, *n*}

= všechny uspořádané *k*-tice tvořené prvky z {1, 2, …, *n*}
s možností opakování hodnot

Např. pro *k* = 2, *n* = 4: (1,1) (1,2) (1,3) (1,4) (2,1), (2,2) (2,3) (2,4)
(3,1) (3,2) (3,3) (3,4) (4,1), (4,2) (4,3) (4,4)

*Řešení*: na každou z *k* pozic vytvářené variace
postupně umístíme každé z *n* čísel

→ zcela totéž jako předchozí úloha
(jenom místo čísel 0, …, *n*-1 umisťujeme čísla 1, …, *n*)

**Kombinace bez opakování**
*k*-prvkové z *n*-prvkové množiny {1, 2, …, *n*}

= všechny *k*-prvkové podmnožiny vybrané z množiny {1, 2, …, *n*}
(bez možnosti opakování hodnot)

Např. pro *k* = 2, *n* = 4: (1,2) (1,3) (1,4) (2,3) (2,4) (3,4)

*Řešení:* generujeme pouze ostře rostoucí *k*-tice
hodnot z množiny {1, 2, …, *n*}

```
k = 3                # počet prvků v kombinaci
n = 5                # z kolika prvků vybíráme
c = [0] * (k+1)      # vytvářená kombinace
# technický trik: c[0]=0, kombinace začíná až v c[1]

def kombinace(p):
    """vypíše všechny k-prvkové kombinace
       z "n" prvků bez opakování,
       "p" je pořadové číslo vybíraného prvku
    """
    if p > k:
        print(c[1:])
    else:
        for i in range(c[p-1]+1, n-(k-p)+1):
            c[p] = i
            kombinace(p+1)

kombinace(1)
```

**Doplnění znamének**

Je dáno *n* kladných celých čísel a požadovaný součet *c*.
Před čísla doplňte znaménka + nebo - tak, aby byl součet čísel se znaménky roven danému *c*. Nalezněte všechna řešení úlohy.

*Řešení:* před každé číslo zkusíme postupně dát + nebo -,
po vytvoření celé *n*-tice znamének otestujeme součet

$\rightarrow 2^n$ možností, tedy časová složitost algoritmu $O(2^n)$

```
cislo = [int(_) for _ in input().split()]
                        # uložení zadaných čísel
n = len(cislo)          # počet zadaných čísel
c = int(input())        # požadovaný výsledný součet
znam = [None] * n       # uložení znamének

def znamenko(p, soucet):
    ''' p = pozice nového znaménka,
        soucet = součet přechozích čísel se znaménky
    '''
    … viz další strana …

znamenko(0, 0)          # zavolání rekurzivní funkce
    # začínáme od indexu 0, dosavadní součet je 0
```

Parametr *soucet* není nutný, funkce si dokáže tento součet sama spočítat na základě údajů uložených v seznamech *cislo*, *znam* na indexech od 0 do *p*-1 (včetně).

```
def znamenko(p, soucet):
    ''' p = pozice nového znaménka,
        soucet = součet přechozích čísel se znaménky
    '''
    if p == n:
        if soucet == c:
            for i in range(n):
                print(znam[i], end = '')
                print(cislo[i], end = '')
            print()
    else:
        znam[p] = '+'
        znamenko(p+1, soucet+cislo[p])
        znam[p] = '-'
        znamenko(p+1, soucet-cislo[p])
```

**Rozklad čísla**

Zadané kladné celé číslo *n* rozložte všemi různými způsoby na součet kladných celých sčítanců. Rozklady lišící se pouze pořadím sčítanců nepovažujeme za různé.

*Příklad:*

$$
\begin{aligned}
5 &= 4 + 1 \\
&= 3 + 2 \\
&= 3 + 1 + 1 \\
&= 2 + 2 + 1 \\
&= 2 + 1 + 1 + 1 \\
&= 1 + 1 + 1 + 1 + 1
\end{aligned}
$$

