

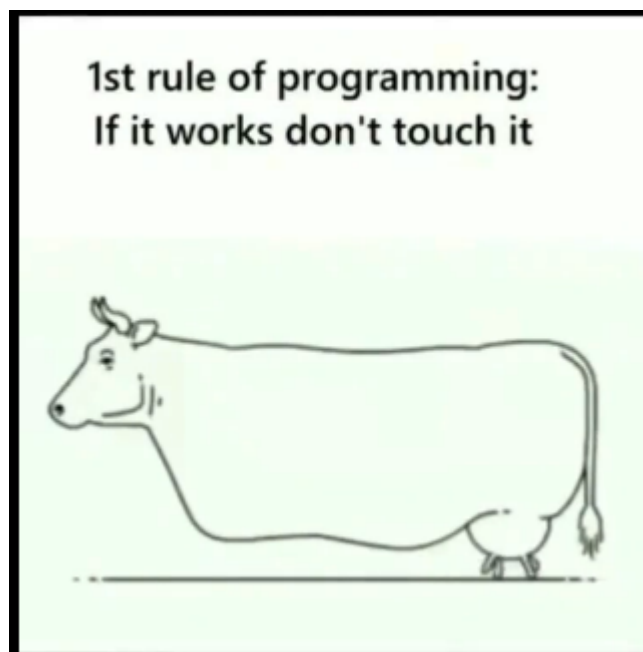
5. cvičení, 8-11-2023

Obsah:

- 0. Farní oznamy
- 1. Domácí úkoly
- 2. Opakování
- 3. Ještě medián: algoritmus s náročností $O(n)$
- 4. Třídění a binární vyhledávání
- 5. (možná) Pythonovské funkce

Farní oznamy

- 1. **Materiály k přednáškám** najdete v GitHub repozitáři <https://github.com/PKvasnick/Programovani-1>. Najdete tam také kód ke cvičením.
 - 2. **Domácí úkoly** Minulý týden jste dostali tři nové domácí úkoly. Už se k nim sešlo hodně řešení a zatím jediný problém vidím v tom, že možná byly příliš lehké.
Také jsme uzavřeli další trojici úloh z před-předchozího týdne. Řešení najdete na GitHubu.
-



- Opravování či vylepšování kódu může vyjít draho - v kódu můžou být ukryté jemnosti, které nejsou na první pohled patrné.
 - Na druhé straně, dokázat přechíst kód a vylepšit ho - přeorganizovat anebo zrychlit - je součástí práce dobrého programátora.
-

Kvíz (else v cyklu for)

```
for i in range(5):
    user_pwd = input().strip()
    if user_pwd == "top_secret_pwd":
        print("Logging in...")
        break
    else:
        print("Invalid password in 3 attempts. Quitting.")
```

`else` slouží typicky tam, kde v cyklu `for` opakujeme pokusy o provedení nějaké akce. Větev `else` pak slouží k obslužení neúspěchu.

Domácí úkoly

Několik poznámek:

Srozumitelnost kódu: Medián

Úloha o mediánu si vyžaduje setřídít seznam a vrátit buď prostřední hodnotu nebo průměr dvou prostředních hodnot, podle toho, zda je počet prvků v seznamu lichý anebo sudý:

$$\begin{aligned} Med(1, 2, \underline{3}, 4, 5) &= 3 \\ Med(1, \underline{2}, \underline{3}, 4) &= \frac{2+3}{2} = 2.5 \end{aligned} \quad (1)$$

Z některých odevzdaných řešení by ale člověk mohl nabýt dojmu, že vypočítat něco takového je docela náročný úkol:

```
...
nums = sorted(nums)
if len(nums)%2==0:

    print((nums[int((len(nums))/2)]+nums[int((len(nums))/2)-1])/2)
else:
    print(nums[int(len(nums)/2)])
```

nebo dokonce

```
...
list.sort()
if len(list)%2!=0:
    print(list[int((len(list)-1)/2)])
else:

print(float((list[int(len(list)/2)]+list[int((len(list)-2)/2)])/2))
```

