# 6. Složitější algoritmy, rekurze

## BAB37ZPR – Základy programování

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

# Přehled témat

- Část 1 – Příklady s využitím rekurze


P6.1 Umocňování

P6.2 Součet pole

P6.3 Řetězec pozpátku

P6.4 Detekce palindromu

P6.5 Číselné soustavy

P6.6 Permutace

P6.7 Mince

P6.8 Hanojské věže

# Část I

## Příklady s využitím rekurze

# Rekurze

- **Rekurze** = odkaz na sama sebe, definice za pomoci sebe sama
- **Rekurzivní funkce** = funkce volá sama sebe (i nepřímo)
- Je to hlavně způsob přemýšlení o řešení problémů
    - Řešení problému za pomoci řešení jednodušší varianty téhož problému
    - O menší instance se nemusíme příliš starat, musíme ale dodržovat **pravidla**
- Obvykle elegantní × je potřeba hlídat efektivitu

Rekurze bývá paměťově náročná.

## Pravidla pro správné použití rekurze

1. Dostatečně jednoduché vstupy musíme vyřešit přímo.
2. Rekurzivní volání musí být v nějakém smyslu jednodušší než je aktuální problém.

# I. Příklady s využitím rekurze

P6.1 Umocňování

P6.2 Součet pole

P6.3 Řetězec pozpátku

P6.4 Detekce palindromu

P6.5 Číselné soustavy

P6.6 Permutace

P6.7 Mince

P6.8 Hanojské věže

## Přímá definice

$$x^n = \Pi_{i=1}^{n} x = \underbrace{x \cdot x \cdot \dots \cdot x}_{n\text{-krát}}$$

## Rekurzivní definice

$$x^0 = 1$$
$$x^{n+1} = x\,x^n \qquad \text{for } n > 0$$

## Anatomie rekurze

- Základní / bázový případ *(base case)*
- Převedení problému na jednodušší
- Odkaz na sebe / Rekurzivní volání

# P6.1 Umocňování (2/3)

- Iterativně

```
>>> def power_iterative(x,n):
...     prod=1.0
...     for i in range(n): prod*=x
...     return prod
...
>>> power_iterative(2.0,10)
1024.0
```

- Rekurzivně

```
>>> def power_recursive(x,n):
...     if n<=0: return 1.0
...     return x*power_recursive(x,n-1)
...
>>> power_recursive(2.0,10)
1024.0
```

- Iterativně

```
>>> def power_iterative(x,n):
...     prod=1.0
...     for i in range(n): prod*=x
...     return prod
...
>>> power_iterative(2.0,10)
1024.0
```

- Rekurzivně

```
>>> def power_recursive(x,n):
...     if n<=0: return 1.0
...     return x*power_recursive(x,n-1)
...
>>> power_recursive(2.0,10)
1024.0
```

# Umocňování



- Přímočará verze

```
>>> def power_recursive(x,n):
...     if n<=0:
...         return 1.0
...     return x*power_recursive(x,n-1)
...
>>> power_recursive(2.0,10)
1024.0
```

- Stručnější verze

```
>>> def power_recursive2(x,n):
...     return x*power_recursive2(x,n-1) if n>0 else 1.0
...
>>> power_recursive(2.0,10)
1024.0
```

Volte verzi, která je pro vás čitelnější.

# Umocňování



- Přímočará verze

```
>>> def power_recursive(x,n):
...     if n<=0:
...         return 1.0
...     return x*power_recursive(x,n-1)
...
>>> power_recursive(2.0,10)
1024.0
```

- Stručnější verze

```
>>> def power_recursive2(x,n):
...     return x*power_recursive2(x,n-1) if n>0 else 1.0
...
>>> power_recursive(2.0,10)
1024.0
```

Volte verzi, která je pro vás čitelnější.

# Odbočka: Volání funkcí

## Co už víme o tom, jak probíhá volání funkcí?

- Zásobník volání (call stack)
  - hodnoty lokálních proměnných a parametrů
  - návratová adresa (kam se máme vrátit)

## Vztah rekurze a iterace

- každý rekurzivní program je možno přepsat jako iterativní za pomoci zásobníku
- (explicitní) zásobník zde simuluje zásobník volání funkcí
  - hodnoty lokálních proměnných
  - kde jsme byli (v původní funkci)
- často, ale ne vždy, se dá zjednoduši

# I. Příklady s využitím rekurze

P6.1 Umocňování

P6.2 Součet pole

P6.3 Řetězec pozpátku

P6.4 Detekce palindromu

P6.5 Číselné soustavy

P6.6 Permutace

P6.7 Mince

P6.8 Hanojské věže

# P6.2 Součet pole – iterativně

```
>>> def sum_array(a):
...     s=0
...     for x in a:
...         s+=x
...     return s
...
>>> a=[68, 0, 61, 34, 2, 51, 29, 10, 5, 45]
>>> sum_array(a)
305
```

Šlo by to napsat bez cyklu?