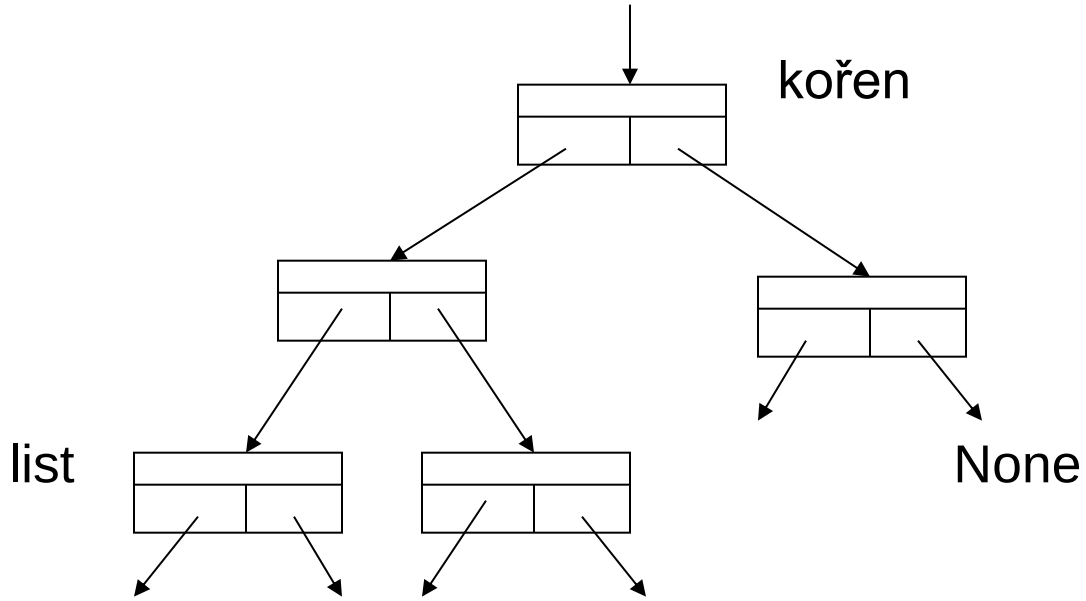
*Řešení:* Aby se neopakovaly stejné rozklady s různým pořadím sčítanců, budeme vytvářet pouze rozklady s nerostoucím pořadím sčítanců. Na každou pozici rozkladu vždy vyzkoušíme všechny přípustné hodnoty (minimálně 1, maximálně kolik ještě zbývá a maximálně kolik je na předchozí pozici). Provádíme, dokud je co rozkládat.

```
n = 7
a = [n+1] * (n+1) # prvek a[0] není součástí rozkladu

def rozklad(z, p):
    """
      z - kolik zbývá rozložit
      p - kolikátý sčítanec vytváříme
    """
    if z == 0:           # rozklad je hotov
        print(a[1:p])
    else:          # přidáme do a[p] p-tý člen rozkladu
        for i in range(1, min(z, a[p-1])+1):
            a[p] = i
            rozklad(z-i, p+1)

rozklad(n, 1);   # rozložit "n", začínáme 1. sčítancem
```

# Binární strom



```
class Vrchol:
    """vrchol binárního stromu"""

    def __init__(self, x = None):
        self.info = x                # uložená hodnota
        self.levy = None             # levý syn
        self.pravy = None            # pravý syn
```

***Příklady použití binárního stromu:***

- halda *(bylo)*
- binární vyhledávací strom *(bude)*
- reprezentace aritmetického výrazu *(bude)*

**Průchod binárním stromem (do hloubky)**
- v každém vrcholu provést zvolenou akci
(např. vypsat hodnotu atributu “info”)

*Varianty průchodu podle pořadí zpracování vrcholů:*

PREORDER – nejprve zpracuje vrchol, pak jde postupně do obou jeho podstromů

INORDER – nejprve jde do levého podstromu, pak zpracuje vrchol, nakonec jde do pravého podstromu

POSTORDER – nejprve jde postupně do obou podstromů vrcholu, pak zpracuje vrchol samotný

```
class Vrchol:
    """vrchol binárního stromu"""

    def __init__(self, x = None):
        self.info = x              # uložená hodnota
        self.levy = None           # levý syn
        self.pravy = None          # pravý syn

    def preorder(self):
        """průchod stromem s kořenem v tomto vrcholu
           metodou preorder, vypisuje hodnoty všech
           vrcholů
        """
        print(self.info)
        if self.levy != None:
            self.levy.preorder()
        if self.pravy != None:
            self.pravy.preorder()
```

```
def inorder(self):
    """průchod stromem s kořenem v tomto vrcholu
       metodou inorder, vypisuje hodnoty všech
       vrcholů"""
    if self.levy != None:
        self.levy.inorder()
    print(self.info)
    if self.pravy != None:
        self.pravy.inorder()

def postorder(self):
    """průchod stromem s kořenem v tomto vrcholu
       metodou postorder, vypisuje hodnoty všech
       vrcholů"""
    if self.levy != None:
        self.levy.postorder()
    if self.pravy != None:
        self.pravy.postorder()
    print(self.info)
```

