Rekurzia
je programovacia technika, ktorá umožňuje elegantne riešiť niektoré úlohy. Elegancia spočíva najmä v jednoduchosti zápisu algoritmu. Rekurziu možno použiť, ak riešenie nejakého problému vedie k riešeniu analogických problémov smerujúcich až k triviálnemu problému.

Úloha R.1
Vytvorte program na výpočet n-faktoriálu pre $n \in Z$ a zároveň $n \geq 0$.

Analýza
Rekurentná definícia výpočtu n-faktoriálu pre $n \geq 0$ hovorí:

0! = 1		výpočet 0! - triviálny problém
n! = n * (n- 1)!	pre n > 0	výpočet n! vedie k analogickému ale jednoduchšiemu problému, výpočtu (n-1)!

V zmysle rekurentnej definície možno napr. 5! vypočítať ako 5*4! = 5*(4*3!) = 5*(4*(3*2!)) = 5*(4*(3*(2*1!))) = 5*(4*(3*(2*(1*0!)))) = 5*(4*(3*(2*(1*1)))) = 120 (zátvorky naznačujú, ako sa uskutoční výpočet)

„Elementárne“ riešenie s použitím lokálnej premennej `vysledok`:

```
def fakt(n):
    if n == 0:
        vysledok = 1                    # nerekurzívna vetva, ukončenie rekurzie
    else:
        vysledok = n*fakt(n-1)          # rekurzívna vetva, volanie funkcie fakt()
    return vysledok
```

Riešenie bez pomocnej premennej `vysledok`:

```
import vratcislo

def fakt(n):
    if n == 0:
        return 1
    else:
        return n*fakt(n-1)

def main():
    n = vratcislo.nezaporneCele("Zadaj n: ")          # použitie modulu vratcislo
    print("{}! = {}".format(n, fakt(n)))

main()
```

Pri použití rekurzie budeme dodržiavať nasledujúce pravidlá:
1. Musí byť určený koniec výpočtu - situácia, keď je problém už tak zjednodušený, že má triviálne („bezvýpočtové“) riešenie. Rekurzia už ďalej nepokračuje. V prípade faktoriálu to je skutočnosť, že 0! = 1.
2. V každom kroku rekurzie musí dôjsť k zjednodušeniu problému. U faktoriálu to je vykonanie jednoduchého úkonu - násobenia a predovšetkým zmenšenie parametra výpočtu o jednotku. Parameter n postupne nadobúda hodnoty smerujúce k triviálnemu riešeniu, pre n>=3: n, n-1, n-2, n-3,..., 1, 0.
3. Program musí obsahovať test, či nenastala koncová (triviálna) situácia. Pre faktoriál to je test rovnosti n == 0. Len ak nenastala triviálna situácia, vykoná sa rekurzívna vetva.

Úloha R.2
Vytvorte program na výpočet mocniny x^n pre $x \in R-\{0\}$ a $n \in N_0$.

Analýza
Rekurentná definícia výpočtu mocniny x^n pre $x \in R-\{0\}$ a $n \in N_0$:

$x^0 = 1$	pre $x \neq 0$	výpočet x^0 - triviálny problém
$x^n = x * x^{n-1}$	pre n > 0	výpočet x^n vedie k analogickému problému, výpočtu x^{n-1}

Poznámka
0^0 nie je definované.

```
import vratcislo

def umocni(x,n):
    if n == 0:
        return 1
    else:
        return x * umocni(x,n-1)

def main():
    while True:
        x = vratcislo.realne("Základ mocniny: ")
        n = vratcislo.nezaporneCele("Exponent: ")
        if x == 0 and n == 0:
            print("Nula na nultú nedefinované!")
        else:
            break
    print("{} na {} = {}".format(x,n,umocni(x,n)))

main()
```

Pripomíname iteračné (iterácia – opakovanie) riešenia:

```
def umocni(x,n):
    vysledok = 1
    for i in range(n):
        vysledok *= x
    return vysledok
```

```
def fakt(n):
    vysledok = 1
    for i in range(n,0,-1):   # alebo (1,n+1):
        vysledok *= i
    return vysledok
```

Každá rekurzia obsahuje rekurzívnu vetvu, v ktorej je vlastne schovaný cyklus – opakované volanie procedúry, a nerekurzívnu vetvu, ktorá zabezpečuje ukončenie „cyklu“. Aktuálne hodnoty premenných pred vnorením do ďalšej funkcie sa odkladajú do zásobníka. **Rekurzívne riešenie úlohy, v porovnaní s iteračným, je vo všeobecnosti časom výpočtu aj použitou pamäťou náročnejšie!** Rekurzia sa používa len pri zložitejších úlohách. Uvedené jednoduché úlohy sú vhodné na pochopenie rekurzie, neodporúčame ich však v praxi používať.
Ak rekurzívna funkcia volá samu seba ako posledný príkaz (rekurzívny výpočet faktoriálu aj mocniny), hovorí sa aj **o chvostovej rekurzii**.

