

Programování 2

3. cvičení, 5-3-2024

Farní oznamy

1. Tento text a kódy ke cvičení najdete v repozitáři cvičení na <https://github.com/PKvasnick/Programovani-2>.

2. **Domácí úkoly:**

- Tento týden jste měli středně těžké domácí úkoly
- U některých jste si poradili výborně (přesmyčky), u dalších byly problémy.
- Budeme se úkolům dnes věnovat
- Pokud se vám zdá, že jsou pro vás domácí úkoly těžké, napište mi. Je zbytečné, abyste dopláceli na to, že jste v Programování 1 nedostali co jste měli.

Dnešní program:

- Kvíz
- Domácí úkoly
- Blnární 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)): # NENENE!
    print(seznam1[i] + seznam2[i])

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

```

- Méně hranatých závorek.
- Ochrana před modifikací dat.

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 nepřírozený konstruktor, který ale umožňuje konstruovat `namedtuple` za běhu.

Domácí úkoly

1. ChatGPT a domácí úkol o anagramech

Chtěl jsem si vyzkoušet, jak ChatGPT zvládne vyřešit jeden z domácích úkolů. Následující je záznam z chatu:

Prosím, vygeneruj kód v jazyce Python, který bude řešit následující úlohu:

Samozřejmě, rád vám s tím pomohu! Prosím, popište mi co nejpodrobněji, co by váš kód měl dělat, abych vám mohl připravit co nejlepší řešení.

Na vstupu je číslo N, pak N řádků se slovy slovníku, pak číslo M a nakonec M řádků s dotazy (slovy). Pro každý dotaz najděte ve slovníku všechny jeho přesmyčky, tedy slova, která jsou složena ze stejných písmen, jen možná v jiném pořadí (každé slovo je tedy svou vlastní přesmyčkou). Přesmyčky ke každému dotazu vypište na samostatný řádek v abecedním

(lexikografickém) pořadí, oddělené mezerami. Pokud k dotazu není nalezena žádná přesmyčka, vypište prázdný řádek.

Jistě, zde je kód v jazyce Python, který vyhovuje vašim požadavkům:

python

```
from collections import Counter

# načtení vstupních hodnot
n = int(input())
dictionary = [input().strip() for _ in range(n)]
m = int(input())
queries = [input().strip() for _ in range(m)]

# vytvoření seznamu přesmyček pro každý dotaz
anagrams = {}
for query in queries:
    query_counts = Counter(query)
    anagram_list = [word for word in dictionary if Counter(word) == query_counts]
    anagram_list.sort()
    anagrams[query] = anagram_list

# vypsání výsledků
for query in queries:
    for anagram in anagrams[query]:
        print(anagram, end=' ')
    print()
```

Kód nejprve načte vstupní hodnoty a uloží je do proměnných `n`, `dictionary`, `m` a `queries`. Poté vytvoří seznam přesmyček pro každý dotaz a uloží je do slovníku `anagrams`, kde klíče jsou dotazy a hodnoty jsou seznamy přesmyček odpovídajících dotazu. Seznam přesmyček se vytvoří tak, že se pro každé slovo ze slovníku spočítají jeho písmena a porovnají se s písmeny dotazu. Pokud se shodují, slovo je přidáno do seznamu přesmyček. Seznam je seřazen lexikograficky.

Nakonec kód projde všechny dotazy a vypíše seznam přesmyček pro každý dotaz. Pokud nebyly nalezeny žádné přesmyčky, vypíše se prázdný řádek.

Toto je dost pomalý kód, protože pro každý dotaz musí projít celým seznamem slov a u každého slova porovnat písmena, z nichž pozůstává. Nešlo by to vyhledávání implementovat efektivněji?

Ano, máte pravdu, že tento kód může být pomalý pro velká vstupní data. Efektivnější řešení by mohlo být použít slovník, kde klíčem by byla hash hodnota převedeného slova a hodnotou by byl seznam všech slov se stejnou hash hodnotou. Potom by stačilo porovnávat hash hodnoty namísto písmen pro každé slovo v seznamu, což by mělo být mnohem rychlejší.

Zde je kód s tímto vylepšením:

python

```
from collections import Counter
```

```

# načtení vstupních hodnot
n = int(input())
dictionary = [input().strip() for _ in range(n)]
m = int(input())
queries = [input().strip() for _ in range(m)]

# vytvoření slovníku hash hodnot a seznamů přesmyček pro každý dotaz
anagrams = {}
for word in dictionary:
    word_hash = hash(frozenset(Counter(word).items()))
    if word_hash in anagrams:
        anagrams[word_hash].append(word)
    else:
        anagrams[word_hash] = [word]

