Programování 2

3. cvičení, 04-03-2025

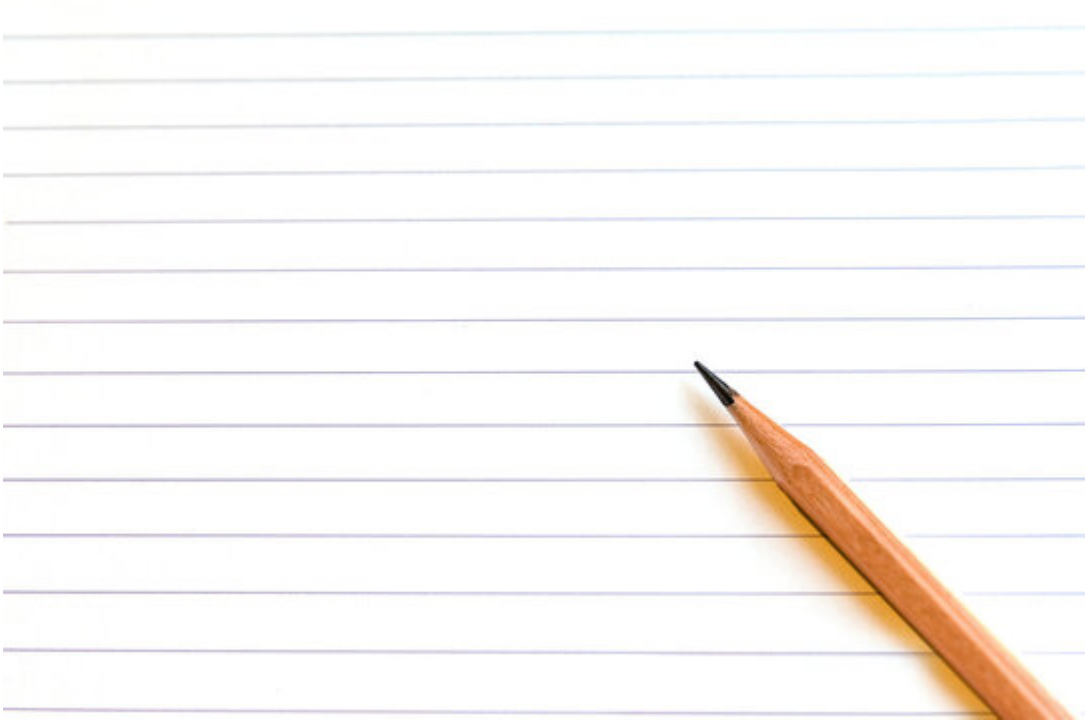
Farní oznamy

1. Tento text a kódy ke cvičení na1jdete v repozitáří cvičení na https://github.com/PKvasnick/Programovani-2.
2. **Domácí úkoly**:
    - Tento týden jste měli jeden lehký a jeden těžší úkol.
    - Jsem globálně spokojený s úrovní řešení.
    - Budeme se úkolům dnes věnovat
    - Pokud se vám zdá, že jsou pro vás domácí úkoly těžké, napište mi. Je zbytečné, abyste doplráceli na to, že jste v Programování 1 nedostali co jste měli.

Dnešní program:

- Kvíz
- Domácí úkoly
- BInární vyhledávání

Na zahřátí



> *"First, solve the problem. Then, write the code." – John Johnson*

Neboli **mějte po ruce tužku a papír**.

Toto se týká několika z vás. Pokud máte vážný problém s programováním, měli byste začít právě tímto.

Druhá důležitá ingredience je mít představu o tom, co v Pythonu lze udělat (metody datových typů, kolekce, jejich vlastnosti a použití, základní algoritmy a pod.) a co musíte naprogramovat "ručně".

Co dělá tento kód

Opakování z minulého cvičení.

```
dict(zip(range(1,4), ["Pascal", "Python", "C++", "Javascript"]))
```

zip a **enumerate**: Pokud to je možné, vyhýbáme se iterování přes indexy

```
for i in range(len(seznam)):   # NENENE!

for s in seznam:               # OK
for i, s in enumerate(seznam): # OK

for i in range(len(seznam1)):  # NE, jen pokud není vyhnutí
    print(seznam1[i] + seznam2[i])

for s1, s1 in zip(seznam1, seznam2): # OK
    print(s1 + s2)
```

- Méně hranatých závorek.
- Ochrana před modifikací dat CO TO ZNAMENÁ?

itertools.pairwise

`zip` a `enumerate` sú len dva príklady iterátorů, kterých existuje celá řada. Jeden další užitečný iterátor:

```
from itertools import pairwise

a = list(range(4))
for p, in in pairwise(a):
    print(p, n)

---
0 1
1 2
2 3
3 4
```

Pojmenované n-tice

```
from collections import namedtuple
Point = namedtuple("Point", "x y")
point = Point(5, 6)
point.x
Out[5]: 5
point.y
Out[6]: 6
point._asdict()
Out[7]: {'x': 1, 'y': 2}
```

Toto je jedna z možností, jak implementovat kontejner s různorodými daty, která spolu souvisí.