Úloha R.3
Napíšte rekurzívnu procedúru na nájdenie najväčšieho spoločného deliteľa dvoch prirodzených čísel, ak viete, že:

NSD (a, 0) = a resp. NSD (0, b) = b

pre a ≠ 0 a zároveň b ≠ 0 NSD (a,b) = { NSD (a mod b, b) ak a > b ; alebo ; NSD (a, b mod a) ak b > a

Riešenie
Naprogramujte si algoritmus tak, ako je uvedený v zadaní. Nižšie ďalšia alternatíva.

```
import vratcislo

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

def main():
    a = vratcislo.cele("Zadaj celé číslo: ")
    b = vratcislo.cele("Zadaj celé číslo: ")
    print("NSD({},{}) = {}".format(a,b, nsd(a,b)))

main()
```

Ďalšou možnosťou je využitie algoritmu

ak a = b NSD (a, b) = a alebo b

pre a ≠ b NSD (a,b) = { NSD (a - b, b) ak a > b ; NSD (a, b - a) ak b > a }

Úloha R.4

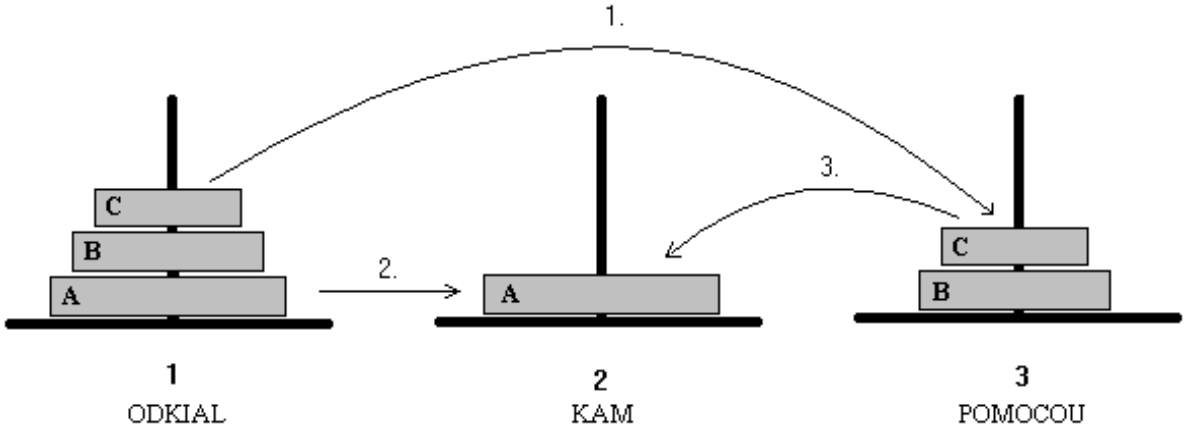
Pravdepodobne najznámejším problémom riešeným štandardne rekurziou sú tzv. Hanojské veže. Legenda hovorí, že v starobylom kláštore ukrytom v ďalekom kúte Ázie stoja tri zlaté kolíky. Na prvom z nich je nasunutých 64 diskov rôznych veľkostí, a to tak, že najväčší leží dole, nad ním leží o niečo menší, nad ním ešte menší atď. Úlohou mníchov je premiestniť túto vežu na druhý kolík, pričom musia dodržať tieto pravidlá:

- naraz sa môže premiestniť iba jeden disk
- väčší disk sa nikdy nesmie položiť na menší
- ktorýkoľvek z troch kolíkov možno použiť ako „odkladací“ na dočasné umiestnenie disku

Podľa legendy svet zanikne vo chvíli, keď mnísi svoju úlohu splnia.

Pozri: https://cs.wikipedia.org/wiki/Hanojsk%C3%A9_v%C4%9B%C5%BEe

Obrázok sme nakreslili pre tri disky (A, B, C). Stojany sú označené 1, 2 a 3. Pre označenie kolíkov sme zvolili premenné ODKIAL, KAM a POM (POMOCOU).



