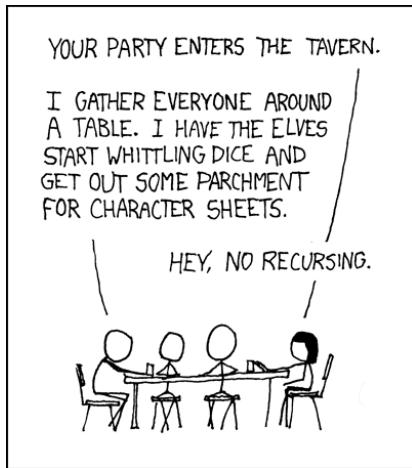


# Rekurze

IB111 Základy programování  
Radek Pelánek

2017

# xkcd: Tabletop Roleplaying

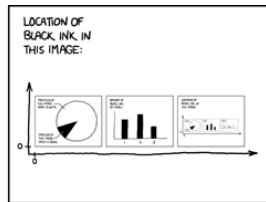
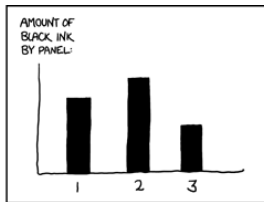
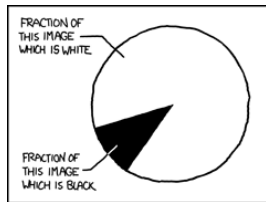


<https://xkcd.com/244/>

To iterate is human, to recurse divine. (L. Peter Deutsch)

- použití funkce při její vlastní definici
- volání sebe sama (s jinými parametry)

# Sebereference



<https://xkcd.com/688/>

# Sebereferenční test

- 1 Které písmeno není správnou odpovědí na žádnou otázku:  
(A) A (B) C (C) B
- 2 Odpověď na 3. otázku je:  
(A) stejná jako na 1. otázku (B) stejná jako na 2. otázku (C) jiná než odpovědi na 1. a 2. otázku
- 3 První otázka, na kterou je odpověď A, je otázka:  
(A) 1 (B) 2 (C) 3

Hlavalamikon

- 5 pirátů si dělí poklad: 100 mincí
- nejstarší pirát navrhne rozdělení, následuje hlasování
- alespoň polovina hlasů  $\Rightarrow$  rozděleno, hotovo
- jinak  $\Rightarrow$  navrhuující pirát zabit, pokračuje druhý nejstarší (a tak dále) (rekurze!)
- priority
  - 1 přežít
  - 2 mít co nejvíce mincí
  - 3 zabít co nejvíc ostatních pirátů

složitější varianty: 6 pirátů a 1 mince, 300 pirátů a 100 mincí

## Rekurze a sebereferece – klíčové myšlenky v informatice

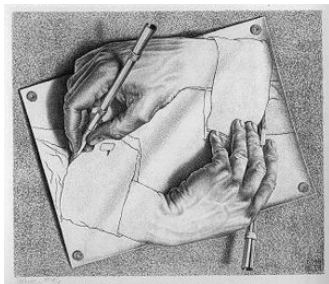
některé souvislosti:

- matematická indukce
- funkcionální programování
- rekurzivní datové struktury (např. stromy)
- gramatiky
- logika, neúplnost
- nerozhodnutelnost, diagonalizace



# Rekurze a sebereferece

... nejen v informatice



M. C. Escher; Drawing Hands, 1948

Gödel, Escher, Bach: An Eternal Golden Braid; Douglas Hofstadter

# Rekurze a úvodní programování

- uvedené aplikace rekurze a sebereferece často poměrně náročné
- hodí se **pořádně** pochopit rekurzi na úrovni jednoduchých programů
- bezprostřední návaznost – Algoritmy a datové struktury

# Faktoriál

$$n! = 1 \cdot 2 \cdots (n - 1) \cdot n$$

$$f(n) = \begin{cases} 1 & \text{pokud } n = 1 \\ n \cdot f(n - 1) & \text{pokud } n > 1 \end{cases}$$

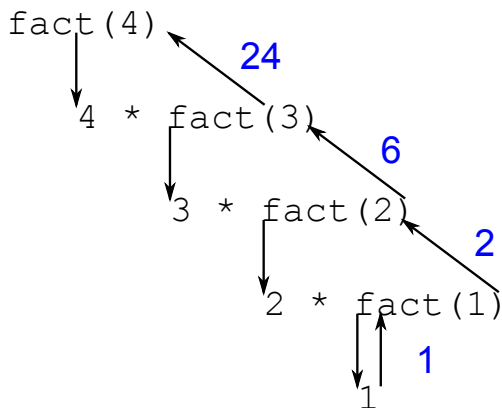
# Faktoriál iterativně (pomocí cyklu)

```
def fact(n):  
    f = 1  
    for i in range(1, n+1):  
        f = f * i  
    return f
```