# P6.2 Součet pole – rekurzivně

```
>>> def sum_array_recursive(a):
...     if len(a)==0:
...         return 0
...     return a[0]+sum_array_recursive(a[1:])
...
>>> a=[68, 0, 61, 34, 2, 51, 29, 10, 5, 45]
>>> sum_array_recursive(a)
305
```

# Nepřímá rekurze

- rekurzivní funkce se nemusí volat přímo, ale i skrz jinou funkci

```
>>> def even(num):
...     print("even", num)
...     odd(num-1)
...
>>> def odd(num):
...     print("odd", num)
...     if num>1:
...         even(num-1)
...
>>> even (3)
even 3
odd 2
even 1
odd 0
```

# Každý cyklus lze nahradit rekurzí (1/2)

- Iterativní verze

```
>>> def count_to(n):
...     for i in range(1,n+1): print(i, end=" ")
...
>>> count_to(5)
1 2 3 4 5
```

- Rekurzivní verze

```
>>> def count_to_recursive(n):
...     count_to_recursive_inner(n,1)
...
>>> def count_to_recursive_inner(n,i):
...     if i<=n:
...         print(i, end=" "); count_to_recursive_inner(n,i+1)
...
>>> count_to_recursive(5)
1 2 3 4 5
```

# Každý cyklus lze nahradit rekurzí (1/2)

- Iterativní verze

```
>>> def count_to(n):
...     for i in range(1,n+1): print(i, end=" ")
...
>>> count_to(5)
1 2 3 4 5
```

- Rekurzivní verze

```
>>> def count_to_recursive(n):
...     count_to_recursive_inner(n,1)
...
>>> def count_to_recursive_inner(n,i):
...     if i<=n:
...         print(i, end=" "); count_to_recursive_inner(n,i+1)
...
>>> count_to_recursive(5)
1 2 3 4 5
```

# Každý cyklus lze nahradit rekurzí (2/2)

- Rekurzivně s vnitřní funkcí

```
>>> def count_to_recursive2(n):
...     def count_to_recursive_inner(i):
...         if i<=n:
...             print(i, end=" ")
...             count_to_recursive_inner(i+1)
...     count_to_recursive_inner(1)
...
>>> count_to_recursive2(5)
1 2 3 4 5
```

# Vnitřní funkce

```
>>> def count_to_recursive2(n):
...     def count_to_recursive_inner(i):
...         if i<=n:
...             print(i, end=" ")
...             count_to_recursive_inner(i+1)
...     count_to_recursive_inner(1)
...
>>> count_to_recursive2(5)
1 2 3 4 5
```

+ Skrytí soukromých funkcí
+ Sdílení proměnných vnější funkce
– Nemožnost znovupoužití
– Nemožnost samostatného odladění

# Porušení pravidel



```
>>> def wrong1(num):
...     return num*wrong1(num-1)
...
>>> wrong1(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in wrong1
  File "<stdin>", line 2, in wrong1
  File "<stdin>", line 2, in wrong1
  [Previous line repeated 983 more times]
RecursionError: maximum recursion depth exceeded
```

## Porušení pravidel 

```
>>> def wrong1(num):
...     return num*wrong1(num-1)
...
>>> wrong1(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in wrong1
  File "<stdin>", line 2, in wrong1
  File "<stdin>", line 2, in wrong1
  [Previous line repeated 983 more times]
RecursionError: maximum recursion depth exceeded
```

```
>>> def wrong2(num):
...     if num<=1:
...         return num
...
...     return 1+wrong1(num)
...
>>> wrong2(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in wrong2
  File "<stdin>", line 2, in wrong1
  File "<stdin>", line 2, in wrong1
  File "<stdin>", line 2, in wrong1
  [Previous line repeated 982 more times]
RecursionError: maximum recursion depth exceeded
```

```
>>> def wrong2(num):
...     if num<=1:
...         return num
...
...     return 1+wrong1(num)
...
>>> wrong2(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in wrong2
  File "<stdin>", line 2, in wrong1
  File "<stdin>", line 2, in wrong1
  File "<stdin>", line 2, in wrong1
  [Previous line repeated 982 more times]
RecursionError: maximum recursion depth exceeded
```