Trošku vadí nepřirozený konstruktor, který ale umožňuje konstruovat `namedtuple` za běhu.

Domácí úkoly

Všimli jste si, co měly domácí úkoly společné?

Minimální a maximální součet

To je docela jednoduchý úkol, přinejmenším proto, že umíme rychle najít slušné řešení: *stačí data setřídit* a vzít k nejmenších a k největších prvků (důkaz?8

```
a = []
while (x := int(input())) != -1:
    a.append(x)

a.sort()

print(sum(a[:k]), sum(a[-k]))
```

Toto je dobré řešení se složitostí $O(n \log n)$. Většinu času můžeme být s takovým řešením spokojení. Co když ale máme obrovské n? Není to tak nerealistické, takováto vyhledávací úloha je docela významná.

Pozorování:

Pro velké n by bylo výhodné, kdybychom měli jednoprůchodové řešení a nemuseli celou posloupnost načítat do paměti. (Ve vašem domácím úkolu jste posloupnost museli předem načíst, protože až pak se dovíte hodnotu k):

```
# Kdyby byla data rozumně uspořádaná, nejdřív bychom načetli k nebo k a n
k = int(input())

def read_from_console():
    while (x := float(input())) != -1:
        yield x

n = len(5)
```

```
smallest_k = [float("inf")] * k

for x in read_from_console():
    if x > smallest_k[0]:
        continue
    smallest_k[0] = x   # dáme na začátek a probubláme na správné místo8
    i = 1
    while smallest_k[i-1] < smallest_k[i] and i < k:
        smallest_k[i-1], smallest_k[i] = smallest_k[i], smallest_k[i-1]

print(sum(smallest_k)
```

Proč tady nemůžeme použít `itertools.pairwise`?

Tady jenom neustále třídíme k-tici, takže složitost bude $O(nk)$. Algoritmus můžeme vylepšit tak, že použijeme lepší způsob vyhledávání pozice pro vložení nové hodnoty a zvolíme datovou strukturu, do které lze levně vkládat nové hodnoty.

Pozorování:

Úplně zbytečně třídíme věci, které nás nezajímají:

- Hodnoty uprostřed
- Hodnoty v k-ticích.

Co s tím?

- jak situace závisí od velikosti k vzhledem k n?
- Příbuzná úloha: k-té největší číslo - budeme řešit podrobněji

Dělící bod

Všechno nalevo menší, všechno napravo větší. Jednoduché.

Řešení s načtením dat

Hledáme hodnotu, která je maximem hodnot s menším indexem, a minimem hodnot s větším indexem. Všechny levé maxima a pravá minima spočteme rekurzivně a pak zjistíme společný index.

```
import sys

def find_partition_point(sequence):
    n = len(sequence)

    if n == 0:
        return -1
    elif n == 1:
        return 0

    max_left = [None] * n
    min_right = [None] * n

    max_left[0] = sequence[0]
    for i in range(1, n):
```

```
        max_left[i] = max(max_left[i - 1], sequence[i])

    min_right[n - 1] = sequence[n - 1]
    for i in range(n - 2, -1, -1):
        min_right[i] = min(min_right[i + 1], sequence[i])

    for j in range(1, n - 1):
        if max_left[j - 1] <= sequence[j] <= min_right[j + 1]:
            return j

    return -1

sequence = []
for line in sys.stdin:
    num = float(line.strip())
    if num == -1:
        break
    sequence.append(num)

print(find_partition_point(sequence))
```

Průběžné řešení

Udržujeme seznam kandidátů na dělící bod:

- index hodnoty zařadíme do seznamu, pokud je největší dosud viděnou hodnotou. Toto pravidlo zaručuje, že kandidát bude levým maximem.
- ze seznamu odstraníme všechny kandidáty s hodnotou větší než aktuální hodnota. Toto pravidlo zaručuje, že v seznamu máme pravá minima.

