Programování 2

5. cvičení, 18-3-2025

Farní oznamy

1. Tento text a kódy ke cvičení najdete v repozitáři cvičení na https://github.com/PKvasnick/Programovani-2.
2. **Domácí úkoly**: 2 nové úkoly od minulého cvičení.
    1. Stejnočastá slova
    2. Maximální počet hostů v restauraci

   Dopadlo to dobře a máme práci, takže se u těchto úkolů nebudeme zastavovat.

Dnešní program:

- Kvíz
- Python: jak rychlý je můj kód
- Pořád třídění

Na zahřátí

> *"Before software can be reusable it first has to be usable." – Ralph Johnson*

Opakovaná použitelnost kódu se přeceňuje. Největší využití mívají krátké kousky kódu. Velké knihovny sami po sobě dědíme zřídka.

Co dělá tento kód

```
x = 5
y = 7

x = x + y
y = x - y
x = x - y

print(f"{x=} {y=}")
```

Co bude rychlejší? Pythonské `x, y = y, x`, anebo série třech sčítání/odečítání?

Zjistíte v následující části.

Jak rychlý je můj kód?

Jak můžeme měřit rychlost kódu?

Příklad: Ciferný součet - tři metody

```
1  def cif_soucet_1(cislo: int) -> int:
2      """mod 10 dává číslici, div 10 zbytek čísla"""
3      soucet = 0
4      while cislo:
5          soucet += cislo % 10
6          cislo //= 10
7      return soucet
8
9  def cif_soucet_2(cislo: int) -> int:
10     return sum(map(int, str(cislo)))
11
12 def cif_soucet_3(cislo: int) -> int:
13     """divmod 10 dává dvojici (zbyvajici_cislo, číslice)"""
14     soucet = 0
15     while cislo:
16         cislo, cislice = divmod(cislo, 10)
17         soucet += cislice
18     return soucet
19
```

Který kód je rychlejší?

```
1  # soubor speedy.py v adresáři code/Ex05
2  from timeit import Timer
3
4  def cif_soucet_1(cislo: int) -> int:
5      """mod 10 dává číslici, div 10 zbytek čísla"""
6      soucet = 0
7      while cislo:
8          soucet += cislo % 10
9          cislo //= 10
10     return soucet
11
12 def cif_soucet_2(cislo: int) -> int:
13     return sum(map(int, str(cislo)))
14
15 def cif_soucet_3(cislo: int) -> int:
16     """divmod 10 dává dvojici (zbyvajici_cislo, číslice)"""
17     soucet = 0
18     while cislo:
19         cislo, cislice = divmod(cislo, 10)
20         soucet += cislice
21     return soucet
22
23 if __name__ == "__main__":
24
25     t1 = Timer("cif_soucet_1(123456789)", "from speedy import cif_soucet_1")
26     print(1, t1.timeit(number = 1000000))
27
```

```
    t2 = Timer("cif_soucet_2(123456789)", "from speedy import cif_soucet_2")
    print(2, t2.timeit(number = 1000000))

    t3 = Timer("cif_soucet_3(123456789)", "from speedy import cif_soucet_3")
    print(3, t3.timeit(number = 1000000))

------
1 0.7988360999999999
2 2.7662923
3 1.2593260999999996
```

Všimněte si, že volání časovače není úplně pohodlné.

Modul `timeit` vlastně volá externí program, který spustí požadovaný kód v operačním systému (je to proto, že `timeit` modifikuje některá nastavení Pythonu (např. vypíná *garbage collector*, tedy mechanizmus, který odstraňuje z paměti objekty, které se dostali mimo jmenný prostor, kde byly vytvořeny). Proto musíte zadat nejen název funkce, kterou chcete spustit, ale také ukázat, kde ji Python najde (`from speedy import cif_soucet_1` atd.).

Pohodlnější je použít *magic* `%%timeit` v IPythonu, tedy např.

- v Pythonské konzoli PyCharmu
- v jakémkoli typu Jupyter notebooku - Google Colab, JupyterLab a pod.

Pusťte si měření víckrát.

