Rekurze

Je cyklus při kterém se některé výpočty opakují pouze voláním programu (funkce):

1) Definujeme funkci a v těle této funkce výpočet, který chceme opakovat.
2) Samotné opakování realizujeme voláním funkce – nejčastěji samotné.
3) Rekurzivní funkce by měla zajistit, aby toto opakování nebylo nekonečné
4) Rekurze se často využívá k řešení úloh, které můžeme rozdělit na menší části a tyto části řešíme voláním funkce.

Nekonečná rekurze

Rekurze v programování tedy znamená, že funkce se volá sama sebe:

```
def xy():
    xy()

xy()
```

Rekurze v nejprve načítá výpočet do jmenného prostoru, takže pokud přesáhne limit cca 1000 vnořených volání, program spadne a objeví se hláška:

```
RecursionError: maximum recursion depth exceeded
```

Triviální případ – base case

Abychom předešli nekonečné rekurzi vkládá se do rekurze test – base case (triviální případ), který určí, kdy a v jakém případě rekurzivní volání končí pomocí podmínkou ‚if'.

3 možnosti zkrácení:

```
def vypis(n):
    if n < 1:
        pass
    else:
        vypis(n-1)
        print(n, end=', ')
```

```
def vypis(n):
    if n < 1:
        return
    vypis(n-1)
    print(n, end=', ')
```

```
def vypis(n):
    if n >= 1:
        vypis(n-1)
        print(n, end=', ')
```

Zásobník – stack

Informace o jmenném prostoru a návratové adrese si Python ukládá do údajové struktury zásobník.

Zásobní pracuje v systému LIFO – last in first out – kdy poslední přidaná položka, je první na řadě, která zpracovaná.

Každé další volání funkce vytváří nový jmenný prostor (položku), který se přidá na vrch zásobníku a při ukončení volání funkce se začne jeden po druhém z tohoto zásobníku odstraní, dokud není zásobník opět vyprázdněn.

Chvostová rekurze (nepravá rekurze)

Jedná se o program, který má rekurzivní volání pouze na konci. Dá se tak snadno nahradit např. cyklem ‚while'.

Pravá rekurze

Je program, který obsahuje některé příkazy před, nebo za rekurzním voláním. Většinou i ty se dají přepsat pomocí více cyklů, ale rekurze zjednodušuje samotný zápis.

Přiklad rekurze – Faktoriál

```
def faktorial(n):
    if n <= 1:
        return 1
    return faktorial(n-1) * n
```

Přiklad rekurze – Otočení řetězce

```
def otoc(retazec):
    if len(retazec) <= 1:
        return retazec
    return otoc(retazec[1:]) + retazec[0]
```

Tato funkce má stále omezení na přibližně 1000 znaků

Možným řešením může být rozdělení textu na menší úseky, ty pak rekurzívně řešit samostatně a po té opět pospojovat do jednoho výsledku (rozděl a panuj):

```
def otoc(retazec):
    if len(retazec) <= 1:
        return retazec
    stred = len(retazec) // 2
    prva = otoc(retazec[:stred])
    druha = otoc(retazec[stred:])
    return druha + prva

print(otoc('Bratislava'))
print(otoc('Bratislava' * 110))
print(otoc('Bratislava' * 220))
povodny = 'Bratislava' * 100000
r = otoc(povodny)
print(len(r), r == povodny[::-1])
```

Binomické koeficienty

Se dají vypočítat pomocí matematického vzorce:

```
bin(n, k) = n! / (k! * (n-k)!)
```

Výpočtem nějakých 3 faktoriálů a jejich dělením.

Tyto koeficienty můžeme zobrazit pomocí Pascalova trojúhelníku:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5  10  10   5   1
```

Pro tuto tabulku poznáme takovýto vztah:

```
bin(n, k) = bin(n-1, k-1) + bin(n-1, k)
```

To znamená, že každé číslo je součtem dvou čísel nad sebou, což se dá rekurzívně přepsat:

```
def bin(n, k):
    if k == 0 or n == k:
        return 1
    return bin(n-1, k-1) + bin(n-1, k)

for n in range(6):
    for k in range(n+1):
        print(bin(n, k), end=' ')
    print()
```

Fibonacciho čísla

Fungují na podobném principu, kdy každý další člen se vypočítá součtem předchozích dvou:

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

>>> for i in range(15):
        print(fib(i), end=', ')
    0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,
```

Tento rekurzivní algoritmus je ale velmi neefektivní vzhledem k tomu, že číslo rychle roste a za chvilku bychom počítaly tak velké čísla, že by operace trvali neuvěřitelně dlouho.

