Programování 2

2. cvičení, 27-2-2023

Farní oznamy

1. Tento text a kódy ke cvičení najdete v repozitáři cvičení na https://github.com/PKvasnick/Programovani-2.
2. **Domácí úkoly**:
    - Na první várku domácích úkolů přišlo docela dost řešení
    - Úloha o dělícím bodu bývá obvykle vnímána jako obtížná

Obsah:

- Kvíz
- Domácí úkoly:
    - Načítání posloupností
    - Kdy lze posloupnost zpracovat průběžně a jak to udělat?
- Resty z minula: sečítání čísel po číslicích

Na zahřátí

> Pokud to funguje, nedotýkej se toho.

Dobrá, nebo špatná rada?

Co dělá tento kód

```
jmena = {"Jana", "Pavel", "Pepa", "Franta"}
dalsi = {"Eva", "Pepa", "Katka", "Standa"}
jmena & dalsi
????
jmena and dalsi
jmena - dalsi
```

Seřaďte podle data narození

```
lide = [
    ("Jana", "Nováková", 1964),
    ("Kateřina", "Kocourová", 1962),
    ("Jozef", "Winkler", 1952),
    ("Petr", "Suchý", 1968),
    ("Jan", "Michal", 1951)
]
```

- `operator.itemgetter`
- lambda

Domácí úkoly

1. Načtení posloupnosti z konzole a její zpracování

Toto děláte u domácích úkolů běžně:

> Vstupní posloupnost načtete z konzoly číslo po čísel, každé číslo na novém řádku.
> Posloupnost je ukončená řádkou s −1, která nepatří do posloupnosti.

Generická verze:

```
def read_from_console() -> list[float]:
    a = []
    while True:
        line = input()
        if "-1" in line:
            break
        a.append(float(line))
    return a
```

Poznámky:

1. Kód `-> list[float]` oznamuje Pythonu, že výstupem funkce je seznam desetinných čísel. Podobně můžeme oznámit i typy parametrů, a analyzátor kódu ve vašem IDE pak zahlásí chybu, pokud někde použijeme nesprávný typ.
2. Logika: Máme nekonečný cyklus a vyskakujeme z něj, pokud narazíme na znak konce posloupnosti.
3. Testujeme `"-1" in line` namísto `line == "-1"` Proč? (Už vás někdy ReCodEx trýznil hláškami "End-of-file error" u testů?)

Ze souboru:

```
def read_from_file(filename: str) -> list[float]:
    a = []
    with open(filename, "r") as infile:
        for lines in infile:
            if "-1" in line:
                break
            a.append(float(line))
    return a
```

Načíst do paměti nebo nenačíst?

U některých úkolů dokážeme čísla zpracovávat postupně a nepotřebujeme mít celou posloupnost uloženou v paměti.

Příklad: : nalezení maximální hodnoty, výpočet průměru a standardní odchylky

V těchto případech má algoritmus formu propagování nějakého vnitřního stavu přes vstupná data: Začneme počátečním stavem (např. nejmenší možnou hodnotou typu `float`, tedy `float('-inf')`) a jak přicházejí data, aktualizujeme ji (např. se u každé vstupní hodnoty ptáme: Je toto největší hodnota, jakou jsem doposud viděl? a pokud je, nastavíme vnitřní stav na tuto hodnotu).

Na konci máme výsledný stav (například maximální hodnotu posloupnosti) a umíme dokázat, jaké má vlastnosti (například že je to opravdu maximum posloupnosti).

Naopak, u jiných úloh potřebujeme mít posloupnost v paměti celou (např. pro výpočet mediánu)

Jak to je pro dělící bod?

Načítání pomocí generátoru

Pokud chceme hledat například maximum posloupnosti a nechceme ji celou načítat, musíme kód pro načítání a hledání promíchat. To je nešťastné, pokud chceme pro zpracování posloupnosti použít stejný kód pro načítání ze standardního vstupu nebo souboru.

```
m = float("-inf")
while "-1" not in (line := input()):
    number = float(line)
    if number > m:
        m = number
```

Poznámky:

- Inicializace: `float("-inf")` je nejmenší číslo v plovoucí desetinné čárce. Jaké je nejmenší celé číslo?
- `:=` je "walrus", čili mroží operátor, pomocí kterého můžeme ukrást hodnotu, kterou načte `input()` do proměnné `line`. To je šikovné a řeší to malý problém s logikou načítacího cyklu: protože v cyklu načítáme, musíme kód zpřeházet, pokud chceme testovat v hlavičce `while` anebo testovat uvnitř cyklu a použít `break`

```
def read_from_console():
    while "-1" not in (line := input()):
        yield float(line)
    return

m = float("-inf")
for number in read_from_console():
    if number > m:
        m = number
print(m)
```

Napište kód, který takto nalezne maximum při načítání posloupnosti ze souboru.

Reduce

Uměli bychom uzavřít otevřený cyklus `while`, resp. `for`, který máme v tomto kódu?