Takovýto kód je nepřehledný. Nešlo by to napsat nějak srozumitelněji, aby bylo jasné na první pohled, že to je správně?

```
(5) 0 1 2 3 4 -> 2
(7) 0 1 2 3 4 5 6 -> 3
```

tedy pro lichý počet prvků máme `len(list)//2`. Pro sudý počet prvků

```
(6) 0 1 2 3 4 5 -> 2,3
(8) 0 1 2 3 4 5 6 7 -> 3,4
```

a pro sudý počet máme `len(list) // 2 - 1` a `len(list) // 2`, takže kód může vypadat nějak takto:

```
midpoint = len(list) // 2
if len(list) % 2 == 1:
    median = list[midpoint]
else:
    median = 0.5*(list[midpoint-1] + list[midpoint])
```

Na medián se dnes ještě jednou podíváme poněkud jinak.

Výplně

aneb proč se oplatí strávit chvíli s tužkou a papírem

Máme matici $n \times n$. Úloha je vyplnit ji spirálově směrem od levého horního rohu do středu čísla $1, 2, \dots, n^2$.

```
1  2  3  4
12 13 14 5
11 16 15 6
10 9  8  7
```

Plánujeme a kreslíme

Implementace: Matici umíme implementovat jako seznam seznamů. Například matici $n \times n$ vyplněnou čísly od 1 do n^2 můžeme vytvořit takto:

```
>>> n = 5
>>> matrix = [[i+n*j for i in range(n)] for j in range(n)]
>>> matrix
[[0, 1, 2, 3, 4], [5, 6, 7, 8, 9], [10, 11, 12, 13, 14], [15, 16,
17, 18, 19], [20, 21, 22, 23, 24]]
```

Maticí procházíme střídavě po řádcích, tedy po všech prvcích jediného podseznamu, a po sloupcích, tedy po stejném prvku všech podseznamů:

```
>>> for col in range(n):
    print(matrix[0][col], end = ' ')
```

```

0 1 2 3 4
>>> for row in range(n):
    print(matrix[row][n-1], end = ' ')

4 9 14 19 24
>>> for col in range(n-1,-1,-1):
    print(matrix[n-1][col], end = ' ')

24 23 22 21 20
>>> for row in range(n-1,-1,-1):
    print(matrix[row][0], end = ' ')

20 15 10 5 0

```

Poslední, co potřebujeme, je spirálovité procházení. Teď jsme procházeli řádky a sloupce 0 a `n-1`. Pro spirálovité vyplňování potřebujeme posouvat horní, pravý, spodní a levý okraj tak, že po přejdení ho vždy posuneme o 1. Zavedeme proměnné `top`, `right`, `bot`, `left`, a pak to už jen všechno sestavíme:

```

# vyplňujeme matici nxn spirálovitě z levého horního rohu
doprostředka čísla
# 1,2,...n*n.

n = int(input())

matrix = [[1 for _ in range(n)] for _ in range(n)] # _ protože
proměnné nepotřebujeme

left, right, top, bot = 0, len(matrix[0])-1, 0, len(matrix)-1
i = 1 # číslo, které vyplňujeme

while left <= right and top <= bot: # pokračujeme, dokud je kam
vyplňovat
    # Horní řádek zleva doprava
    for col in range(left, right+1):
        matrix[top][col] = i # vyplníme a inkrementujeme
        i += 1
    top += 1 # číslování je shora zleva

    for row in range(top, bot +1):

```

```

        matrix[row][right] = i
        i += 1
    right -= 1

    for col in range(right, left-1, -1):
        matrix[bot][col] = i
        i += 1
    bot -= 1

    for row in range(bot, top-1, -1):
        matrix[row][left] = i
        i += 1
    left += 1

for row in range(n):
    for col in range(n):
        print(f'{matrix[row][col]:3}', end = ' ')
    print('\n')

```

Opakování

Seznamy, řezy, metody

Index vrací položku, řez (slice) vrací seznam

```

P y t h o n
0 1 2 3 4 5 6
0 1 2 3 4 5

```

```

>>> s = [i for i in range(10)]
>>> s
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> s[3]
3
>>> s[3:4]
[3]
>>> s[::3]
[0, 3, 6, 9]
>>> s[10::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> s[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