## Průchod do hloubky bez použití rekurze – pomocí zásobníku

```
DO_ZÁSOBNÍKU(Kořen)
dokud ZÁSOBNÍK není prázdný
   P = ZE_ZÁSOBNÍKU
   AKCE(P.info)
   jestliže P.pravy != None: DO_ZÁSOBNÍKU(P.pravy)
   jestliže P.levy != None: DO_ZÁSOBNÍKU(P.levy)
```

## Průchod do šířky (po vrstvách) – pomocí fronty

```
DO_FRONTY(Kořen)
dokud FRONTA není prázdná
   P = Z_FRONTY
   AKCE(P.info)
   jestliže P.levy != None: DO_FRONTY(P.levy)
   jestliže P.pravy != None: DO_FRONTY(P.pravy)
```

*Časová složitost* všech uvedených metod průchodu:

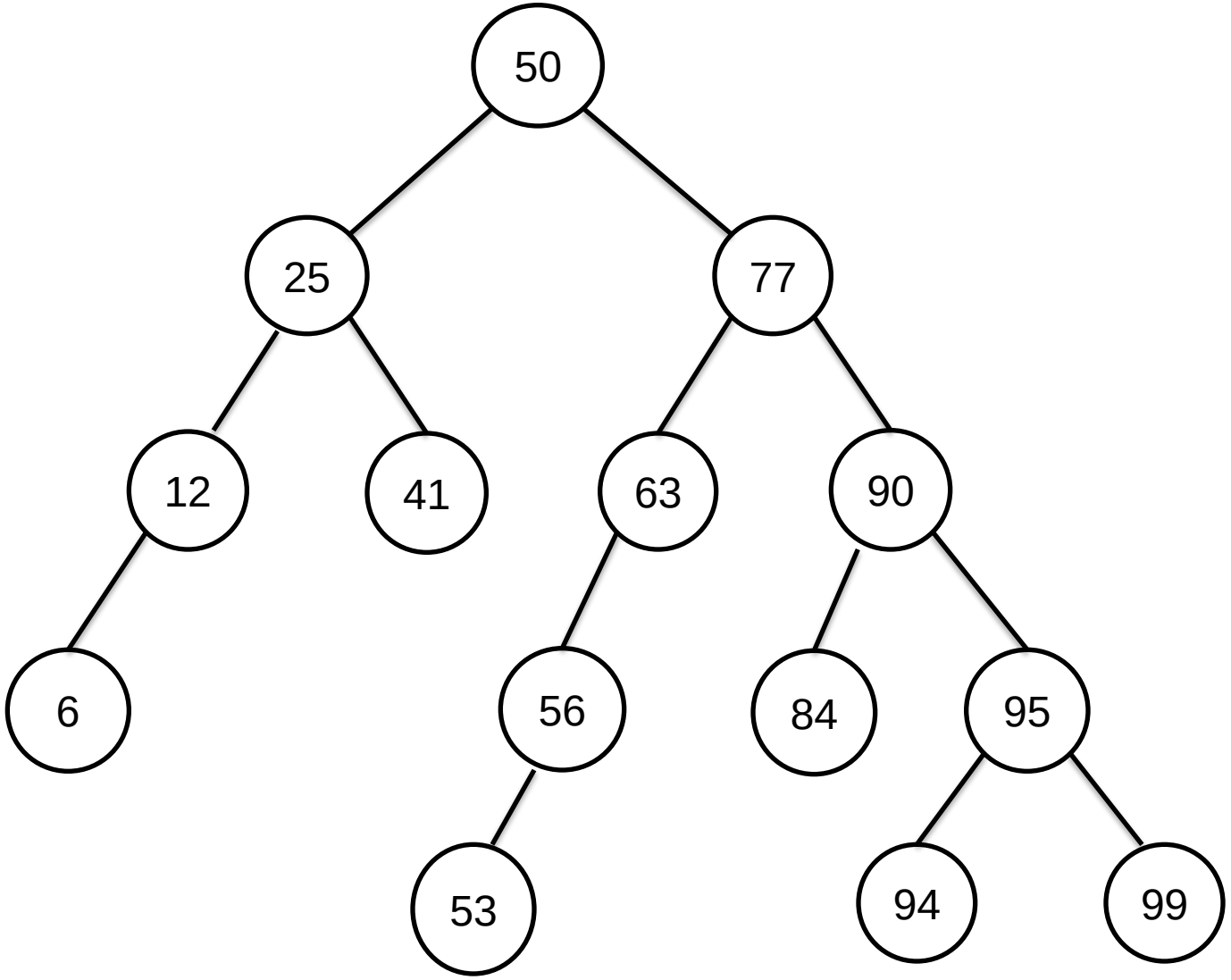
O(*N*), kde *N* je počet vrcholů ve stromu

- neboť každý vrchol stromu je navštíven právě jednou a jeho zpracování má konstantní časovou složitost

# Binární vyhledávací strom (BVS)

(BST – Binary Search Tree)

- datová struktura pro ukládání a vyhledávání dat podle klíče
- pro každý vrchol platí:
  všechny záznamy uložené v levém podstromu mají menší klíč
  všechny záznamy uložené v pravém podstromu mají větší klíč
  (platí pro ***všechny*** záznamy v podstromu, nestačí jen pro syny!)