Můžeme použít funkci `functools.reduce`, která dělá přibližně toto:

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = function(value, element)
    return value
```

Tedy funkce `reduce` propaguje a aktualizuje nějaký stav přes posloupnost.

```
from functools import reduce

def read_from_console():
    while "-1" not in (line := input()):
        yield float(line)
    return

maximum = reduce(max, read_from_console, float("-inf"))
print(maximum)
```

Takovýto kód bude rychlý, protože cyklus se vykonává uvnitř funkce, a tedy běží v C a ne v Pythonu.

Podobné úlohy

- rozhodnout, zda je posloupnost čísel monotonní a jak (konstantní, rostoucí, neklesající, klesající, nerostoucí)
- v posloupnosti čísel nalézt druhou největší hodnotu a počet jejích výskytů
- v posloupnosti čísel určit délku nejdelšího souvislého rostoucího úseku
- v posloupnosti čísel určit počet různých hodnot
- v posloupnosti čísel nalézt souvislý úsek se součtem K (pro zadanou hodnotu K)
- v posloupnosti kladných čísel nalézt souvislý úsek se součtem K (pro zadanou hodnotu K)
- v posloupnosti čísel nalézt souvislý úsek s maximálním součtem.

2. Úloha o dělícím bodu

Odkud se úloha vzala?

U třídícího algoritmu typu quicksort a u řady dalších úloh (například u algoritmu pro nalezení mediánu v lineárním čase) používáme částečné setřídění dat pomocí pivotu: Zvolíme si nějakou hodnotu z posloupnosti a zpřeházíme data tak, že menší hodnoty budou vlevo od pivotu a větší vpravu.

Úloha o dělícím bodu je pak úlohou nalézt pivot, když jsme zapomněli, která hodnota to je. Tudíž opravdu nemá valný praktický význam.

Analýza

Podle definice ze zadání hledáme index i v posloupnosti, pro který platí $max(a_{j|j<i}) < a_i < min(a_{j|j>i})$ Takže bychom měli dělat zhruba (až na okrajové případy) toto:

```
...
n = len(a)
for i in range(n):
    if max(a[:i]) < a[i] < min(a[i+1:]):
        print(i)
        break
```

Toto má ale složitost $O(n^2)$, a určitě to jde zlepšit, stačí si uvědomit, že počítáme takovéto věci:

$$\begin{gathered} \max(a_0) \\ \max(a_0, a_1) \\ \max(a_0, a_1, a_2) \\ \dots \end{gathered} \tag{1}$$

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 mít celou posloupnost v paměti.

Sekvenční řešení

Můžeme také zkusit sekvenční řešení - tedy řešení bez načtení celé posloupnosti do paměti. Stav, který potřebujeme aktualizovat v době načítání, je:

- **seznam kandidátů** - ten musíme propagovat až do konce načítání, protože nemůžeme prohlásit nějakou pozici v posloupnosti za dělící bod předtím, než jsme viděli celou posloupnost. Seznam proto, že cestou můžeme nalézt několik dělících bodů. I když vracet budeme pouze první, až na konci posloupnosti můžeme určit, který to bude (nebo že to nebude žádný). Posloupnost kandidátů bude rostoucí, následující dělící bod musí mít hodnotu větší než předchozí.

  ```
  - Do seznamu kandidátů zapíšeme načtenou hodnotu, pokud je větší než průběžné
  maximum.

  - Naopak ze seznamu odstraníme ty kandidáty, jejichž hodnota není menší než
  právě načtená hodnota.
  ```

- **průběžné maximum** - aktualizujeme obvyklým způsobem: je to největší hodnota, kterou jsme dosud viděli.

Vzorové řešení

Varianta s načtením posloupnosti

- pole s (levými) kumulativními maximy a (pravými) kumulativními minimy definujeme "s převisem", tedy `cum_max[i] = max(a[0], a[1], ... a[i-1])` a `cum_min[i] = min(a[i+1], a[i+2], ... a[n-1])`:

```
cum_max[0] = a[0]
cum_max[i] = max(cum_max[i-1], a[i-1])
cum_min[-1] = a[-1]
cum_min[i] = min(cum_min[i+1], a[i+1])
```

- Výhoda takovéhoto uspořádání je v tom, že se nám do vyhledávání nepletou okrajové případy. Abychom dostali jednoduchý kód, po vyřešení okrajových případů voláme `sys.exit()`.