```

Metody seznamů

- `list.append(x)` přidává položku na konec seznamu.
- `list.extend(iterable)` rozšíří seznam připojením všech prvků `iterable` na konec seznamu.
- `list.insert(i, x)` vloží položku `x` na pozici `i`.
- `list.remove(x)` odstraní ze seznamu první položku s hodnotou `x`.
- `list.pop([i])` odstraní položku na zadané pozici v seznamu anebo poslední položku, pokud index není zadán.
- `list.clear()` odstraní všechny položky ze seznamu.
- `list.index(x[, zacek[, konec]])` vrátí index (počítaný od 0) v seznamu, kde se nachází první položka s hodnotou rovnou `x`.
- `list.count(x)` určí, kolikrát se `x` nachází v seznamu.
- `list.sort(, klíč=None, reverse=False)` utřídí položky seznamu *na místě*.
- `list.reverse()` na místě obrátí pořadí prvků v seznamu.
- `list.copy()` vrací plytkou kopii seznamu.

Logický operátor `in`

Zjišťuje, zda se v iterovatelném objektu nachází daná hodnota.

```
list = [10, 20, 30, 40, 50 ];

if 10 in list:
    print("y se nachází v seznamu")
```

Upozornění: Pro seznam je to operace se složitostí $O(n)$ - seznam se musí prohledat celý.

Pomůcka - opakování

- `*[a, b, c, d]` rozbalí seznam na `a, b, c, d`.
 - `_` zastupuje proměnnou, kterou jinak nepoužíváme a nepotřebujeme definovat její jméno.
 - separátor v příkazu `print`: `print(1, 2, 3, 4, sep="x")` vytiskne `1x2x3x4\n`. Pro připomenutí, už jsme měli parametr `end`: `print(1, 2, 3, 4, end = "x")` vytiskne `1 2 3 4x`. Jestli neuvedete `end`, použije se znak konce řádku `\n`, pokud neuvedete `sep`, použije se mezera .
-

Ještě medián

- Vaše řešení obsahovala v nějakém kroku setřídění načtené posloupnosti, něco jako

```
seznam.sort
```


anebo

```
seznam = sorted(seznam)
```
- Je ale potřebné seznam setřídít? O hodnotě mediánu přece nerozhoduje pořadí největších a nejmenších hodnot, takže bychom potencionálně mohli ušetřit nějakou práci.

Příbuzná úloha: Najít v seznamu k -tou největší hodnotu. Pro posloupnost s lichým počtem n členů je medián `$n // 2$` -á největší (nebo nejmenší) hodnota, pro sudé n potřebujeme členy `$n//2 - 1$` a `$n//2$` , takže pokud vyřešíme tuto obecnější úlohu, bude její řešení použitelné i pro medián.

Rozděl a panuj Budeme postupovat tak, že vytvoříme sérii částečných uspořádání tak, abychom po každém kroku mohli eliminovat část posloupnosti, ve které se medián určitě nenachází.

1. Zvolíme si hodnotu - *pivot* - a rozdělíme posloupnost na tři podseznamy, *menší*, "pivoty" a "větší", s hodnotami menšími, rovnými, resp. většími nežli pivot.
2. Podle toho, do kterého seznamu připadne hledaný index *k*, skončíme anebo pokračujeme jenom s jedním z těchto podseznamů a vracíme se do kroku 1.
3. Pokračujeme, dokud nedospějeme k podseznamu délky 1. Hodnota, kterou obsahuje, je to, co hledáme.