# Porušení pravidel 3/3

```
>>> def wrong3(num):
...     if num > 1000:
...         return 1000
...     return 1+wrong3(num)
...
>>> wrong3(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in wrong3
  File "<stdin>", line 4, in wrong3
  File "<stdin>", line 4, in wrong3
  [Previous line repeated 982 more times]
  File "<stdin>", line 2, in wrong3
RecursionError: maximum recursion depth exceeded in comparison
```

```
>>> def wrong3(num):
...     if num > 1000:
...         return 1000
...     return 1+wrong3(num)
...
>>> wrong3(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in wrong3
  File "<stdin>", line 4, in wrong3
  File "<stdin>", line 4, in wrong3
  [Previous line repeated 982 more times]
  File "<stdin>", line 2, in wrong3
RecursionError: maximum recursion depth exceeded in comparison
```

# Recursion Error

- Velikost zásobníku volání je typicky nějak omezena
- V Pythonu implicitně na 1000 volání, dá se změnit
- Překročení limitu zásobníku volání je chyba
- V Pythonu `Recursion Error`

Jinde též stack overflow

```
>>> import sys
>>> print(sys.getrecursionlimit())
1000
```

Znamená to, že nemáme při programování používat rekurzi?

- **NE!** znamená to, že ji máme používat rozumně
- Nezapomeňte, že rekurze je i způsob přemýšlení

# I. Příklady s využitím rekurze

P6.1 Umocňování

P6.2 Součet pole

P6.3 Řetězec pozpátku

P6.4 Detekce palindromu

P6.5 Číselné soustavy

P6.6 Permutace

P6.7 Mince

P6.8 Hanojské věže

# P6.3 Řetězec pozpátku



- Iterativně

```
>>> def reverse_iterative(s):
...     r="" # result
...     for i in range(len(s)-1,-1,-1):
...         r+=s[i]
...     return r
...
>>> print(reverse_iterative("dobry vecer"))
recev yrbod
```

- Iterativně

```
>>> def reverse_iterative(s):
...     r="" # result
...     for i in range(len(s)-1,-1,-1):
...         r+=s[i]
...     return r
...
>>> print(reverse_iterative("dobry vecer"))
recev yrbod
```

# P6.3 Řetězec pozpátku 2/2

- Rekurzivně

```
>>> def rev_rec1(s):
...     if len(s)==0:
...         return ""
...     return rev_rec1(s[1:])+s[0]
...
>>> print(rev_rec1("dobry vecer"))
recev yrbod
```

- Pythonská verze

```
>>> print("dobry vecer"[::-1])
recev yrbod
```

- Rekurzivně

```
>>> def rev_rec1(s):
...     if len(s)==0:
...         return ""
...     return rev_rec1(s[1:])+s[0]
...
>>> print(rev_rec1("dobry vecer"))
recev yrbod
```

- Pythonská verze

```
>>> print("dobry vecer"[::-1])
recev yrbod
```

# P6.3 Řetězec pozpátku



- Rekurzivně

```
>>> def rev_rec1(s):
...     if len(s)==0:
...         return ""
...     return rev_rec1(s[1:])+s[0]
...
>>> print(rev_rec1("dobry vecer"))
recev yrbod
```

- Pythonská verze

```
>>> print("dobry vecer"[::-1])
recev yrbod
```

# I. Příklady s využitím rekurze

P6.1 Umocňování

P6.2 Součet pole

P6.3 Řetězec pozpátku

P6.4 Detekce palindromu

P6.5 Číselné soustavy

P6.6 Permutace

P6.7 Mince

P6.8 Hanojské věže

# Detekce palindromu



- Palindrom je text, který má stejný význam nezávisle na směru čtení (zleva nebo zprava)
- Implementace v Pythonu je velmi jednoduchá díky triku z nimulého příkladu

```
>>> def je_palindrom_iter(slovo):
...     return slovo == slovo[::-1]
...
>>> je_palindrom_iter('ahoj')
False
>>> je_palindrom_iter('racecar')
True
>>> je_palindrom_iter('krk')
True
```

# P6.4 Detekce palindromu



- Palindrom je text, který má stejný význam nezávisle na směru čtení (zleva nebo zprava)
- Implementace v Pythonu je velmi jednoduchá díky triku z nimulého příkladu

```
>>> def je_palindrom_iter(slovo):
...     return slovo == slovo[::-1]
...
>>> je_palindrom_iter('ahoj')
False
>>> je_palindrom_iter('racecar')
True
>>> je_palindrom_iter('krk')
True
```

