Programování 1 pro matematiky

6. cvičení, 04-11-2025

Obsah:

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** Dva ze tří úkolů, které jste dostali minulý týden, bývají vnímány jako těžší (nejdelší souvislá podposloupnost a dělící bod). Řešení přišlo tento týden významně méně, takže dáme několik rad.

> "Without requirements or design , programming is the art of adding bugs to an empty text file." - Louis Srygley

K tomuto mě inspirovala některá řešení domácích úkolů.

- Mějte na paměti specifikaci a řiďte se jí.
- Naučte se vědomě analyzovat problém, než začnete psát kód. Složitost problémů, které budete řešit, bude vzrůstat.
- Toto všechno platí i pro řešení, které vám pošeptá hlas z Internetu. Nemáte zakázáno zeptat se AI, ale máte odpovědnost za výsledek.

```
def med():                      # špatná funkce
    cisla = []
    while True:
        try:
            vstup = input().strip()
            if not vstup:       # maskuje špatná data
                continue
            try:
                hodnota = int(vstup)
            except ValueError:  # maskuje špatná data
                continue
            if hodnota == -1:
                break
```

```
            cisla.append(hodnota)
        except EOFError:        # maskuje špatná data
            break
        except KeyboardInterrupt:
            # brání uživateli ukončit program stiskem Ctrl-C
            break
            # Ctrl-C pouze ukončí vstup. Proč???
    if not cisla:               # maskuje špatná vstupní data
        return

    cisla.sort()
    n = len(cisla)
    if n % 2 == 1:
        median = cisla[n // 2]
    else:
        vyssie = n // 2
        nizsie= vyssie - 1
        median = (cisla[vyssie] + cisla[nizsie]) / 2.0
    print(median)

if __name__ == "__main__":
    med()
```

Tento program projde testy v ReCodExu, ale obsahuje hromadu chyb. Python je správně, funkce je správně, ale chování je v naprostém rozporu se zadáním.

Pokud čtete data od náhodného uživatele, je rozumné být velice defenzivní a obrnit se proti všemu, co se může na vstupu objevit.

Pokud máte jasně specifikované zadání, je to úplně jiná situace.

Kvíz

Co se vypíše? (inspirováno náklonností některých z vás k příkazu `pass`)

```
>>> x = ...
>>> if x is ...:
        print(f"{x=}")

???
```

Tři tečky běžně používáme tam, kde chceme naznačit, že jsme něco vynechali, a stejný význam mají v Pythonu:

```
>>> ...
Ellipsis

>>> Ellipsis
Ellipsis

>>> ... is Ellipsis
True
```

Typické použití:

```
def nedelej_nic():
    ...

nedelej_nic()
```

Když začínáte psát kostru programu, často potřebujete zavést funkce, které potřebuje hlavní kód, ale chcete je úplně napsat později.

I když je to úplně z jiného soudku, ještě si můžeme připomenout jeden objekt, a to podtržítko `_`. To používáme tam, kde má být proměnná, ale chceme naznačit, že tato proměnná nás nezajímá :

```
from random import randint
print([randint(1,10) for _ in range(10)])

_, x = divmod(20, 7)
```

Domácí úkoly

Od jednoduššího k složitějšímu

Dělící bod:

Když si sednete nad nový domácí úkol, mnohdy je dobré začít dělat jednoduché věci, například si definovat funkci, která zkontroluje, zda máme správné řešení. Takovou funkci napíšeme zpravidla lehce, vycházeje důsledně ze zadání:

```
def is_partition_point(p: int, data: list[int]) -> bool:
    left = (data[:p]==[]) or (max(data[:p]) < data[p])
    right = (data[p+1:] == []) or (min(data[p+1:]) > data[p])
    return left and right
```

Taková funkce, jakkoli je ošklivá, nám dává základní řešení: otestujeme pro každý prvek posloupnosti, zda je dělícím bodem. To dobře funguje, i když jen pro krátké posloupnosti: složitost takového algoritmu je $O(n^2)$, protože pro každou hodnotu opakovaně procházíme celou posloupnosti pro nalezení maxima a minima.

Stačí si ale uvědomit, že počítáme takovéto věci:

$$\max(a_0)$$
$$\max(a_0, a_1)$$
$$\max(a_0, a_1, a_2)$$
$$\dots$$

přičemž zjevně platí

$$\max(a_0, a_1, \dots, a_k, a_{k+1}) = \max(\max(a_0, a_1, \dots, a_k), a_{k+1})$$

a tedy můžeme všechny tyto veličiny spočíst v čase $O(n)$. Podobně to platí z druhé strany,

$$\min(a_{n-k-1}, a_{n-k}, \dots, a_{n-2}, a_{n-1}) = \min(a_{n-k-1}, \min a_{n-k}, \dots, a_{n-2}, a_{n-1})$$

Vidíme tedy, že najít dělící bod posloupnosti lze v čase $O(n)$. Protože na začátku potřebujeme spočítat kumulativní maxima a minima, potřebujeme celou posloupnost v paměti.

> **Note**
>
> AI mimochodem od letoška toto řešení zná. Pouze dva z vás použili řešení s kvadratickou složitostí a dostali radu a možnost nápravy, protože zjevně kód naprogramovali sami. Další dva použili řešení pouze s jedním polem kumulativních minim, které je docela nové.

Pro tuto úlohu ale existuje i jednoprůchodový algoritmus, a není tak složitý, jak by se na první pohled zdálo. Musíte propagovat seznam kandidátů a podle nové položky ho upravit:

- přidat hodnotu jako nového kandidáta, pokud je největší z těch, které jsme dosud viděli
- odstranit ty kandidáty, které jsou větší než nová hodnota.