# Faktoriál rekurzivně

```
def fact(n):  
    if n == 1: return 1  
    else: return n * fact(n-1)
```

# Faktoriál rekurzivně – ilustrace výpočtu



# Příklad: výpis čísel

Vymyslete funkci, která:

- vypíše čísla od 1 do N
- pomocí rekurze – bez použití cyklů `for`, `while`

# Příklad: výpis čísel

Vymyslete funkci, která:

- vypíše čísla od 1 do N
- pomocí rekurze – bez použití cyklů `for`, `while`

```
def sequence(n):  
    if n > 1:  
        sequence(n-1)  
    print(n)
```

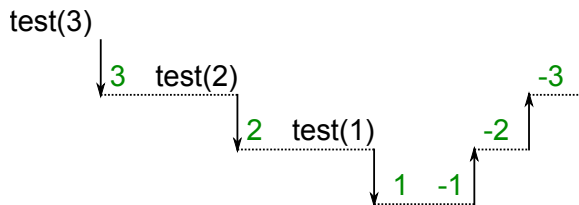


# Co udělá tento program?

```
def test(n):  
    print(n)  
    if n > 1:  
        test(n-1)  
    print(-n)
```

```
test(5)
```

# Ilustrace zanořování



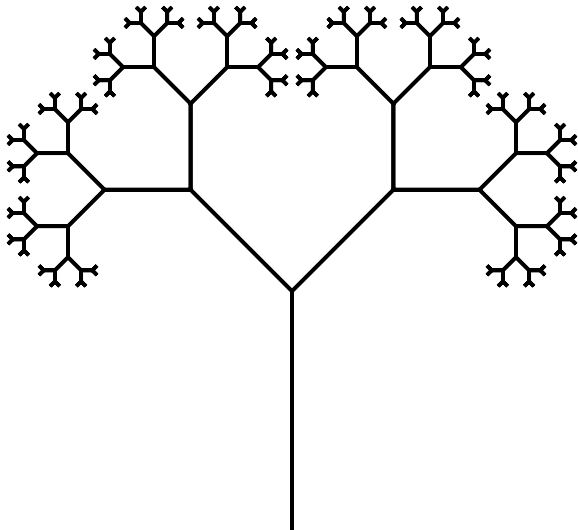
# Nepřímá rekurze

```
def even(n):  
    print("even", n)  
    odd(n-1)
```

```
def odd(n):  
    print("odd", n)  
    if n > 1:  
        even(n-1)
```

```
even(10)
```

# Rekurzivní stroměček



# Rekurzivní stromček

nakreslit stromček znamená:

- udělat stonek
- nakreslit dva menší stromčky (pootočené)
- vrátit se na původní pozici

# Stromeček želví grafikou

```
def tree(length):  
    forward(length)  
    if length > 10:  
        left(45)  
        tree(0.6 * length)  
        right(90)  
        tree(0.6 * length)  
        left(45)  
    back(length)
```

# Podoby rekurze, odstranění rekurze

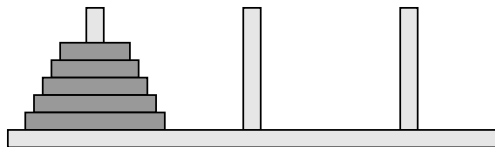
- koncová rekurze (tail recursion)
  - rekurzivní volání je poslední příkaz funkce
  - lze vešměs přímočaře nahradit cyklem
- „plná“ rekurze
  - „zanořující se“ volání
  - např. stromeček
  - lze přepsat bez použití rekurze za použití zásobníku
  - rekurzivní podoba často výrazně elegantnější

# Hanojské věže aneb O konci světa

- video:
  - [http://www.fi.muni.cz/~xpelane/IB111/hanojske\\_veze/](http://www.fi.muni.cz/~xpelane/IB111/hanojske_veze/)
  - <https://www.youtube.com/watch?v=8yaoED8jc8Y>
- klášter kdesi vysoko v horách u města Hanoj
- velká místnost se třemi vyznačenými místy
- 64 různě velkých zlatých disků
- podle věštby mají mniši přesouvat disky z prvního na třetí místo
- a až to dokončí ...