# P6.4 Detekce palindromu 2/2

- Jak na palindrom s rekurzí? Platí, že
  1. První a poslední písmeno textu jsou stejná
  2. Text mezi prvním a posledním písmenem je sám palindromem

```
>>> def je_palindrom_rek(slovo):
...     if len(slovo) <= 1:
...         return True
...     else:
...         return slovo[0] == slovo[-1] and je_palindrom_rek(slovo[1:-1])
...
>>> je_palindrom_rek('ahoj')
False
>>> je_palindrom_rek('racecar')
True
>>> je_palindrom_rek('krk')
True
```

# Detekce palindromu



- Jak na palindrom s rekurzí? Platí, že
  1. První a poslední písmeno textu jsou stejná
  2. Text mezi prvním a posledním písmenem je sám palindromem

```
>>> def je_palindrom_rek(slovo):
...     if len(slovo) <= 1:
...         return True
...     else:
...         return slovo[0] == slovo[-1] and je_palindrom_rek(slovo[1:-1])
...
>>> je_palindrom_rek('ahoj')
False
>>> je_palindrom_rek('racecar')
True
>>> je_palindrom_rek('krk')
True
```

# I. Příklady s využitím rekurze

P6.1 Umocňování

P6.2 Součet pole

P6.3 Řetězec pozpátku

P6.4 Detekce palindromu

P6.5 Číselné soustavy

P6.6 Permutace

P6.7 Mince

P6.8 Hanojské věže

## Problém

- Převeď číslo `n` v soustavě se základem `b` na řetězec.
- $5_{10} = 101_2$ protože $1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5$.
- Pokud $b > 10$ používáme $\mathrm{A} = 10$, $\mathrm{B} = 11, \ldots$
- Např. $2016_{10} = 7\mathrm{E}0_{16}$ protože $7 \cdot 16^2 + 14 \cdot 16^1 + 0 \cdot 16^0 = 2016$

## Myšlenka řešení

- Pokud `n < b` pak vrať číslici odpovídající `n`.
- Jinak
  - Najdi `n // b` v soustavě `b`.
  - Přidej nakonec číslici odpovídající `n % b`

# P6.5 Číselné soustavy


- Vrať `n` jako řetězec v číselné soustavě se základem `base`.

```
>>> def to_str(n, base):
...     assert(n>=0)
...     cislice = "0123456789ABCDEF"
...     if n < base:
...         return cislice[n]
...     return to_str(n // base, base) + cislice[n % base]
...
>>> print(to_str(5,2))
101
>>> print(to_str(2016,16))
7E0
```

- Vrať n jako řetězec v číselné soustavě se základem base.

```
>>> def to_str(n, base):
...     assert(n>=0)
...     cislice = "0123456789ABCDEF"
...     if n < base:
...         return cislice[n]
...     return to_str(n // base, base) + cislice[n % base]
...
>>> print(to_str(5,2))
101
>>> print(to_str(2016,16))
7E0
```

# P6.5 Číselné soustavy 2/4

- Vrať `n` jako řetězec v číselné soustavě se základem `base`.

```
>>> def to_str(n, base):
...     assert(n>=0)
...     cislice = "0123456789ABCDEF"
...     if n < base:
...         return cislice[n]
...     return to_str(n // base, base) + cislice[n % base]
...
>>> print(to_str(5,2))
101
>>> print(to_str(2016,16))
7E0
```

- Každé rekurzivní řešení je možné napsat bez rekurze

Ale může to být těžké.

- Zde Nutnost zapamatovat si lokální proměnné v jednotlivých voláních

Například v zásobníku (stack).

```
>>> def to_str_nonrecursive(n,base):
...     cislice = "0123456789ABCDEF"
...     stack=[n] # hodnoty n
...     n//=base
...     while n>0:
...         stack+=[n]
...         n//=base
...     result=""
...     for m in stack[::-1]:
...         result+=cislice[m % base]
...     return result
...
```

- Nerekurzivní řešení bez zásobníku

```
>>> def to_str_nonrecursive2(n,base):
...     assert(n>=0)
...     cislice="0123456789ABCDEF"
...     result=""
...     while True:
...         result=cislice[n % base]+result
...         n//=base
...         if n==0: break
...     return result
...
```

- Přidávání na začátek řetězce je pomalé (lineární).
- Skrytě kvadratický algoritmus.

