REKURZIA A BACKTRACKING

Mechanizmus volania funkcií:
• zapamätá sa návratová adresa
• vytvorí sa nový menný priestor funkcie
• v ňom sa vytvárajú lokálne premenné aj parametre
• po skončení vykonania tela funkcie sa zruší menný priestor
• vykonávanie programu sa vráti na návratovú adresu

Rekurzia v programovaní znamená, že funkcia najčastejšie zavolá samú seba, t.j. že funkcia je definovaná pomocou samej seba. Na prvej ukážke vidíme rekurzívnu funkciu, ktorá nerobí nič iné, len volá samú seba:

```
def xy():
   xy()

xy()

RuntimeError: maximum recursion depth exceeded
```

Takýto program veľmi rýchlo skončí chybovou správou.
To, že funkcia naozaj volala samú seba, môžeme vidieť, keď popri rekurzívnom volaní urobíme nejakú akciu, napr. vypisovanie nejakého počítadla:

```
def xy(p):
    print('volanie xy({})'.format(p))
    xy(p+1)

xy(0)
```

Treba si uvedomiť, že každé zvýšenie počítadla znamená rekurzívne volanie a teda vidíme, že ich bolo skoro tisíc. My už vieme, že každé volanie funkcie (bez ohľadu na to, či je rekurzívna alebo nie) spôsobí, že Python si niekde zapamätá nielen návratovú adresu, aby po skončení funkcie vedel, kam sa má vrátiť, **ale aj menný priestor tejto funkcie**. Python má na tieto účely rezervu okolo **1000 vnorených volaní**. Ak toto presiahneme, tak sa dozvieme správu „RuntimeError: maximum recursion depth exceeded“.

Chvostová rekurzia (nepravá rekurzia)

Aby sme nevytvárali nikdy nekončiace programy, t.j. nekonečnú rekurziu, niekde do tela rekurzívnej funkcie musíme vložiť test, ktorý zabezpečí, že v niektorých prípadoch rekurzia predsa len skončí. Najčastejšie to budeme riešiť tzv. triviálnym prípadom: na začiatok podprogramu umiestnime podmienený príkaz if, ktorý otestuje **triviálny prípad**, t.j. prípad, keď už nebudeme funkciu rekurzívne volať, ale vykonáme len nejaké „nerekurzívne“ príkazy.

Môžeme si to predstaviť aj takto: rekurzívna funkcia rieši nejaký komplexný problém a pri jeho riešení volá samu seba (rekurzívne volanie) väčšinou s nejakými pozmenenými údajmi. V niektorých prípadoch ale rekurzívne volanie na riešenie problému nepotrebujeme, ale vieme to vyriešiť „triviálne“ aj bez nej (riešenie takejto úlohy je už „triviálne“).
V takto riešených úlohách vidíme, že funkcia sa skladá z dvoch častí:
• pri splnení nejakej podmienky, sa vykonajú príkazy bez rekurzívneho volania (triviálny prípad),
• inak sa vykonajú príkazy, ktoré v sebe obsahujú rekurzívne volanie.
Zrejme, toto má šancu fungovať len vtedy, keď po nejakom čase naozaj nastane podmienka triviálneho prípadu, t.j. keď sa tak menia parametre rekuzívneho volania, že sa k triviálnemu prípadu nejako blížime. V nasledujúcej ukážke môžete vidieť, že rekurzívna špirála sa kreslí tak, že sa najprv nakreslí úsečka dĺžky d, korytnačka sa otočí o 60 stupňov vľavo a dokreslí sa špirála väčšej veľkosti. Toto celé skončí, keď už budeme chcieť nakresliť špirálu väčšiu ako 100 - takáto špirála sa už nenakreslí. Triviálnym prípadom je tu nič, t.j. žiadna akcia pre príliš veľké špirály:

```
#Rekurzia - spirala
def spir(d):
   if d > 100:
      pass # nerob nic
   else:
      t.fd(d)
      t.lt(60)
      spir(d+3)

import turtle
turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
spir(10)
```