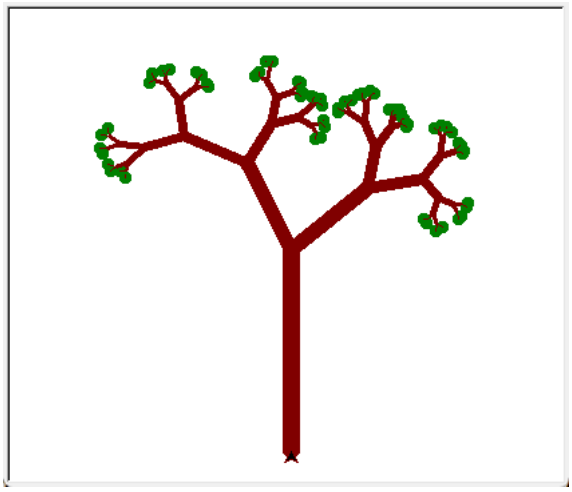
Proto operace s velkými čísli, je vždy lepší psát nerekurzívně.

Binární stromy

Jsou rekurzivní kresby, pro větvení (stromů), kde pro každou úroveň platí stejná, ale malinko obměněná pravidla:

1) Při úrovni 1 se kreslí pouze kmen
2) Při druhé a dalších úrovní se na konci předchozí čáry větev rozvětví – jedna se ohne doleva a pokračuje ve výpočtu a kreslení a druhá po té vpravo zopakuje výpočet.
3) Po skončení kreslení se pero nachází v bodě, kde kreslení začalo.
4) Levé i pravé větvení může mít stejné hodnoty, nebo můžou se měnit – např. zmenšovat.
5) Úroveň stromu vypovídá o počtu rekurzivních vnoření.
6) Pokud využijeme náhodný generátor, můžeme vytvářet stromy, které jsou různé.

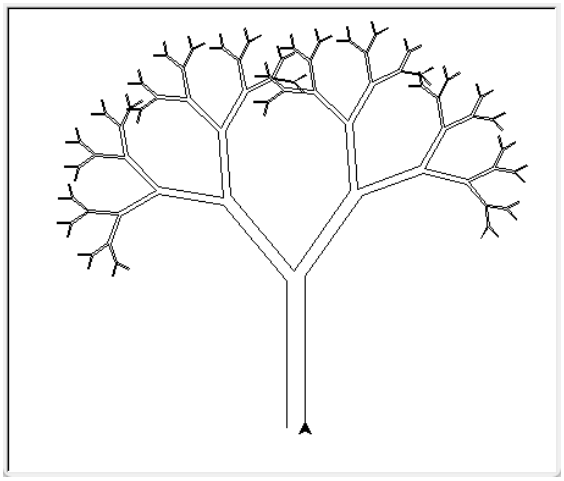
```
7)  import turtle
8)  import random
9)
10) def strom(n, d):
11)     t.pensize(2*n + 1)
12)     t.fd(d)
13)     if n == 0:
14)         t.dot(10, 'green')
15)     else:
16)         uhol1 = random.randint(20, 40)
17)         uhol2 = random.randint(20, 60)
18)         t.lt(uhol1)
19)         strom(n-1, d * random.randint(40, 70) / 100)
20)         t.rt(uhol1 + uhol2)
21)         strom(n-1, d * random.randint(40, 70) / 100)
22)         t.lt(uhol2)
23)     t.bk(d)
24)
25) turtle.delay(0)
26) t = turtle.Turtle()
27) t.lt(90)
28) t.pencolor('maroon')
29) strom(6, 150)
```



Binární strom se dá nakreslit i bez použití rekurze a to tak, že v každém větvení se vytvoří nové pero, které pokračuje jiným směrem a v poslední úrovni se nakreslí zelená tečka´.

V následujícím řešení je uvedeno, že při každém triviálním případě, udělá pero malý úkrok vpravo a nevrací se po stejných čarách a vzniká tak vektorový kmen:

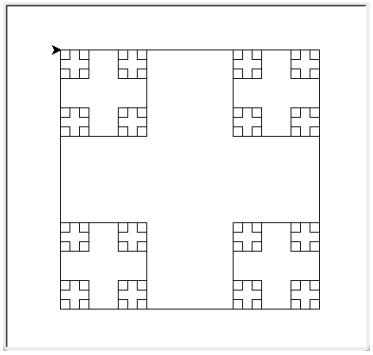
```
import turtle

def strom(n, d):
    t.fd(d)
    if n == 0:
        t.rt(90)
        t.fd(1)
        t.lt(90)
    else:
        t.lt(40)
        strom(n-1, d*0.67)
        t.rt(75)
        strom(n-1, d*0.67)
        t.lt(35)
    t.bk(d)

turtle.delay(0)
t = turtle.Turtle()
t.lt(90)
strom(6, 120)
```