# I. Příklady s využitím rekurze


P6.1 Umocňování

P6.2 Součet pole

P6.3 Řetězec pozpátku

P6.4 Detekce palindromu

P6.5 Číselné soustavy

P6.6 Permutace

P6.7 Mince

P6.8 Hanojské věže

# P6.6 Permutace

## Problém

Vytiskni všechny permutace prvků dané množiny `M`.

## Myšlenka řešení

- Vezmi každý prvek $m_i$ z `M`.
- Najdi všechny permutace prvků $M \setminus \{m_i\}$
- Ke každé na začátek přidej $m_i$

# P6.6 Permutace – implementace 1/2

```
>>> def tisk_permutaci(m):
...     """ Vytiskne vsechny permutace prvku v 'm' """
...     tisk_permutaci_acc(m, "")
...
>>> # acc - retezec pridavany na zacatek
>>> def tisk_permutaci_acc(m, acc):
...     if len(m)==0:
...         print(acc)
...     else:
...         for i in range(len(m)):
...             tisk_permutaci_acc(m[:i]+m[i+1:], acc+m[i]+" ")
...
```

# P6.6 Permutace – implementace 2/2

```
>>> tisk_permutaci(['a', 'b', 'c'])
a b c
a c b
b a c
b c a
c a b
c b a
```

# I. Příklady s využitím rekurze


P6.1 Umocňování

P6.2 Součet pole

P6.3 Řetězec pozpátku

P6.4 Detekce palindromu

P6.5 Číselné soustavy

P6.6 Permutace

P6.7 Mince

P6.8 Hanojské věže

# P6.7 Mince

## Problém

Vypiš všechny způsoby, jak zaplatit $x$ Kč mincemi.
Hodnoty mincí $h = \{50, 20, 10, 5, 2, 1\}$ Kč.

## Myšlenka řešení

- Vyber největší minci $h_i \leq x$. Pak jsou dvě možnosti:
  - Použij $h_i$ — zaplať $x - h_i$ pomocí $h_i, h_{i+1}, \ldots, h_n$ a přidej jednu $h_i$.
  - Nepoužij $h_i$ — zaplať $x$ pomocí $h_{i+1}, \ldots, h_n$

# Mince – implementace



```
>>> def zaplat(x):
...     h=[50, 20, 10, 5, 2, 1] # hodnoty mincí sestupně
...
...     def doplat(x, m, i):
...         """ m - kolik zaplaceno v počtech mincí
...             i - kterou mincí začít """
...         if x==0:
...             vytiskni_platbu(m, h)
...         else:
...             if x >= h[i]: # zaplať minci h[i]
...                 doplat(x-h[i], m[:i] + [m[i]+1] + m[i+1:], i)
...             if i < len(h)-1: # zaplať menšími, lze-li
...                 doplat(x, m, i+1)
...     doplat(x, len(h)*[0], 0) # zacatek fce zaplat
...
```

# P6.7 Mince – implementace 2/3

```
>>> def vytiskni_platbu(m,h):
...     """ m - pocty minci, h - hodnoty """
...     for j in range(len(h)):
...         if m[j]>0:
...             print("%3d*%3dKc" % (m[j],h[j]), end="")
...     print("")
...
```

# P6.7 Mince – implementace 3/3)

```
>>> zaplat(12)
  1* 10Kc  1*  2Kc
  1* 10Kc  2*  1Kc
  2*  5Kc  1*  2Kc
  2*  5Kc  2*  1Kc
  1*  5Kc  3*  2Kc  1*  1Kc
  1*  5Kc  2*  2Kc  3*  1Kc
  1*  5Kc  1*  2Kc  5*  1Kc
  1*  5Kc  7*  1Kc
  6*  2Kc
  5*  2Kc  2*  1Kc
  4*  2Kc  4*  1Kc
  3*  2Kc  6*  1Kc
  2*  2Kc  8*  1Kc
  1*  2Kc 10*  1Kc
 12*  1Kc
```

# Mince – rychlejší varianta



```
>>> def zaplat2(x):
...     h=[50, 20, 10, 5, 2, 1] # hodnoty minci sestupne
...
...     def doplat(x, m, i):
...         """ m - kolik zaplaceno v poctech minci
...             i - kterou minci zacit """
...         if x==0:
...             vytiskni_platbu(m, h)
...         else:
...             if x>=h[i]: # zaplat minci h[i]
...                 m[i]+=1
...                 doplat(x-h[i], m, i)
...                 m[i]-=1  # úklid
...             if i<len(h)-1: # zaplat mensimi
...                 doplat(x, m, i+1)
...     doplat(x, len(h)*[0], 0)
```