Otázka je, jak pro daný seznam zvolit pivot. My použijeme náhodný pivot - prostě náhodně zvolíme za pivot některou hodnotu ze seznamu. To není optimální, ale funguje to docela dobře.

```
from random import randint

seznam = [randint(1,100) for _ in range(10)]
# while (hodnota := int(input())) != -1:
#     seznam.append(hodnota)

k = int(input())
print(f"{k=}")

assert(0 <= k - 1 < len(seznam))

low = 0
high = len(seznam)
while high - low > 1:
    pivot = seznam[randint(low, high - 1)]
    print(f"{low=} {high=} {pivot=} {seznam=}")
    low_numbers = [x for x in seznam[low:high] if x < pivot]
    pivots = [x for x in seznam[low:high] if x == pivot]
    high_numbers = [x for x in seznam[low:high] if x > pivot]
    seznam = seznam[:low] + low_numbers + pivots + high_numbers +
seznam[high:]
    mid1 = low + len(low_numbers)
    mid2 = mid1 + len(pivots)
    if k-1 < mid1:
        high = mid1
    elif k-1 >= mid2:
        low = mid2
    else:
        low = mid1
        high = mid2
        break
```

```

print(seznam)
print(seznam[low:high][0])

----- výstup -----
3
k=3
low=0 high=10 pivot=6 seznam=[7, 6, 5, 6, 7, 7, 4, 3, 3, 5]
low=0 high=5 pivot=5 seznam=[5, 4, 3, 3, 5, 6, 6, 7, 7, 7]
low=0 high=3 pivot=3 seznam=[4, 3, 3, 5, 5, 6, 6, 7, 7, 7]
[3, 3, 4, 5, 5, 6, 6, 7, 7, 7]
4

Process finished with exit code 0

```

Proč má toto náročnost $O(n)$? Když zanedbáme délku seznamu “pivoty”, máme

$$T(n) \approx n + \frac{n}{2} + \frac{n}{4} + \dots = 2n \quad (2)$$

Při náhodném výběru pivotu má algoritmus tuto náročnost pouze v průměru. Pro deterministický algoritmus potřebujeme nějak inteligentněji zvolit pivot. Typicky se pro tento účel používá medián mediánů: seskupíme data do skupin po pěti, a z každé vypočteme medián. Z těchto mediánů vypočteme medián, který slouží jako pivot, i když můžeme hierarchicky postupovat se seskupováním do pětic.

Binární vyhledávání a třídění

Vyhledávání v setříděném seznamu

To je to, co potřebují dělat funkce `index` a `count` - najít hodnotu v setříděném seznamu, nebo zjistit, jestli se tam nachází, nebo v kolikrát.

Algoritmus: Půlení intervalu (proto *binární*).

Náročnost: $\log(n)$

```

#!/usr/bin/env python3
# Binární vyhledávání v setříděném seznamu

```

```

kde = [11, 22, 33, 44, 55, 66, 77, 88]
co = int(input())

# Hledané číslo se nachází v intervalu [l, p]
l = 0
p = len(kde) - 1

while l <= p:
    stred = (l+p) // 2
    if kde[stred] == co:    # Našli jsme
        print("Hodnota ", co, " nalezena na pozici", stred)
        break
    elif kde[stred] < co:
        l = stred + 1      # Jdeme doprava
    else:
        p = stred - 1      # Jdeme doleva
else:
    print("Hledaná hodnota nenalezena.")

```

Aplikace: Řešení algebraických rovnic, minimalizace

Celočíselná druhá odmocnina

```

# Emulate math.isqrt

n = int(input())

l = 0
p = n # velkorysé počáteční meze

while l < p:
    m = int(0.5 * (l+p))
    # print(l, m, p)
    if m*m == n: # konec
        print(f"{n} is a perfect square of {m}") # format string
        break
    elif m*m < n:
        l = m
    else:
        p = m
    if p-l <= 1:

```

```
print(f"{n} is not pefect square, isqrt is {l}")
break
```

Úloha: Odmocnina reálného čísla

Řešení rovnice $\cos(x) = x$

```
# solve x = cos(x) by bisection
from math import pi, cos

l = 0.0
p = pi/2.0

while p - l > 1.0e-6: # Tolerance
    m = 0.5*(l + p)
    print(l, m, p)
    if m - cos(m) == 0:
        print(f"Found exact solution x = {m}")
        break
    elif m - cos(m) < 0:
        l = m
    else:
        p = m
else:
    e = 0.5 * (p-l)
    print(f"Converged to solution x = {m}+/-{e}")
```

"Bisection" je bezpečná, ale nikoli rychlá metoda hledání kořenů rovnice a minimalizace.