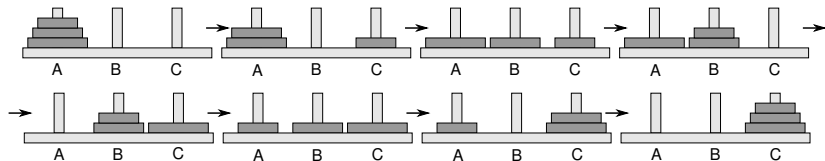


# Hanojské věže: pravidla



- $N$  disků různých velikostí naskládaných na sobě
- vždy může být jen menší disk položen na větším
- možnost přesunout jeden horní disk na jiný kolíček
- cíl: přesunout vše z prvního na třetí

# Hanojské věže: řešení



# Hanojské věže: výstup programu

```
>>> solve(3, "A", "B", "C")
```

```
A -> B
```

```
A -> C
```

```
B -> C
```

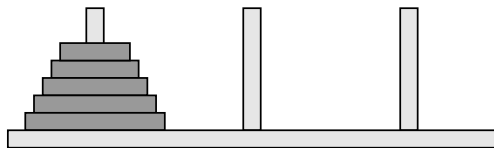
```
A -> B
```

```
C -> A
```

```
C -> B
```

```
A -> B
```

# Hanojské věže: rekurzivní řešení



```
def solve(n, start, end, auxiliary):  
    if n == 1:  
        print(start, "->", end)  
    else:  
        solve(n-1, start, auxiliary, end)  
        solve(1, start, end, auxiliary)  
        solve(n-1, auxiliary, end, start)
```

# Řešení včetně vypisování stavu

```
*  
***  
*****  
-----
```

```
***  
***** *
```

```
***** *** *
```

```
      *  
***** ***
```

```
      *  
     *** *****
```

```
*   *** *****
```

# Stav úlohy, reprezentace

- stav úlohy = rozmístění disků na kolících
- disky na kolíku A  $\rightarrow$  seznam
- celkový stav:
  - 3 proměnné, v každé seznam – nepěkné
  - *seznam seznamů*, kolíky interně značíme 0, 1, 2
- příklad stavu (6 disků):  $[[4], [5, 2, 1], [6, 3]]$

# Řešení včetně vypisování stavu

```
def solve(n, start, end, aux, state):  
    if n==1:  
        disc = state[start].pop()  
        state[end].append(disc)  
        print_state(state)  
    else:  
        solve(n-1, start, aux, end, state)  
        solve(1, start, end, aux, state)  
        solve(n-1, aux, end, start, state)  
  
def solve_hanoi(n):  
    state = [list(range(n,0,-1)), [], []]  
    print_state(state)  
    solve(n, 0, 2, 1, state)
```

# Vypisování stavu – jednoduše

```
def print_state(state):  
    print(state)  
    print("--")
```

```
def print_state(state):  
    for i in range(3):  
        print("Peg", chr(ord('A')+i), state[i])  
    print("--")
```



## Vypisování stavu – textová grafika

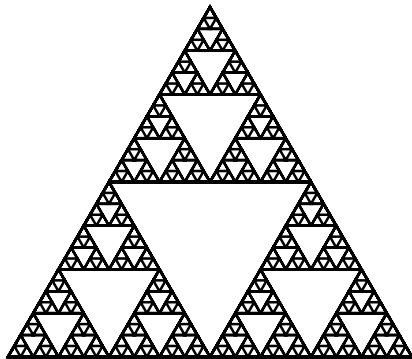
```
def print_state(state):
    n = sum([len(s) for s in state])
    for y in range(n+1):
        line = ""
        for peg in range(3):
            for x in range(-n+1, n):
                if len(state[peg]) > n-y and
                    abs(x) < state[peg][n-y]:
                    line += "*"
                else:
                    line += " "
            line += "  "
        print(line)
    print("-"*(n*6+1))
```