# P6.7 Mince – rychlejší varianta 

```
>>> zaplat2(12)
  1*  10Kc   1*   2Kc
  1*  10Kc   2*   1Kc
  2*   5Kc   1*   2Kc
  2*   5Kc   2*   1Kc
  1*   5Kc   3*   2Kc   1*   1Kc
  1*   5Kc   2*   2Kc   3*   1Kc
  1*   5Kc   1*   2Kc   5*   1Kc
  1*   5Kc   7*   1Kc
  6*   2Kc
  5*   2Kc   2*   1Kc
  4*   2Kc   4*   1Kc
  3*   2Kc   6*   1Kc
  2*   2Kc   8*   1Kc
  1*   2Kc  10*   1Kc
 12*   1Kc
```

# I. Příklady s využitím rekurze

P6.1 Umocňování

P6.2 Součet pole

P6.3 Řetězec pozpátku

P6.4 Detekce palindromu

P6.5 Číselné soustavy

P6.6 Permutace

P6.7 Mince

P6.8 Hanojské věže

# P6.8 Hanojské věže



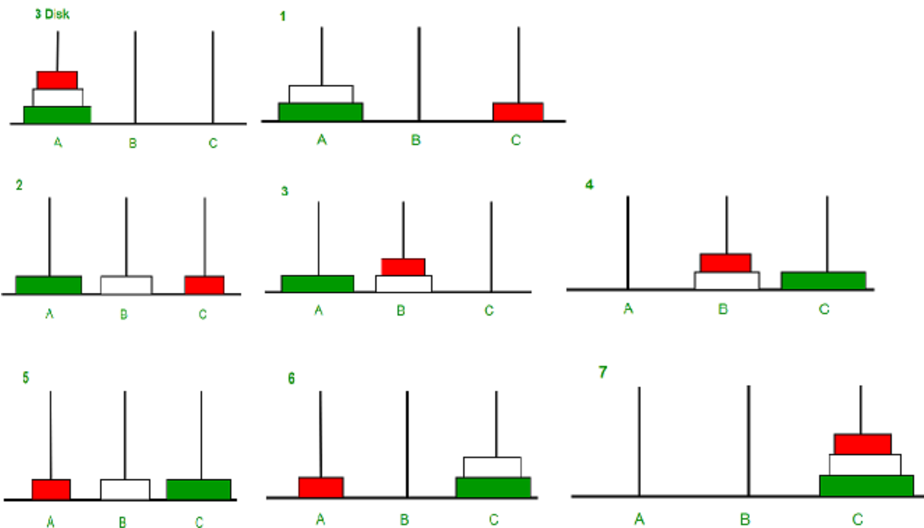
## Problém

Hanojská věž je matematický hlavolam, ve kterém máme tři tyče a `n` disků. Cílem hlavolamu je přesunout celou hromadu na jinou tyč.

## Pravidla

1. Najednou lze přesunout pouze jeden disk.
2. Každý přesun spočívá v tom, že se vezme horní disk z jedné hromádky a umístí se na jinou hromádku, tj. disk lze přesunout pouze tehdy, je-li nejvyšším diskem na hromádce.
3. Žádný disk nesmí být umístěn na horní část menšího disku.

**Poznámka:** Přenesení horních `n-1` disků ze zdrojové tyče na pomocnou tyč lze opět považovat za nový problém a lze jej řešit stejným způsobem.


3 Disk
A
B
C
1
A
B
C
2
A
B
C
3
A
B
C
4
A
B
C
5
A
B
C
6
A
B
C
7
A
B
C

```
>>> def TowerOfHanoi(n, odkud, kam, pomoc):
...     if n==1:
...         print ("Přesouvám 1 z", odkud, "na", kam); return
...     TowerOfHanoi(n-1, odkud, pomoc, kam)
...     print ("Přesouvám", n, "z", odkud, "na", kam)
...     TowerOfHanoi(n-1, pomoc, kam, odkud)
...
>>> TowerOfHanoi(3, 'A', 'B', 'C')
Přesouvám 1 z A na B
Přesouvám 2 z A na C
Přesouvám 1 z B na C
Přesouvám 3 z A na B
Přesouvám 1 z C na A
Přesouvám 2 z C na B
Přesouvám 1 z A na B
```