Myšlienkový postup pre N diskov je nasledovný:

Aby sme najväčší disk (v obrázku označený A) mohli presunúť z kolíka 1 na kolík 2, musíme:

- najprv presunúť nad ním ležiace disky, t.j. vežu zloženú z N-1 diskov, na kolík 3 pomocou kolíka 2 – analogický problém, ako presunúť N diskov
- potom môžeme disk A premiestniť z kolíka 1 na kolík 2 a
- nakoniec presunúť vežu so zvyšnými N-1 diskami z kolíka 3 na kolík 2 pomocou kolíka 1.

```
def prenesVezu(pocet, odkial, kam, pom):
    if pocet > 0:
        prenesVezu(pocet-1, odkial, pom, kam)
        prenesDisk(odkial, kam)
        prenesVezu(pocet-1, pom, kam, odkial)

def prenesDisk(z,na):
    print(z," -> ",na)

prenesVezu(3,1,2,3)
```

Jednoduchosť procedúry PrenesVezu prekvapí asi každého. Úspešnosť rekurzie spočíva v tom, že ak počítač nevie niečo vyhodnotiť, odloží to do zásobníka. Tak počítač odkladá PrenesVezu pre čoraz menej diskov, aktualizuje ODKIAL, KAM, POM – uvedomte si, ktoré parametre sú formálne a ktoré skutočné(!), až niet čo preniesť a pokračuje ďalším príkazom PrenesDisk atď.

Určte rekurzívnu a nerekurzívnu vetvu procedúry PrenesVezu.

Úloha R.5

Vytvorte rekurzívnu funkciu na výpočet kombinačného čísla $\binom{n}{k}$ ($n \geq 0$, $k \geq 0$, $n \geq k$, n, k prirodzené čísla) – koeficientu Pascalovho trojuholníka.

Analýza

Vzorec na výpočet kombinačného čísla $\binom{n}{k}$ = n!/((n-k)!*k!)

Rekurentná definícia:

pre k = 0 alebo k = n sa $\binom{n}{k}$ = 1 inak

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

Riešenie aj s kontrolou:

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

while True:
    n = int(input("Prirodzené číslo n: "))
    k = int(input("Prirodzené číslo k: "))
    if n >= k:
        break
    else:
        print("n < k - nedovolené!")

def fakt(n):
    if n == 0:
        return 1
    else:
        return n*fakt(n-1)

def kombCislo(n,k):
    return fakt(n)//fakt(n-k)//fakt(k)

print("{} nad {} = {}".format( n, k, NnadK(n,k) ))
print("Kontrola:", kombCislo(n,k))
```

Úloha R.6*

Napíšte nerekurzívnu a rekurzívnu funkciu na výpočet členov Fibonacciho postupnosti po zadaný index ≥ 0, ak viete, že:

$F_0 = 0$ a $F_1 = 1$

$F_i = F_{i-1} + F_{i-2}$ pre $i > 1$

Členy Fib. postupnosti: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,...

Variácie na tému výpočet členov Fibonacciho postupnosti (o týchto funkciách sa pobavíme pri študijnom texte Správnosť a výpočtová zložitosť algoritmu a programu):

```
po_index = int(input("Fibonacciho čísla po index: "))

# ITERÁCIA
# najefektívnejší výpočet
def fib_iteracne_for(po_index):
    prva_hodnota, druha_hodnota = 0,1          # použité len dve premenné!
    for inx in range(po_index):
        print(prva_hodnota, end=", ")
        prva_hodnota, druha_hodnota  = druha_hodnota, prva_hodnota + druha_hodnota
    print(prva_hodnota)

print("\nNajefektívnejší výpočet pomocou iterácie:")
fib_iteracne_for(po_index)
```

```
# použitie while
def fib_iteracne_while(po_index):
    inx = 0
    prva_hodnota = 0
    druha_hodnota = 1
    while inx <= po_index:
        if inx <= 1:
            nasledujuca = inx
        else:
            nasledujuca = prva_hodnota + druha_hodnota
            prva_hodnota = druha_hodnota
            druha_hodnota = nasledujuca
        print(nasledujuca, end=", ")
        inx += 1
''' odstránenie čiarky za posledným členom pre while inx < po_index:
    if inx != 0:
        print(prva_hodnota + druha_hodnota)
    else:
        print(inx)
'''
fib_iteracne_while(po_index)