Vkládání prvků a vyhledávání v setříděných datech

Klíčové slovo pro tuto úlohu je **binární vyhledávání**, protože v setříděném seznamu lze vyhledávat *fundamentálně* rychleji než v nesetříděném: $\log n$ vs. n, tedy 10 vs. 1000, 20 vs. 1 000 000. Proto používáme abecedně setříděné seznamy věcí, jmenoslovy, slovníky, telefonní seznamy atd.

Vyhledávání v setříděném seznamu

Úloha je najít hodnotu v setříděném seznamu, nebo zjistit, jestli se tam nachází, nebo kolikrát.

Podobně můžeme prohledávat interval na reálné ose, pokud definujeme "rozlišovací schopnost", tedy nejmenší interval, který ještě chceme prohledávat unvitř.

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 nachazí 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.")
```

Co kdybychom chtěli nalézt místo pro vložení nové hodnoty do seznamu?

To je podobná, i když mírně odlišná aplikace: nezajímá nás, zda hodnota existuje v seznamu, takže výsledek nemůže být neúspěšný: novou hodnotu můžeme vždycky vložit na správné místo. V aplikačním smyslu je toto fundamentálnější úloha.

```
# Vložení nové položky do setříděného seznamu

def insort(co: int, kam : list[int]) -> list[int]:
    '''
    Zařadí hodnotu co na správné místo v setříděném seznamu kam
    '''

    l = 0
    if co < kam[l]:
        return [co] + kam
    p = len(kam) - 1
    if co > kam[p]:
        return kam + [co]

    while p-l > 1:
        stred = (l + p) // 2
        strval = kam[stred]
        if strval > co:
            p = stred      # Jdeme doprava
        else:
            l = stred      # Jdeme doleva

    return kam[:p] + [co] + kam[p:]

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

print(insort(co, kam))
```

Co kdybychom chtěli nalézt všechny stejné hodnoty, nejen jednu?

Všimněte si jemných rozdílů v algoritmech:

1. Hodnota v `p` striktně menší než hledaná:

```
while p-l > 1:
        stred = (l + p) // 2
        strval = kam[stred]
        if strval > co:
            p = stred      # Jdeme doprava
        else:
            l = stred      # Jdeme doleva
```

2. Hodnota v `l` striktně menší než hledaná:

```
while p-l > 1:
        stred = (l + p) // 2
        strval = kam[stred]
        if strval >= co:
            p = stred      # Jdeme doprava
        else:
            l = stred      # Jdeme doleva
```

První algoritmus vrací pozici na konci série stejných hodnot, druhý na jejím začátku.

Binární vyhledávání v Pythonu: modul `bisect`

```
import bisect

nums = [1,2,3,3,3,4,5,8]

print(bisect.bisect_left(nums, 3))
print(bisect.bisect_right(nums,3))
print(nums[bisect.bisect_left(nums, 3):bisect.bisect_right(nums,3)])
---
2
5
[3, 3, 3]
```

Alternativní řešení: využití algoritmu pro spojování setříděných seznamů

Umíme velmi efektivně spojovat setříděné seznamy (v jediném průchodu, tedy v čase $O(n)$).

---

k-tý největší prvek

Průběžné řešení:

```python
from collections import deque
from bisect import insort

buffer = deque()
def update_buffer(val:int, limit = 1) -> None:
    global buffer
    insort(buffer, val)
    if len(buffer) > limit:
        buffer.popleft()
```

`insort` sice využívá binární vyhledávání, takže rychle najde, kam se má hodnota vložit, ale samotné vkládání je O(n).

Můžeme výrazně ušetřit, pokud se nejdřív zeptáme, zda nová hodnota není menší než levá hodnota buffru.

Alternativně lze udržovat bufr utříděný pomocí probublávání: posouváme hodnotu, dokud nenajdeme v bufru nějakou menší, a tam ji vložíme. Maximum pak máme vlevo.

Řešení s načtením dat: rekurzivní dělení intervalu

Začneme malými úlohami:

1. Jak rychle vypočíst medián 5 čísel?
2. Mám 25 čísel. Rozdělím je na pětice a z mediánů pětic spočtu medián. Jak blízko bude k mediánu všech 25 čísel?
3. Jak efektivně přeorganizovat seznam tak, aby hodnoty menší než p byly nalevo od p a větší než p napravo?

Domácí úkoly

1. **Hledání v setříděném poli** - zadání jasné, ale váš kód musí být opravdu rychlý, tato úloha nic neodpustí.
2. **Přesmyčky** neboli anagramy: jak rychle zjistit, že jedno slovo je permutací písmen jiného slova? Opravdu rychle?1