Rekurziu môžeme používať nielen pri kreslení pomocou korytnačky, ale napr. aj pri výpise pomocou print(). V nasledujúcom príklade vypisujeme vedľa seba čísla n, n-1, n-2, ..., 2, 1:

```
#Rekurzia vypis cisel do n
def vypis(n):
   if n < 1:
      pass # nic nerob len skonci
   else:
      print(n, end=', ')
      vypis(n-1)

vypis(20)
```

Zrejme je veľmi jednoduché prepísať to bez použitia rekurzie, napr. pomocou while-cyklu. Poexperimentujme, a vymeňme dva riadky: vypisovanie print() s rekurzívnym volaním vypis(). Po spustení vidíme, že aj táto nová rekurzívna funkcia sa dá prepísať len pomocou

while-cyklu (resp. for-cyklu), ale jej činnosť už nemusí byť pre každého na prvý pohľad až tak jasná - odtrasujte túto zmenenú verziu:

```
def vypis(n):
    if n < 1:
        pass # nic nerob len skonci
    else:
        vypis(n-1)
        print(n, end=', ')

vypis(20)
```

Pravá rekurzia

Rekuzie, ktoré už nie sú obyčajné chvostové, sú na pochopenie trochu zložitejšie. Pozrime takéto kreslenie špirály:

```
#Rekurzia - spirala
def spir(d):
    if d > 100:
        pass # nerob nic
    else:
        t.fd(d)
        t.lt(60)
        spir(d+3)

import turtle
turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
spir(10)
```

Nejaké príkazy sú pred aj za rekurzívnym volaním. Aby sme to lepšie rozlíšili, triviálny prípad nastaví inú farbu pera.
Aj takéto rekurzívne volanie sa dá prepísať pomocou dvoch cyklov:

```
def spir(d):
    pocet = 0
    while d <= 100: # co sa deje pred rekurzívnym volaním
        t.fd(d)
        t.lt(60)
        d += 3
        pocet += 1
    t.pencolor('red') # triviálny prípad
    while pocet > 0: # co sa deje po vynáraní z rekurzie
        d -= 3
        t.fd(d)
```

```
        t.lt(60)
        pocet -= 1

import turtle
turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
spir(1)
```

Aj v ďalších príkladoch môžete vidieť pravú rekurziu. Napr. vylepšená funkcia vypis vypisuje postupnosť čísel:

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

vypis(20)
```

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

vypis(20)
```

Funkcia na otočenie reťazca:

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

print(otoc('Bratislava'))
#print(otoc('Bratislava'*100))
```

Táto funkcia pracuje na tomto princípe:
• krátky reťazec (prázdny alebo jednoznakový) sa otáča jednoducho: netreba robiť nič, lebo on je zároveň aj otočeným reťazcom

• dlhšie raťazce otáčame tak, že z neho najprv odtrhneme prvý znak, otočíme zvyšok reťazca (to je už kratší reťazec) a k nemu na koniec prilepíme odtrhnutý prvý znak
Toto funguje dobre, ale veľmi rýchlo narazíme na limity rekurzie: dlhší reťazec ako 1000 znakov už táto rekurzia nezvládne.

Táto funkcia už pracuje pre 1000-znakový reťazec správne, ale opäť nefunguje pre reťazce dlhšie ako 2000.
Ďalšie vylepšenie tohto algoritmu už nie je také zrejmé:
- reťazec rozdelíme na dve polovice (pritom jedna z nich môže byť o 1 krašia ako druhá)
- každú polovicu samostatne otočíme
- tieto dve otočené polovice opäť zlepíme dokopy, ale v opačnom poradí: najprv pôjde druhá polovica a za ňou prvá:

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

print(otoc('Bratislava'))
print(otoc('Bratislava'*100))
print(otoc('Bratislava'*200))
r = otoc('Bratislava'*100000)
print(len(r),r == ('Bratislava'*100000)[::-1])
```

Ďalšie rekurzívne obrázky

Napíšeme funkciu, ktorá nakreslí obrázok **stvorce** úrovne n, veľkosti a s týmito vlastnosťami:
• pre n = 0 nerobí nič
• pre n = 1 kreslí štvorec so stranou dĺžky a
• pre n > 1 kreslí štvorec, v ktorom v každom jeho rohu (smerom dnu) je opäť obrázok stvorce ale už zmenšený: úrovne n-1 a veľkosti a/3
Štvorce v každom rohu štvorca:

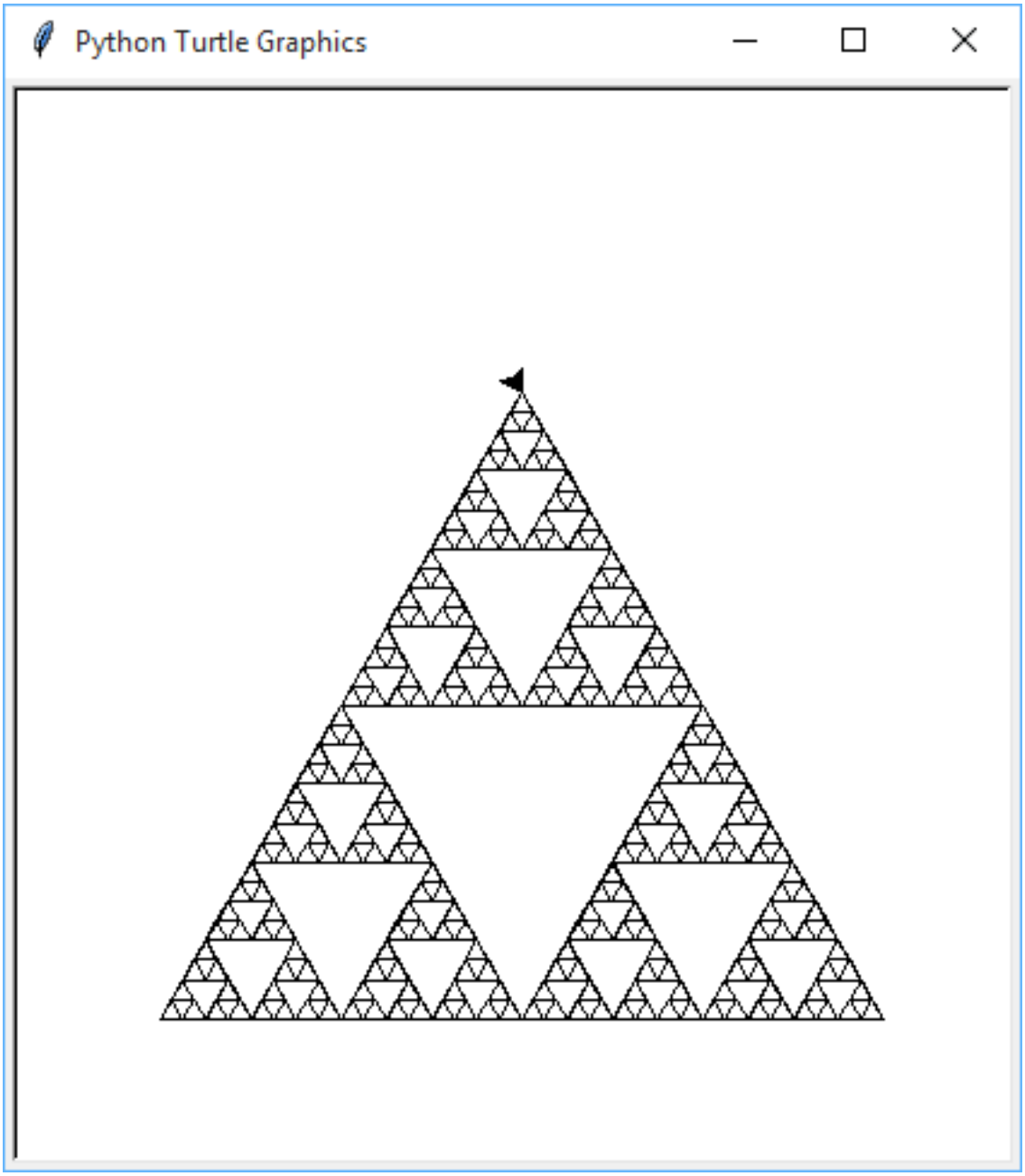
```
def stvorce(n, a):
    if n == 0:
        pass
    else:
        for i in range(4):
            t.fd(a)
            t.rt(90)
            stvorce(n-1, a/3) # skúste:stvorce(n-1, a*0.45)
```

```
import turtle
turtle.delay(0)
t = turtle.Turtle()
stvorce(4, 300)
```