# využitie zoznamu (pre po_index >= 1)
def fib_v_zozname(po_index):
    fib = [0,1]
    for i in range(2,po_index+1):
        fib.append(fib[-1] + fib[-2])
    return fib

print("\nIteračný výpočet s ukladaním do zoznamu:")
zoznam = fib_v_zozname(po_index)
print(zoznam)
print("Ten istý zoznam vypísaný ako reťazec:")
retazec = ', '.join( str(clen) for clen in zoznam )
print(retazec)

# REKURZIA
def fib_rek_def(index):
    if index == 0:
        return 0
    elif index == 1:
        return 1
    else:
        return fibdef(index-1)+fibdef(index-2)

def fib_rek(index):
    if index < 2:
        return index
    else:
        return fib_rek(index-1)+fib_rek(index-2)

print("\nRekurzívny výpočet:")
for inx in range(po_index):
    print(fib_rek(inx), end=", ")
print(fib_rek(po_index))

# zefektívnenie rekurzie použitím dict
def fib_pom_slovnika(index, index_hodnota={}):
    if index in index_hodnota:        # najpravdepod.udalosť, hodnota už bola počítaná - je v slovníku
        return index_hodnota[index]
    elif index > 1:                   # vypočítanie novej hodnoty a vloženie do slovníka (rek.vetva)
        index_hodnota[index] = fib_pom_slovnika(index-1) + fib_pom_slovnika(index-2)
        return index_hodnota[index]
    return index                      # pre index == 0 alebo index == 1 (nerekurzívna vetva)
```

```
print("\nRekurzívny výpočet s využitím slovníka (dict):")
#print("Hodnota na indexe {} je {}".format(po_index, fib_pom_slovnika(po_index)))
for fib_inx in range(0,po_index):
    print(fib_pom_slovnika(fib_inx), end=", ")
print(fib_pom_slovnika(po_index))
```

Výpis po spustení „variácií“:

```
Fibonacciho čísla po index: 19

Najefektívnejší výpočet pomocou iterácie:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181,
Iteračný výpočet s ukladaním do zoznamu:
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181]
Ten istý zoznam vypísaný ako reťazec:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181

Rekurzívny výpočet:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181

Rekurzívny výpočet s využitím slovníka (dict):
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181
```

Úloha R.7

Vytvorte rekurzívnu funkciu na prevod prirodzeného čísla z desiatkovej do dvojkovej sústavy. Výsledkom nech je reťazec obsahujúci znaky čísla dvojkovej sústavy.

Analýza

Algoritmus (Informatika pre stredné školy, učebnica pre 1.ročník, Ivan Kalaš a kolektív, SPN, str.34):

1. Desiatkové číslo vydelíme 2.
2. Zapíšeme zvyšok (0 alebo 1).
3. Výsledok delenia opäť vydelíme 2.
4. Zvyšok zapíšeme pred predchádzajúci zvyšok.
5. Opakujeme 3. a 4. krok tak dlho, kým výsledok delenia nie je 0.

Riešenie

```
def preved10na2(cislo10):
    if cislo10 == 0:
        return ""
    else:
        return preved10na2(cislo10 // 2) + str(cislo10 % 2)

cislo10 = int(input("Previesť ezáporné číslo s desiatkovej do dvojkovej sústavy: "))
if cislo10 == 0:
    print("0 = 0")
else:
    print("{} = {}".format(cislo10, preved10na2(cislo10)))
print("Kontrola:", bin(cislo10))
```

Úloha R.8*

Vytvorte rekurzívnu procedúru na rozklad prirodzeného čísla na súčin prvočísel (prvočiniteľov); prirodzené číslo = $c_1.c_2. \ldots .c_n$, kde c_i je prvočíslo pre i = 1 až n; napr. 84 = 2 * 2 * 3 * 7.

Riešenie študenta Samuela Novelinku:

```
import math

def rozloz(cislo):
    for delitel in range(2,math.ceil((math.sqrt(cislo)))):
        if(cislo%delitel == 0):
            return(str(delitel)+" * "+rozloz(cislo//delitel))
    return str(cislo)
```

```
rozlozit = int(input("Rozložiť na súčin prvočísel číslo: "))
print(rozlozit," = ",rozloz(rozlozit))
```

Úloha R.9*
Vytvorte a použite rekurzívnu funkciu, ktorá vráti index prvého výskytu zľava hľadaného prvku v poli; ak sa prvok v poli nevyskytuje, vráti -1.

Analýza

Úloha v podstate vyžaduje rekurzívne naprogramovanie lineárneho vyhľadávania v poli.

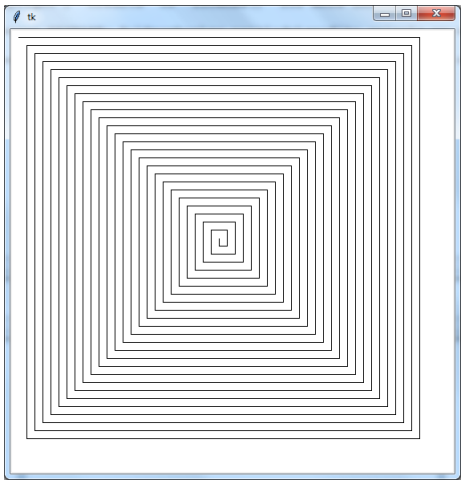
Riešenie