50
25
77
12
41
63
90
6
56
84
95
53
94
99

Při vyhledávání není třeba procházet celý strom, stačí projít jednu cestu od kořene k listu
→ časová složitost algoritmu vyhledávání je v nejhorším případě určena ***výškou stromu***

**Výška *H* binárního stromu o *N* uzlech**
= délka nejdelší cesty z kořenu do listu

minimum – vyvážený strom
$N = 2^0 + 2^1 + \ldots 2^H = 2^{H+1} - 1 \quad \rightarrow \; H \approx \log_2 N$
maximum – degenerovaný strom $H \approx N$

v průměrném případě výška O(log *N*)

## Hledání hodnoty v BVS

```
def hledej(p, x):
    """
      hledání hodnoty "x" v BVS
      s kořenem ve vrcholu "p"
    """
    while p != None and p.info != x:
        if x < p.info:
            p = p.levy
        else:
            p = p.pravy
    return p
```

Rekurzivní řešení téže funkce:

```
def hledej(p, x):
    """
      hledání hodnoty "x" v BVS
      s kořenem ve vrcholu "p"
    """
    if p == None:
        return None
    elif x == p.info:
        return p
    elif x < p.info:
        return hledej(p.levy, x)
    else:
        return hledej(p.pravy, x)
```

Totéž zapsáno jinak:

```
def hledej(p, x):
    """
      hledání hodnoty "x" v BVS
      s kořenem ve vrcholu "p"
    """
    if p == None or x == p.info:
        return p
    return hledej(p.levy, x) if x < p.info \
        else hledej(p.pravy, x)
```

**Přidání hodnoty do BVS**
- novou hodnotu musíme přidat tam, kde ji budeme hledat
- přidává se vždy *do nového listu*
- postupuje se stejně jako při hledání hodnoty v BVS, dokud se nenarazí na odkaz **None** a do toho místa se přidá nový uzel
- časová složitost je opět dána výškou stromu, v průměrném případě O(log *N*)

*Technická realizace*:
- buď při průchodu stromem od kořene k listu udržovat pomocný ukazatel o jeden krok pozadu, pomocí něj pak připojit do stromu nový uzel
- nebo řešení pomocí rekurze

Rekurzivní řešení:

```
def pridej(p, x):
    """přidání hodnoty "x" do BVS
       s kořenem ve vrcholu "p„
       (pokud tam hodnota "x" ještě není)
    """
    if p is None:
        p = Vrchol(x)
    elif x < p.info:
        p.levy = pridej(p.levy, x)
    elif x > p.info:
        p.pravy = pridej(p.pravy, x)
    return p
```

**Vypuštění hodnoty z BVS**

- nejprve průchodem od kořene směrem k listu najít vrchol s vypouštěnou hodnotou (a jeho předchůdce ve stromě)
- pokud je to ***list***, zruší se (v předchůdci nastavit **None**)
- pokud má jen ***jednoho následníka***, vrchol se zruší a jeho následník se přepojí místo něj (předchůdce tedy bude místo na rušený vrchol ukazovat na jeho jediného následníka)
- pokud má ***dva následníky***, vrchol se nemůže fyzicky zrušit, smaže se jen jeho dosavadní hodnota a nahradí se jinou vhodnou hodnotou ze stromu. Tou je buď nejmenší hodnota z pravého podstromu nebo naopak největší hodnota z levého podstromu rušeného vrcholu. Tato náhradní hodnota leží jistě v listu nebo ve vrcholu s jediným následníkem – její původní vrchol tedy umíme snadno zrušit.

- časová složitost je opět dána výškou stromu, v průměrném případě $O(\log N)$ – celkově se prošlo stromem pouze jednou od kořene k listu

**Hledání hodnoty v BVS s použitím zarážky**
(alternativa k předchozímu řešení)

- podobné jako hledání v seznamu pomocí zarážky
- jeden speciální vrchol navíc slouží jako zarážka při hledání (vkládá se do něj hledaná hodnota)
- všechny odkazy „levy“, „pravy“ ve vrcholech stromu s hodnotou **None** nahradíme odkazy na zarážku