Rekurzívny obrázok na rovnakom princípe ale **trojuholníkového tvaru** navrhol poľský matematik Sierpiňský (http://en.wikipedia.org/wiki/Sierpinski_triangle) ešte v roku 1915:

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


import turtle
turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
t.rt(60)
trojuholniky(6, 400)
```



Dračia krivka (http://en.wikipedia.org/wiki/Dragon_curve), ktorá sa skladá z dvoch „zrkadlových“ funkcií: ldrak a pdrak. Všimnite si zaujímavú vlastnosť týchto dvoch rekurzívnych funkcií: prvá rekurzívne volá samu seba ale aj druhú a druhá volá seba aj prvú. Našťastie Python toto zvláda veľmi dobre:

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

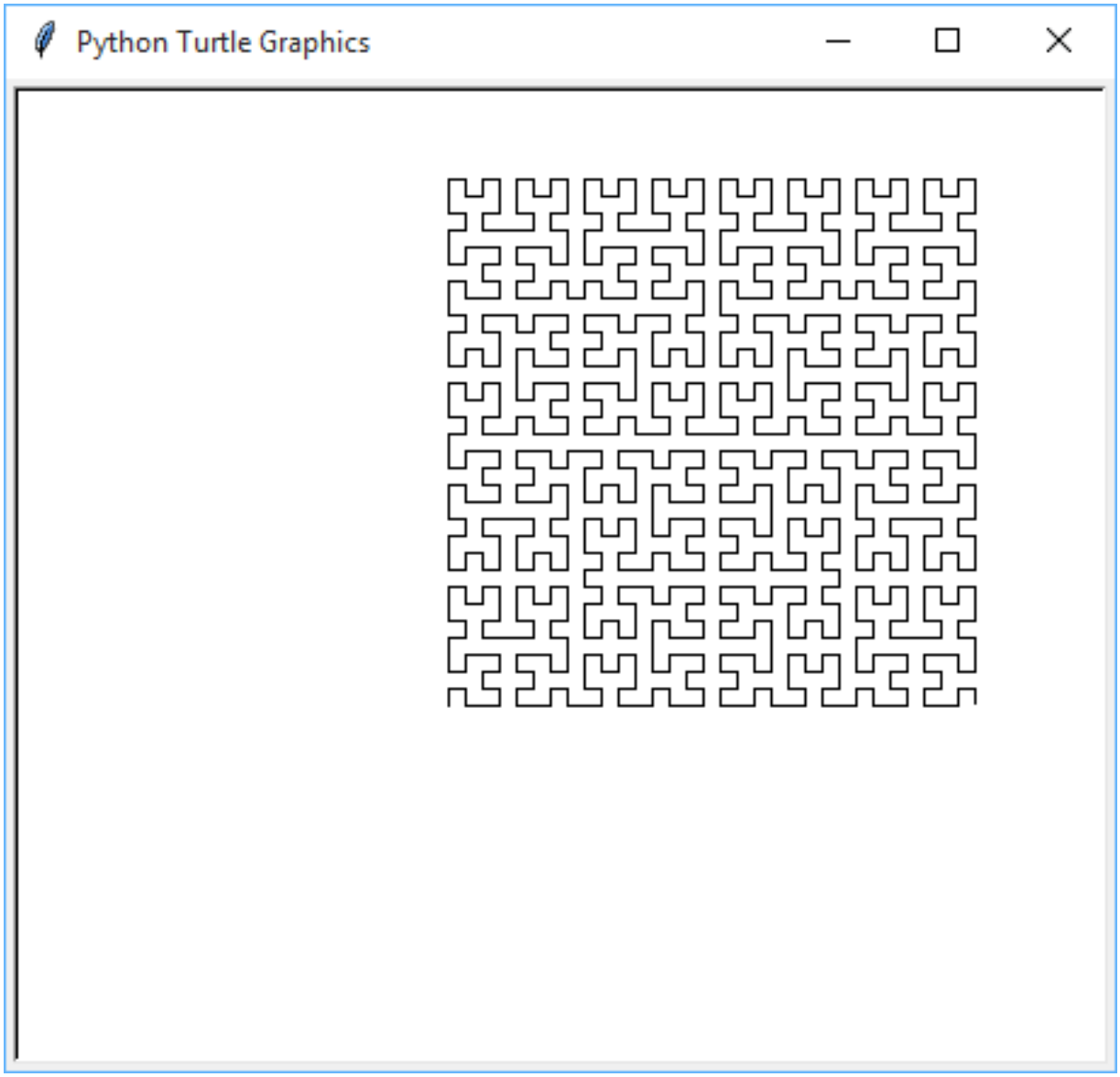
```
def pdrak(n, s):
    if n == 0:
        t.fd(s)
    else:
        ldrak(n-1, s)
        t.rt(90)
        pdrak(n-1, s)