# Sierpińského fraktál

rekurzivně definovaný geometrický útvar



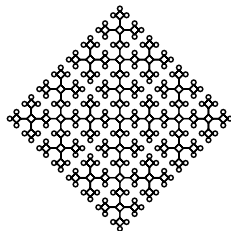
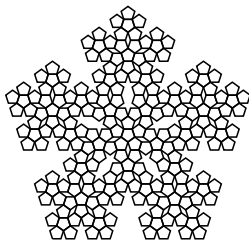
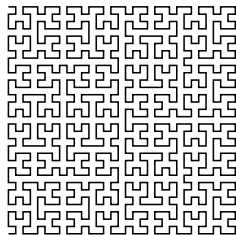
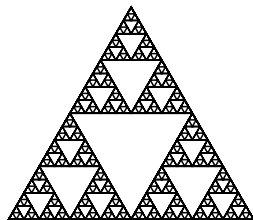
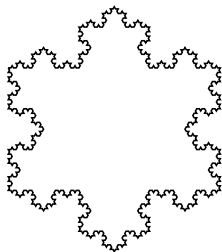
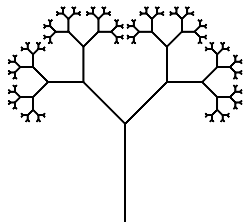
# Sierpińského fraktál



# Sierpińského fraktál: kód

```
def sierpinski(n, length):  
    if n == 1:  
        triangle(length)  
    else:  
        for i in range(3):  
            sierpinski(n - 1, length)  
            forward((2 ** (n - 1)) * length)  
            right(120)
```

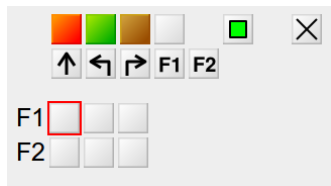
# Další podobné fraktály



- `tutor.fi.muni.cz`, úloha Robotanik
- jednoduché „grafické“ programování robota
- těžší příklady založeny na rekurzi
- vizualizace průběhu „výpočtu“, zanořování a vynořování z rekurze

# Robotanik – Kurz počítání

rekurze jako „paměť“



### Strom kytek 2

↑ ← → F1 F2 F3

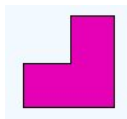
F1  
F2  
F3

Rychlost animace:  
1 2 3 4 5

Zobrazit zásobník  
 Klávesové zkratky  
 Přesun více příkazů

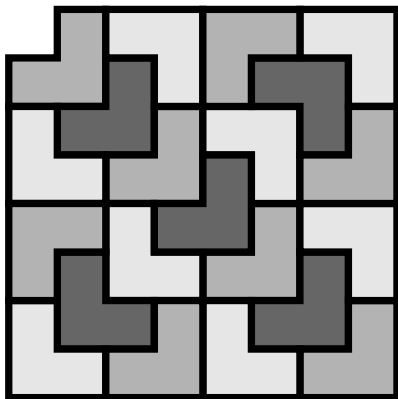
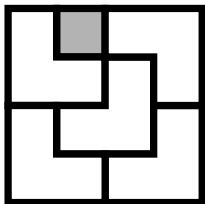


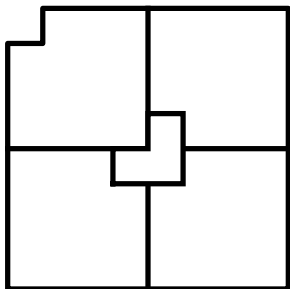
# Pokrývání plochy L kostičkami



- mřížka  $8 \times 8$  s chybějícím levým horním polem
- úkol: pokrýt zbývající políčka pomocí L kostiček
- rozšíření:
  - rozměr  $2^n \times 2^n$
  - chybějící libovolné pole
  - obarvení 3 barvami, aby sousedi byli různí

# Ukázky řešení





- rozdělit na čtvrtiny
- umístit jednu kostku
- rekurzivně aplikovat řešení na jednotlivé části

# Přemýšlení o funkcích (rekurzivních obzvlášť)

obecně pro funkce:

- **ujasnit vstupně-výstupní chování** než začnu psát funkci

rekurzivní funkce navíc:

- při psaní funkce předpokládám, že mám již funkci hotovou
- problém převedu na řešení menšího problému (který již umím vyřešit)
- ošetřím „koncovou podmínku“

# Otočení řetězce rekurzivně

```
>>> reverse("star wars")  
'sraw rats'
```

# Otočení řetězce rekurzivně

```
>>> reverse("star wars")  
'sraw rats'
```

```
def reverse(s):  
    if len(s) <= 1: return s  
    return reverse(s[1:]) + s[0]
```

# Kontrolní otázka

Co dělá následující funkce (vstup je řetězec)?

```
def magic(s):  
    if len(s) <= 1: return s  
    return magic(s[1:])
```

# Seznam vnořených seznamů

Seznam čísel a vnořených seznamů čísel (SČAVSČ) je seznam, který obsahuje čísla nebo SČAVSČ.