Další rekurzivní obrázky:

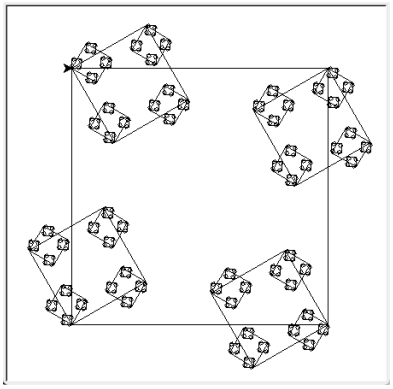
Nakresli čtverec a v každém jeho čtverci čtverec (jedná se o chvostovou rekurzi):

```
def stvorce(n, a):
    if n == 0:
        pass
    else:
        for i in range(4):
            t.fd(a)
            t.rt(90)
            stvorce(n-1, a/3)
```



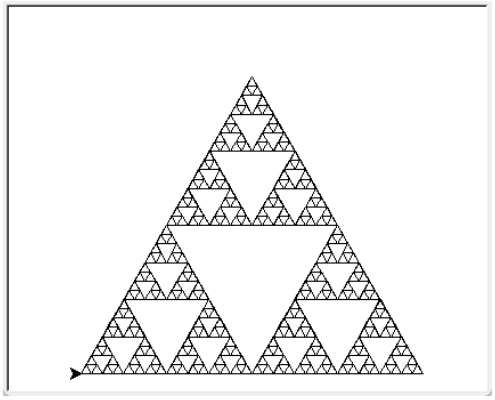
Když ale těsně před voláním rekurze otočíme pero o 30 stupňů a po návratu z rekurze těchto 30 stupňů vrátíme:

```
def stvorce(n, d):
    if n > 0:
        for i in range(4):
            t.fd(d)
            t.rt(90)
            t.lt(30)
            stvorce(n-1, d/3)
            t.rt(30)
```



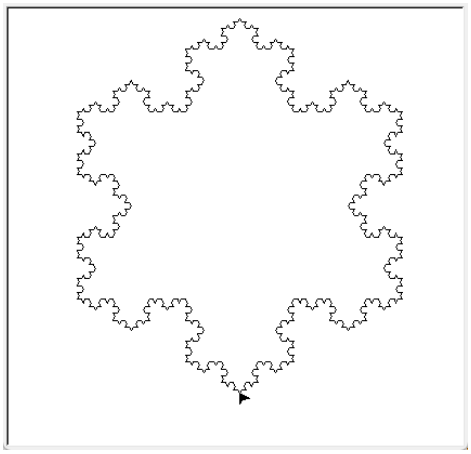
Sierpiňského trojúhelník:

```
def trojuholniky(n, a):
    if n > 0:
        for i in range(3):
            t.fd(a)
            t.lt(120)
            trojuholniky(n-1, a/2)
```



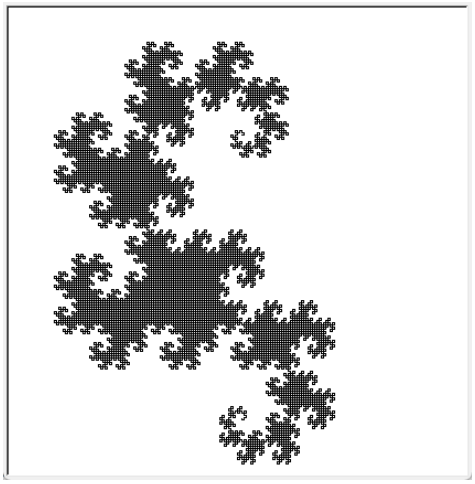
Sněhová vločka:

```
def vlocka(n, d):
    if n == 0:
        t.fd(d)
    else:
        vlocka(n-1, d/3)
        t.lt(60)
        vlocka(n-1, d/3)
        t.rt(120)
        vlocka(n-1, d/3)
        t.lt(60)
        vlocka(n-1, d/3)
def sneh_vlocka(n, d):
    for i in range(3):
        vlocka(n, d)
        t.rt(120)
```



C-křivka (dračí křivka):

```
def drak(n, s, u=90):
    if n == 0:
        t.fd(s)
    else:
        drak(n-1, s, 90)
        t.lt(u)
        drak(n-1, s, -90)
```



Hilbertova křivka:

```
def hilbert(n, s, u=90):
    if n > 0:
        t.lt(u)
        hilbert(n-1, s, -u)
        t.fd(s)
        t.rt(u)
        hilbert(n-1, s, u)
        t.fd(s)
        hilbert(n-1, s, u)
        t.rt(u)
        t.fd(s)
        hilbert(n-1, s, -u)
        t.lt(u)
```