import turtle
turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
t.ht()
ldrak(12, 6)
```

Hilbertova krivka (http://en.wikipedia.org/wiki/Hilbert_curve), ktorá sa tiež skladá z dvoch zrkadlových častí (ako dračia krivka) a preto ich definujeme jednou funkciou a parametrom u (t.j. uhol pre ľavú a pravú verziu):

```
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)

import turtle
turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
t.ht()
hilbert(5, 7)
```



Generovanie štvoríc čísel

Program, ktorý vypíše všetky 4-ciferné čísla zložené len z cifier 0 až n-1 a pritom sa žiadna cifra neopakuje viackrát. Takéto štvorice budeme generovať štyrmi vnorenými for-cyklami:

```
#Rekurzia - 4 cisla
def stvorice(n):
   pocet = 0

   for i in range(n):
      for j in range(n):
         for k in range(n):
            for l in range(n):
               if i!=j and i!=k and j!=k and i!=l and j!=l and k!=l:
```

```
                print(i, j, k, l)

                pocet += 1

    print('pocet =', pocet)

#Hlavny program
stvorice(4)
```

Program okrem všetkých vyhovujúcich štvoríc vypíše aj ich počet (zrejme pre n=4 ich bude 24, t.j. 4!).

Budeme riešiť generovanie n-tíc čísel pomocou rekurzívnej funkcie. Začnime s úlohou, v ktorej generujeme všetky štvorice čísel z intervalu 0 až n-1, pričom čísla sa nemôžu aj opakovať. Úlohu budeme riešiť pre ľubovoľné n.
n-ticu postupne vytvárame v n-prvkovom poli pole a keď je kompletná, tak ju vypíšeme. Využijeme na to množinu, ktorú vytvoríme zo všetkých n prvkov poľa. Ak je aj táto množina n-prvková, žiaden prvok v poli sa neopakoval a teda máme vyhovujúce riešenie:

```
n = 4
pole = [0]*n
def generuj(i):
    for j in range(n):
        pole[i] = j
        if i == n-1:
            if len(set(pole))==n:
                print(*pole)
        else:
            generuj(i+1)

generuj(0)
```

Program generuje 24 rôznych štvoríc.

Úloha generovania n-tíc riešená **cez objekty**. Namiesto globálnych premenných pracujeme s atribútmi triedy. Zapíšme rekurzívne generovanie n-tíc ako metódu triedy Uloha. Ukážeme riešenie, v ktorom sa vnárame do rekurzie len vtedy, keď doterajšia vygenerovaná časť riešenia vyhovuje podmienkam. Pripravili sme pomocnú funkciu moze(i,j), ktorá otestuje, či momentálna hodnota j môže byť priradená na i-tu pozíciu výsledného poľa:

```
class Uloha:
    def __init__(self, n):
        self.n = n
        self.pole = [0]*n
```

```
    def ries(self):
        self.pocet = 0
        self.generuj(0)
        print('pocet =', self.pocet)

    def moze(self, i, j): # ci môžeme na i-tu pozíciu dat j
        return j not in self.pole[:i]

    def generuj(self, i):
        for j in range(self.n):
            if self.moze(i, j):
                self.pole[i] = j
                if i == self.n-1:
                    print(*self.pole)
                    self.pocet += 1
                else:
                    self.generuj(i+1)