```
import sys

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

n = len(a)

if n == 0:
    print(-1)
    sys.exit()

if n == 1:
    print(0)
    sys.exit()

cum_max = [a[0]] * n
cum_min = [a[-1]] * n

for i in range(1,n):
    cum_max[i] = max(cum_max[i-1], a[i-1])

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

if a[0] < cum_min[0]:
    print(0)
    sys.exit()

for i in range(1, n-2):
    if cum_max[i] < a[i] < cum_min[i]:
        print(i)
        sys.exit()

if a[-1] > cum_max[-1]:
    print(n-1)
    sys.exit()
```

```
print(-1)
```

Průběžná varianta

Protože si musíme pamatovat nejen hodnoty kandidátů ale i jejich pozici, bude seznam kandidátů tvořen dvojicemi (index, hodnota). Abychom si zjednodušili kód, do seznamu zapíšeme inicializační položku (-1, -1.0e-10) a maximum nastavíme také na -1.0e-10. Pro praktické účely by takováto inicializace měla postačovat, pokud hrozí, že naše posloupnost bude obsahovat obrovská záporná čísla, je potřeba inicializaci upravit.

Kód řešení je pak překvapivě jednoduchý:

```
MIN_VALUE = float('-inf')
candidates = [[-1, MIN_VALUE]]
maximum = MIN_VALUE

index = 0
while (n := int(input())) != -1:
    if n > maximum:
        maximum = n
        candidates.append([index, n])
    else:
        while candidates[-1][1] >= n:
            candidates.pop()
    index += 1

if len(candidates) == 1:
    print(-1)
else:
    print(candidates[1][0])
```

Obvyklé problémy

Tato úloha bývá vnímána jako těžká. Většina problémů pochází z nepochopení zadání. Je proto potřeba dobře si zadání přečíst, a pak začít s řešením, vycházejícím důsledně z definice. To bude $O(n^2)$, a alespoň máte něco v ruce, než začnete přemýšlet, jak řešení zefektivnit. Asi nejdůležitější je uvědomit si, že musíte vidět celou posloupnosti, než můžete o některé položce říct, že je dělícím bodem.

Z minula

Součet dvou čísel

Na vstupu načteme dvě celá čísla jako znakové řetězce. Na výstupu má váš kód vypsat jejich součet, ale máte povolenou sečítat pouze číslice 0-9, ne celá čísla. Můžete si představit, že pro sčítání máte k dispozici tabulku typu

	0	**1**	**...**	**8**	**9**
0	(0, 0)				
1	(1, 0)	(2, 0)			

	0	1	...	8	9
2	(2, 0)	(3, 0)			
...					
8	(8, 0)	(9, 0)		(6, 1)	
9	(9, 0)	(0, 1)		(7, 1)	(8, 1)

kde první číslo jsou jednotky součtu a druhé číslo je "přenos".

Poznámky

- Jak to naprogramovat, aby kód byl hezký a rozumně efektivní? ("uzavřené" cykly: namísto `for` a `while` atd.)
- Funkce `zip` a funkce `itertools.zip_longest`
- Oplatí se pro takováto čísla (uspořádané seznamy číslic) zavést třídu?
- Samozřejmě budeme chtít také odečítat, násobit a dělit. Umíme vytvořit nějaký sjednocující algoritmus?

Zkusíme jednoduché věci:

1. Načtení čísla

```
digits = [int(c) for c in input()]
```

2. Dvojice číslic pozpátku:

```
from itertools import zip_longest
```

Itertools.zip_longest()

This iterator falls under the category of Terminating Iterators. It prints the values of iterables alternatively in sequence. If one of the iterables is printed fully, the remaining values are filled by the values assigned to `fillvalue` parameter. **Syntax:**

```
zip_longest( iterable1, iterable2, fillval)
```

```
zip_longest(get_digits_reversed(digits1), get_digits_reversed(digits2),0)
```

Alternativa: protože výsledek může být o číslici delší než nejdelší z čísel, můžeme také čísla předem doplnit nulami na potřebnou délku a použít obyčejný `zip`:

```
delka = max(len(a0), len(b0)) + 1
a0 = a0.zfill(delka)
b0 = b0.zfill(delka)
```

Pozor, toto funguje pro znakové řetězce, ne pro seznamy číslic.

4. Sečítání s přechodem:

```
def add_and_carry(d1:int, d2:int) -> {int, int}:
    return divmod(d1+d2,10)
```

Na výstupu budeme mít dvojici (přechod, součet). Jak ale budeme takovéto dvojice sečítat v cyklu?

```
from itertools import accumulate
```

itertools.**accumulate**(*iterable*[, *func*, *, *initial=None*])

Make an iterator that returns accumulated sums, or accumulated results of other binary functions (specified via the optional *func* argument).

If *func* is supplied, it should be a function of two arguments. Elements of the input *iterable* may be any type that can be accepted as arguments to *func*. (For example, with the default operation of addition, elements may be any addable type including `Decimal` or `Fraction`.)

Usually, the number of elements output matches the input iterable. However, if the keyword argument *initial* is provided, the accumulation leads off with the *initial* value so that the output has one more element than the input iterable.

Funkce `accumulate` bude postupně zpracovávat dvě dvojice číslic a z nich vytvoří novou dvojici. Jak to uděláme?

5. Teď už je lehké dát všechno dohromady:

```
from itertools import zip_longest

def get_digits_reversed(digits):
    yield from reversed(digits)

def add_and_carry(d1:int, d2:int) -> {int, int}:
    return divmod(d1+d2,10)

digits1 = [int(c) for c in input()]
digits1 = [int(c) for c in input()]
...
```

A co kdybychom si namísto počítání vytvořili tabulku a jenom z ní odčítali součty a přenosy?