Ještě jiné řešení: I když zadání požaduje řešení s lineární složitostí, co by se stalo, kdybychom data setřídili? Přesněji, kam by se dostal dělící bod?

Zůstane na místě.

Takže úlohu můžeme řešit i tak, že data setřídíme a hledáme "pevné body". Protože takovéto řešení má složitost jen nepatrně horší než lineární a vyznačuje se velkou elegancí, určitě bych vám ho uznal. -

> **Note**
>
> A AI ho nezná.

Binární vyhledáváni a třídění

Vyhledávání v setříděném seznamu

To je to, co potřebujeme použít jako náhradu generických metod `list.index` a `list.count` v setříděném seznamu. Typicky potřebujeme najít hodnotu v setříděném seznamu, nebo zjistit, jestli se tam nachází, nebo v kolikrát.

Algoritmus: Půlení intervalu (proto *binární*).

Náročnost: log(n)

```
# 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.")
```

> **Important**
>
> Binární vyhledávání je tak propastně rychlejší než "sekvenční" vyhledávání položku po položce, že použití generických metod na setříděný seznam je důkazem programátorské negramotnosti.

Příklad: Setříděný seznam s N = 1 000 000 položkami

- Sekvenční vyhledávání: průměrně 500 000 srovnání
- Binární vyhledávání: řádově $\log_2(1000000) \approx 20$ srovnání.

Python: modul `bisect`: vyhledávání v setříděném seznamu

- nalezení levé/pravé polohy pro vložení
- třídění s klíčem (uživatelskou funkcí položky)

```
# Binární vyhledávání v setříděném seznamu
from bisect import bisect_left

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

pos = bisect_left(kde, co)  # Hledá nejmenší index pro vložení
if kde[pos] == co:
    print(f"Hledaná hodnota {co} nalezena v poloze {pos}")
else:
    print("Hledaná hodnota nenalezena.")
```

Nalezení polohy pro vložení je v jistém smyslu fundamentálnější operace než vyhledání hodnoty, protože má univerzálnější využití a pro vyhledání udělá prakticky všechnu práci.

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

Úloha: Najděte řešení rovnice $2^x + x = 5$.

Úloha: Napište funkci, která bude počítat Lambertovu funkci W, tedy funkci, která je řešením rovnice $W(xe^x) = x$.

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 umísťujeme 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)
```

Technická poznámka: Tady nemůžeme použít funkci min, protože potřebujeme ne hodnotu minima, ale také jeho index. Na to bychom potřebovali nějakou takovouto funkci:

```
from random import shuffle  # jenom pro testovací data
from operator import itemgetter

def argmin(data):
    return min(enumerate(data), key=itemgetter(1))


data = list(range(10))
shuffle(data)
print(data)
min_index, min_value = argmin(data)
print(min_index, min_value)
```

Funkci `random.shuffle` potřebujeme jenom pro generování náhodné posloupnosti. `enumerate` je iterátor, který z posloupnosti `[1, 2, 3]` udělá posloupnost dvojic `(index, hodnota)`, tedy `[(0, 1), (1, 2), (2, 3)]`. V této posloupnosti pak vyhledáme minimum tak, že se díváme na druhou položku dvojic.

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)
```

Quicksort

Toto je jeden z algoritmů se složitostí O(n log n).

Princip: Zvolíme si "prostřední" hodnotu, *pivot*, a rozdělíme data na tři části: hodnoty menší než pivot, hodnoty rovné pivotu a hodnoty větší než pivot. Postup rekurzivně opakujeme pro nesetříděné části.

```
from random import randint

def quicksort(data):
    if len(data) < 1:
        return data
    if len(data) == 2:
        return  [min(data), max(data)]

    pivot = data[randint(0, len(data)-1)]
    left = [i for i in data if i < pivot]
    pivots = [i for i in data if i == pivot]
    right = [i for i in data if i > pivot]
    return quicksort(left) + pivots + quicksort(right)


data = [randint(1,100) for _ in range(20)]
print(data)
sdata = quicksort(data)
print(sdata)

---
[14, 38, 34, 75, 69, 84, 57, 69, 67, 64, 60, 66, 93, 3, 60, 67, 27, 45, 64, 36]
[3, 14, 27, 34, 36, 38, 45, 57, 60, 60, 64, 64, 66, 67, 67, 69, 69, 75, 84, 93]
```

Počet srovnání: V každém kroku přibližně n, kroků je přibližně $\log_2 n$.-

Třídění v Pythonu: `list.sort`

Python poskytuje zabudovanou metodu `sort` pro seznamy. Je to velice rychlé třídění, vhodné pro obecné použití.

Signatura funkce je `list.sort(reverse=True|False, key=myFunc)`

`key` je klíč, funkce, která pro každou hodnotu v seznamu vypočte veličinu, podle které se pak hodnoty setřídí. Typicky používáme

- **lambda funkce** (podrobněji probereme později) - malé nepojmenované funkce, např. `lambda x: x**2`
- `operator.itemgetter(index)` - funkce, která z argumentu vrátí položku s indexem `index`.

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 (Dalšími jsou třídy a moduly).

```
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 = "Haf"):
    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 vypočíst 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 ni.

Domácí úkoly

Minimální a maximální součet - v dané posloupnosti čísel najděte k-tici s největším a k-tici s nejmenším součtem.

Vyhledávání ve velkých datech - tady se od vás bude chtít kód, který bude rychle vyhledávat ve velkých datech. Nepolekejte se, pokud vaše řešení nesplní všechny testy, ale snažte se jich splnit co nejvíc.