*rekurzivní datová struktura*

```
scavsc = [[2, 8], 4, [3, [1, 7], 6], [2, 4]]
```

Jak vypočítat součet všech čísel?



# Součet vnořených seznamů

```
def nested_list_sum(alist):  
    total = 0  
    for x in alist:  
        if isinstance(x, list):  
            total += nested_list_sum(x)  
        else:  
            total += x  
    return total
```

# Příklady použití rekurze v informatice

- Euclidův algoritmus – NSD
- vyhledávání opakovaným půlením
- řadicí algoritmy (quicksort, mergesort)
- generování permutací, kombinací
- fraktály
- prohledávání grafu do hloubky
- gramatiky

# Euklidův algoritmus rekurzivně

```
def nsd(a,b):  
    if b == 0:  
        return a  
    else:  
        return nsd(b, a % b)
```

# Vyhledávání opakovaným půlením

- hra na 20 otázek
- hledání v seznamu
- hledání v binárním stromu

## Vyhledávání: rekurzivní varianta

```
def binary_search(value, alist, lower, upper):  
    if lower > upper:  
        return False  
    mid = (lower + upper) // 2  
    if alist[mid] < value:  
        return binary_search(value, alist, mid+1, upper)  
    elif alist[mid] > value:  
        return binary_search(value, alist, lower, mid-1)  
    return True
```

- quicksort
  - vyber pivota
  - rozděl na menší a větší
  - zavolej quicksort na podčásti
- mergesort
  - rozděl na polovinu
  - každou polovinu seřaď pomocí mergesort
  - spoj obě poloviny

# Generování permutací, kombinací

- permutace množiny = všechna možná pořadí
  - příklad: permutace množiny  $\{1, 2, 3, 4\}$
  - jak je vypsát systematicky?
  - jak využít rekurzi?
- $k$ -prvkové kombinace  $n$ -prvkové množiny = všechny možné výběry  $k$  prvků
  - příklad: 3-prvkové kombinace množiny  $\{A, B, C, D, E\}$
  - jak je vypsát systematicky?
  - jak využít rekurzi?

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

```
def combinations(alist, k):
    if k == 0: return [[]]
    if len(alist) < k: return []
    output = []
    for comb in combinations(alist[1:], k-1):
        comb.append(alist[0])
        output.append(comb)
    output.extend(combinations(alist[1:], k))
    return output
```



# Nevhodné použití rekurze

- ne každé použití rekurze je efektivní
- Fibonacciho posloupnost (králíci):

$$f_1 = 1$$

$$f_2 = 1$$

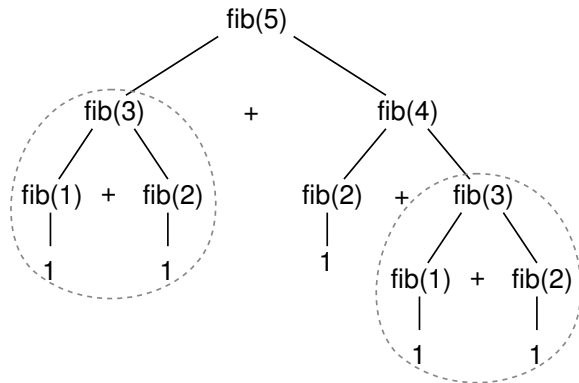
$$f_n = f_{n-1} + f_{n-2}$$

- 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- **Vi Hart:** Doodling in Math: Spirals, Fibonacci, and Being a Plant

# Fibonacciho posloupnost: rekurzivně

```
def fib(n):  
    if n <= 2: return 1  
    else: return fib(n-1) + fib(n-2)
```

# Fibonacciho posloupnost: rekurzivní výpočet



# Problém $N$ dam

- šachovnice  $N \times N$
- rozestavit  $N$  dam tak, aby se vzájemně neohrožovaly
- zkuste pro  $N = 4$

pozn. speciální případ „problému splnění podmínek“

- **rekurze**: využití **rekurze** pro definici sebe sama
- logické úlohy: Hanojské věže, L kostičky, dámy na šachovnici
- fraktály
- aplikace v programování: vyhledávání, řazení, prohledávání grafu
- klíčová myšlenka v informatice

*nezapomeňte na piráty*