```
import pole

def hladaj(lavy_index):
    if lavy_index == len(pole):
        return -1
    else:
        if pole[lavy_index] == hladat:
            return lavy_index
        else:
            return hladaj(lavy_index+1)

pole = pole.vytvorIntNahodne(1,10)
hladat = int(input("Hľadať hodnotu: "))
inx = hladaj(0)
if inx > -1:
    print("Index prvého výskytu: ",inx)
else:
    print("Hodnota sa v poli nevyskytuje!")
```

Poznámka
Aj binárne vyhľadávanie možno napísať cez rekurzívnu funkciu.

Na záver dva príklady na rekurziu aj z grafiky. Podstatou takýchto úloh je naprogramovanie „základného“ obrazca, po nakreslení ktorého pokračuje rekurzia nakreslením znova „toho istého“ obrazca ale so zmenenými parametrami. `tkinter` je grafická knižnica ktorá umožňuje vytvoriť grafické plátno (okno) a v ňom vykresliť napríklad lomenú čiaru so zadanými súradnicami.

Úloha R.10*
Vytvorte program, ktorý rekurzívne nakreslí, po zadaní medzery, špirálu podľa obrázka.



Analýza

Špirála sa skladá „zo štvorcov s posunutými stenami smerom dovnútra“. Každý nový „štvorec“ začína v bode, kde skončil predchádzajúci „štvorec“.

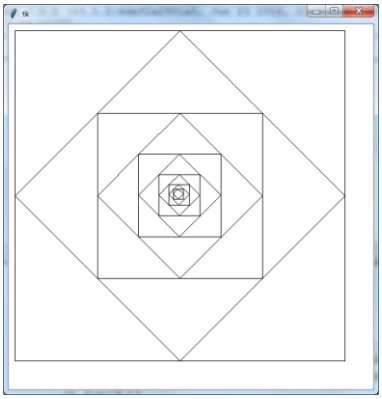
Riešenie

```
import tkinter

def spirala(dlzka, x, y):
    medzera = 10
    if dlzka>0:
        xd = x + dlzka
        yd = y + dlzka
        xm = x + medzera
        ym = y + medzera
        g.create_line(x, y, xd, y, xd, yd, xm, yd, xm, ym)
        spirala(dlzka-2*medzera, x+medzera, y+medzera);
```

```
g = tkinter.Canvas(width=550, height=550, bg="white")
g.pack()
spirala(500,10,10)
```

Úloha R.11*

Analyzujte, podľa priloženého riešenia, ako dôjde k vykresleniu obrázka vpravo.



Riešenie

```
import tkinter

def stvorce(pocet, dlzka, x, y):
    if pocet > 0:
        xd = x + dlzka
        yd = y + dlzka
        xd2 = x + dlzka//2
        yd2 = y + dlzka//2
        xd4 = x + dlzka//4
        yd4 = y + dlzka//4
        g.create_line(x,y, xd,y, xd,yd, x,yd, x,yd2, xd2,yd, xd,yd2, xd2,y, xd4,yd4)
        stvorce(pocet-1, dlzka//2, xd4, yd4);
        g.create_line(xd4,yd4, x,yd2, x,y)      # prečo je tu tento príkaz a čo robí?

g = tkinter.Canvas(width=550, height=550, bg="white")
g.pack()
stvorce(6,500,10,10)
```

NA ZÁVER

Rekurzia je veľmi elegantnou programovacou technikou, avšak pamäťovo aj časovo omnoho náročnejšou, ako je iteračné riešenie (použitie for-cyklu, while-cyklu) toho istého problému. Vysvetlili sme ju na jednoduchých úlohách (R.1 až R.9), ktoré by ste však, okrem Hanojských veží, nikdy nemali riešiť rekurziou. Rekurzia sa využíva pri riešení zložitejších problémov, ktoré sú rozobraté napríklad v knihe N.Wirtha Algoritmy a štruktúry údajov.