Třídění

Opakovaný výběr minima

Opakovaně vybíráme minimum a příslušnou hodnotu umístíme na začátek seznamu.

```
# Třídění opakovaným výběrem minima

x = [31, 41, 59, 26, 53, 58, 97]
```

```

n = len(x)
for i in range(n):
    pmin = i
    for j in range(i+1, n):
        if x[j] < x[pmin]:
            pmin = j
    x[i], x[pmin] = x[pmin], x[i]

print(x)

```

Bublinové třídění

Postupně "probubláváme" hodnoty směrem nahoru opakovaným srovnáváním se sousedy

```

# Třídění probubláváním

x = [31, 41, 59, 26, 53, 58, 97]

n = len(x)
for i in range(n-1):
    nswaps = 0
    for j in range(n-i-1):
        if x[j] > x[j+1]:
            x[j], x[j+1] = x[j+1], x[j]
            nswaps += 1
    if nswaps == 0:
        break

print(x)

```

Samozřejmě se brzo podíváme na efektivnější metody třídění.

Funkce

Pokud chceme izolovat určitou část kódu, například proto, že dělá dobře definovaný generalizovatelný úkol anebo úkol často používaný, používáme funkce. Funkce je jeden ze základních nástrojů pro organizaci a vytváření opakovaně použitelného kódu (Dalším jsou třídy).

```
def hafni/():  
    print("Haf!")  
  
hafni()  
hafni()
```

Funkce má jméno, pro které platí běžná pravidla pro vytváření identifikátorů. Kde to je vhodné, doporučuji používat rozkazovací způsob.

```
def hafni(n):  
    for i in range(n):  
        print("haf!")
```

`n` je tady parametr funkce. Do hodnoty `n` se při spuštění funkce překopíruje hodnota z volání funkce a platí tady všechny varování ohledně kopírování - o tom budeme vícekrát mluvit později.

Máme Python 3.9, takže modernější verze funkce bude vypadat takto:

```
def hafni(n:int): # Uvádíme očekávaný typ parametru  
    for _ in range(n): # Používáme nepojmenovanou proměnnou  
        print("haf!")
```

Uvedením typu parametru zabezpečíme, že interpret nás bude varovat, pokud použijeme parametr nesprávného typu. To někdy pomáhá, a jindy nám to zabraňuje psát generický kód.

Návratová hodnota a příkaz return

```
def plus(x,y):  
    return x+y  
  
print(plus(1,2))  
print(plus("Ne", "hafnu!"))
```

Příkaz `return výraz` ukončí vykonávání funkce a vrací jako hodnotu funkce `výraz`.

Nepovinné parametry

```
def hafni(krat:int = 1, zvuk:str = "Ha"):
    for _ in range(krat):
        print(zvuk)

hafni()
hafni(5)
hafni(zvuk = "Miau!")
hafni(krat = 5, zvuk = "kokrh!")
```

Viditelnost proměnných: lokální a globální jmenný prostor

```
zvuk = "kuku!"
kolik_hodin = 0

def zakukej():
    print(zvuk)
    kolik_hodin += 1
```

Proměnné `kolik_hodin` přiřazujeme, a Python ji musí uvnitř funkce zřídit. Implicitní předpoklad je, že chcete zřídit novou proměnnou. Pokud chcete použít proměnnou z globálního prostoru jmen, musíte to Pythonu říci.

```
zvuk = "kuku!"
kolik_hodin = 0

def zakukej():
    global kolik_hodin
    print(zvuk)
    kolik_hodin += 1
```

Mimochodem, tato funkce dělá něco, čemu se typicky chceme vyhnout: ovlivňuje proměnnou, která není jejím parametrem. Toto nazývá vedlejší efekt a je to nejčastěji symptom špatného programování.

Správná funkce by měla být čistá, tedy by měla vypočítat a odevzdat svou návratovou hodnotu bez toho, aby měnila hodnoty nějakých proměnných, včetně svých parametrů.

Příklady funkcí, které určitě nejsou čisté, jsme viděli: jsou to metody seznamu, které nějak přetvářejí seznam na místě: `sort`, `reverse`. Tyto funkce mění seznam, který je volá a nevracejí hodnotu. Je to proto, že jde spíše o metody třídy `list`, tedy funkce, které patří do nějaké vyšší datové struktury a operují nad ní.

Domácí úkoly

1. Najít v posloupnosti dat nejdelší souvislou rostoucí podposloupnost
2. Najít dělicí bod posloupnosti
3. Pyramidová výplň