Uloha(4).ries()
```

Backtracking

Generovanie všetkých možných n-tíc, ktoré spĺňajú nejakú podmienku, je základom algoritmu, ktorému hovoríme prehľadávanie s návratom, resp. **backtracking**. Tento algoritmus teda:

- v každom kroku vyskúša všetky možnosti (napr. pomocou for-cyklu)
- pre každú možnosť preverí, či spĺňa podmienky (napr. metódou moze(...))
- ak áno, zaeviduje si túto hodnotu (hovoríme tomu, že sa zaznačí ťah) väčšinou v nejakých interných štruktúrach (pole, množina, asociatívne pole, ...)
- skontroluje, či riešenie nie je už kompletné a vtedy ho spracuje (napr. ho vypíše, alebo niekam zaznačí)
- ak riešenie ešte nie je kompletné, treba generovať ďalší prvok, čo zabezpečí rekurzívne volanie
- na konci cyklu, v ktorom sa postupne skúšajú všetky možnosti, treba ešte vrátiť stav pred zaevidovaním hodnoty
- (hovoríme tomu, že sa odznačí ťah)

Schématický zápis algoritmu:

```
def backtracking(param):
    if hotovo:
            vypis_riesenie()
    else:
```

```
        for i in vsetky_moznosti:
            if moze(i):
                zaznac_tah(i)
                backtracking(param1)
                odznac_tah(i)
```

Pomocou backtrackingu môžeme riešiť úlohy typu:
• matematické hlavolamy (8 dám na šachovnici, domček jedným ťahom, kôň na šachovnici, sudoku, ...)
• rôzne problémy na grafoch (nájsť cestu s najmenším ohodnotením z A do B, vyhodiť max. počet hrán, aby platila nejaká podmienka)
Vo všeobecnosti je **backtracking** veľmi **neefektívny algoritmus** tzv. **brute force** (hrubá sila), pomocou ktorého sa dá vyriešiť veľké množstvo úloh (postupne vyskúšam všetky možnosti). V praxi sa mnoho problémov dá vyriešiť oveľa efektívnejšie.
Pre backtracking väčšinou platí, že jeho zložitosť je exponenciálna, t.j. čím je úloha väčšieho rozsahu, tým
rýchlejšie rastie čas na jeho vyriešenie. Pri algoritmoch triedenia sme videli obrovský rozdiel vo výkone bublinkového
a rýchleho (quick-sort) triedenia. Pritom bublinkové triedenie má zložitosť rádovo n**2 a pre väčšie pole je už neprijateľne pomalé. Čo potom algoritmy, ktorých zložitosť je rádovo 2**n.

Problém 8 dám

Na šachovnicu s 8x8 treba umiestniť 8 dám tak, aby sa navzájom neohrozovali (vodorovne, zvislo ani uhlopriečne). Zrejme v každom riadku a tiež v každom stĺpci musí byť práve jedna dáma. Dámy očíslujeme číslami od 0 do 7 tak, že i-ta dáma sa bude nachádzať v i-tom riadku. Potom každé rozloženie dám na šachovnici môžeme reprezentovať osmicou čísel (v poli riesenie): i-te číslo potom určuje číslo stĺpca i-tej dámy.

```
class Damy:
  def __init__(self, n):
    self.n = n
    self.riesenie = [None]*n

  def ries(self):
    self.stlpec = set()
    self.u1 = set()
    self.u2 = set()
    self.pocet = 0
    self.hladaj(0)
    # print('pocet rieseni:', self.pocet)
    if self.pocet == 0:
      print('ziadne riesenie')