# vypsání výsledků
for query in queries:
    query_hash = hash(frozenset(Counter(query).items()))
    if query_hash in anagrams:
        anagram_list = sorted(anagrams[query_hash])
        print(' '.join(anagram_list))
    else:
        print()

```

Kód funguje téměř stejně jako původní kód, ale namísto porovnávání písmen pro každé slovo se používá hash hodnota vytvořená z počtu výskytů jednotlivých písmen v daném slově pomocí třídy `Counter`. Slovník `anagrams` obsahuje seznamy přesmyček pro každou hash hodnotu, což by mělo být rychlejší než hledání přesmyček v celém seznamu slov.

Pozor, tato implementace má ale jednu výraznou nevýhodu. Pokud nějaká dva různá slova mají stejnou hash hodnotu, budou přidána do stejného seznamu přesmyček, i když ve skutečnosti nejsou přesmyčkami. Tento problém lze řešit pomocí nějakého dalšího mechanismu na kontrolu, zda jsou slova skutečně přesmyčkami.

Abychom předešli konfliktům hash hodnot, nemohli bychom namísto hashe vytvořit pro každé slovo nové slovo tak, že uspořádáme písmena slova podle abecedy? Takovéto abecedně seřazené slovo bychom pak mohli používat namísto hashe. Co si o tom myslíš?

Ano, použití abecedně seřazených slov místo hashovací funkce je další způsob, jak řešit konflikty hash hodnot, a mělo by to fungovat dobře. Toto řešení je sice jednodušší, ale může být náročnější na paměť, protože pro každé slovo musíme uložit jeho abecedně seřazenou verzi a porovnávat ji s abecedně seřazenými verzemi v seznamu při vyhledávání.

Zde je kód, který používá abecedně seřazené slovo namísto hashe:

python

```

# načtení vstupních hodnot
n = int(input())
dictionary = [input().strip() for _ in range(n)]
m = int(input())
queries = [input().strip() for _ in range(m)]

```

```

# vytvoření slovníku přesmyček pro každý dotaz
anagrams = {}
for word in dictionary:
    sorted_word = ''.join(sorted(word))
    if sorted_word in anagrams:
        anagrams[sorted_word].append(word)
    else:
        anagrams[sorted_word] = [word]

# vypsání výsledků
for query in queries:
    sorted_query = ''.join(sorted(query))
    if sorted_query in anagrams:
        anagram_list = sorted(anagrams[sorted_query])
        print(' '.join(anagram_list))
    else:
        print()

```

Tento kód funguje podobně jako předchozí řešení s hashovací funkcí, ale namísto hashovací funkce používá abecedně seřazené verze slov. Pokud jsou dvě slova přesmyčky, budou mít stejnou abecedně seřazenou verzi a budou přidána do stejného seznamu přesmyček. Pokud tedy máme slova "abcd" a "dcba", abecedně seřazené verze obou slov budou "abcd", a proto budou přidána do stejného seznamu přesmyček.

Obě řešení mají své výhody a nevýhody a vhodné řešení závisí na konkrétní situaci. Pokud jsou slova vstupu relativně krátká, použití abecedně seřazených slov může být efektivnější. Na druhé straně, pokud jsou slova delší a mají velký počet různých znaků, může použití hashovací funkce být rychlejší a méně náročné na paměť.

-- konec GPT chatu --

Co si o tom myslíte?

Tento chat jsem vedl s GPT3. GPT4x má pro tuto úlohu úspěšnost 80%, tedy dost vysokou a znamená to, že automaticky nabízí řešení 2 nebo 3, protože řešení 1 je sice rychlejší než typická implementace, kterou jsem viděl v domácích úkolech, ale i tak neprojde posledními testy v ReCodExu, kde se používají velké seznamy slov.

2. Vkládání prvků (a binární vyhledávání)

Na vstupu je zadána vzestupně uspořádaná posloupnost x_1, \dots, x_N a čísla y_1, \dots, y_M . Zařaďte čísla y_i na správná místa do posloupnosti.

Formát vstupu a výstupu: Na vstupu je $N + M + 2$ celých čísel na samostatných řádcích. Na prvním je číslo N , pak následují x_1 až x_N , za nimi M a konečně y_1 až y_M . Všechna čísla $x_1, \dots, x_N, y_1, \dots, y_M$ jsou navzájem různá. Na výstup vypište $N + M$ prvků výsledné posloupnosti na samostatných řádcích.

Příklad vstupu:

```
3
10
20
30
2
15
35
```

Odpovídající výstup:

```
10
15
20
30
35
```

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 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.")

```

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.

```

# 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 $O(n)$).

3. k-tý největší prvek

Tato úloha je zadána špatně., protože až po načtení dat se dovíte, kolikátý prvek máte hledat.

Takže o průběžném řešení jenom stručně:


```
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 vypočítá 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?