  def moze(self, i, j): # ci môže položit dámu na pozíciu (i,j)
    return j not in self.stlpec and i+j not in self.u1 and i-j not in self.u2
```

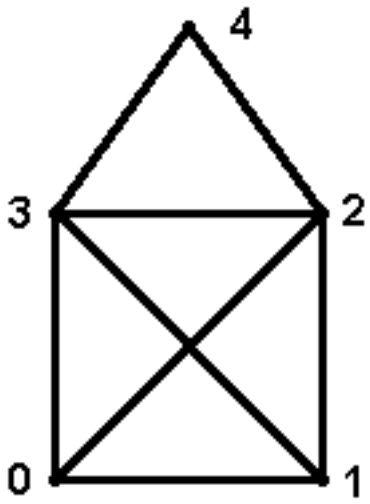
```
    def vypis(self):
        for k in range(self.n):
            r = ['.']*self.n
            r[self.riesenie[k]] = 'o'
            print(' '.join(r))
        print('='*2*self.n)

    def hladaj(self, i):
        if self.pocet:
            return 0
        for j in range(self.n):
            if self.moze(i, j):
                # zaznac položenie dámy
                self.riesenie[i] = j
                self.stlpec.add(j)
                self.u1.add(i+j)
                self.u2.add(i-j)
                if i == self.n-1:
                    self.vypis()
                    self.pocet += 1
                else:
                    self.hladaj(i+1)
                        # odznac polozenie damy
                self.riesenie[i] = None
                self.stlpec.remove(j)
                self.u1.remove(i+j)
                self.u2.remove(i-j)

Damy(8).ries()
```

Domček jedným ťahom

Úloha - potrebujeme zistiť, koľkými rôznymi spôsobmi sa dá nakresliť domček jedným ťahom. Pri kreslení môžeme po každej čiare prejsť len raz.



Obrázok domčeka je vlastne neorientovaný graf s 5 vrcholmi. Každé nájdené riešenie vypíšeme v tvare postupnosti vrcholov, cez ktoré sa prechádza pri kreslení domčeka. Keďže hrán je v grafe 8, tak táto postupnosť bude obsahovať presne 9 vrcholov: jeden štartový a 8 nasledovných vrcholov.
Graf budeme reprezentovať čo najjednoduchšie, napr. pomocou poľa množín susedností. Pre každý z vrcholov si treba pamätať množinu jeho susedov:

```
graf = [{1,2,3}, # susedia vrcholu 0
{0,2,3}, # susedia vrcholu 1
{0,1,3,4}, # susedia vrcholu 2
{0,1,2,4}, # susedia vrcholu 3
{2,3}, # susedia vrcholu 4
]
```

```
class Domcek:
  def __init__(self):
    self.g = [{1,2,3}, {0,2,3}, {0,1,3,4}, {0,1,2,4}, {2,3}]

  def ries(self):
    self.pocet = 0
    for i in range(len(self.g)): # postupne vyskúša pre všetky vrcholy
      self.riesenie = [i]
      self.hladaj()
    print('pocet rieseni:', self.pocet)

  def hladaj(self):
    v1 = self.riesenie[-1]
    for v2 in self.g[v1]:
```

```
        # zaznac tah
        self.riesenie.append(v2)
        self.g[v1].remove(v2)
        self.g[v2].remove(v1)
        if len(self.riesenie) == 9:
            print(*self.riesenie)
            self.pocet += 1
        else:
            self.hladaj()
        self.riesenie.pop()
        self.g[v1].add(v2)
        self.g[v2].add(v1)

Domcek().ries()
```

Prvý vrchol riešenia (štartový) sa do poľa priraďuje ešte pred zavolaním backtrackingu v štartovej metóde ries: keďže môžeme začínať z ľubovoľného vrcholu, v tejto metóde štartujeme backtracking v cykle postupne so všetkými možnými začiatočnými vrcholmi

• v backtrackingovej metóde hladaj:

– v1 obsahuje zatiaľ posledný vrchol riešenia, na ktorý bude teraz nadväzovať nasledovný vrchol v2

– zaznačenie ťahu uskutočníme tak, že práve prejdenú hranu dočasne z grafu odstránime (graf je neorientovaný preto treba odstraňovať oba smery tejto hrany)

– riešenie je kompletné vtedy, keď obsahuje presne 9 vrcholov

– odznačenie ťahu (teda návrat do pôvodného stavu pre zaznačením) urobíme vrátením dočasne odstránenej hrany