Všimněte si, že měření času *není stabilní*: potřebujeme velkou statistiku, abychom získali jakž-takž konzistentní představu o relativní rychlosti kódů. Je to proto, že měření běží v multitaskovém prostředí a nikdy nevíme, kdy operační systém vlákno s testy odloží bokem a začne si vyřizovat něco co právě považuje za důležitější.

Který kód je rychlejší?

Nejpomalejší kód je ten, který vypadá nejmoudřeji - funkce `cif_soucet_2` . Používá `sum` a `map` a žádný otevřený cyklus, jenomže dělá konverzi z `int` na `str` a zpátky, a to je složitá operace. Z jiného pohledu vlastně vidíme, že použít tento rychlý trik k nalezení ciferního součtu čísla není o tolik pomalejší než ostatní metody.

Nejrychlejší kód je, zdá se, `cif_soucet_1`, nejspíš proto, že ho kompilátor dokáže dobře zoptimalizovat.

`cif_soucet_3` je znatelně pomalejší, nejspíš proto, že vytváří a rozbaluje tuply.

Čtení z konzole

Jedna varianta, kterou jsme dosud neměli:

Nežádej o povolení, ale o prominutí

```python
from contextlib import suppress

with suppress(EOFError):
    while p := input():
        print(int(p))

```

1. Když už není co číst, `input()` vyvolá výjimku. Takže výjimku zachytíme a klidně pokračujeme dál. Případný řádek s ukončovacím řetězcem lehce odstraníme. Takovýto postup se v Pythonu docela často používá, i když bychom měli výjimky používat nejvíc pro ošetření stavů, které nejsou pod naši kontrolou.
2. Pro zachycení výjimky nepoužíváme blok `try-except`, ale *context manager* `suppress`, tedy prostředí, které umí vrátit věci do pořádku, když se v bloku kódu stane něco ošklivého. Context managery nejspíš znáte jako prostředí, kde komunikujeme s textovým souborem, ale jejich použití je mnohem širší.
3. Toto je celkem dobrý nástroj pro načítání dat, které nemají ukončovací řetězec. Současně je platformově nezávislý, bude vám fungovat na laptopu i v ReCodExu.

Třídění (stále)

Z minula: Quickselect, quickort

Obě metody sérii částečných třídění podle pivotu, při kterých se seznam dělí na hodnoty menší než pivot, rovné pivotu a větší než pivot. Jediný rozdíl je v tom, že QuickSelect hledá k-tou největší hodnotu, a proto po každém dělení pokračuje tou částí původního seznamu, kde se tato hodnota určitě nachází. QuickSort třídí celé pole a tudíž se rekurzivně volá na všechny tři části.

Symbolicky, kód pro QuickSort funguje takto:

```python
from random import randint

def quick_sort(b):
    if len(b) < 2:
        return b
    if len(b) == 2:
        return [min(b), max(b)]
    pivot = b[randint(0, len(b)-1)]
    lows = [x for x in b if x < pivot]
    pivots = [x for x in b if x == pivot]
    highs = [x for x in b if x > pivot]
    return quick_sort(lows) + pivots + quick_sort(highs)


data = [randint(1,100) for _ in range(10)]

print(data)
print(quick_sort(data))
```

Proč symbolicky?

- Pro každou úroveň volání vytvářime další kopii původního seznamu.
- Provádíme mnoho manipulací s poli
- Rekurze není zadarmo a může narazit na omezení Pythonu.

Tedy máme poměrně hezký kód, který je určitě funkční, ale není úplně konkurenceschopný.

Dvě hlavní věci, které můžeme zlepšit, jsou volba pivotu a přerovnání dat při uspořádání podle pivotu.

Volba pivotu: median-of-medians

Optimální pivot je medián, tedy hodnota, od které je 50% ostatních hodnot menších a 50% větších. Jenomže výpočet mediánu, což je úloha ekvivalentní tomu, co řešíme (nalézt n//2 největší hodnotu), hierarchciky počítáme mediány pětic čísel, až dospějeme k rozumnému odhadu pivotu. Podrobnosti zde: https://en.wikipedia.org/wiki/Median_of_medians.

Přerovnání dat

Jak podle vás funguje tento kód?

Udělejte si z toho cvičení v porozumění kódu. Pomůže vám tužka a papír. Všimněte si, jak vám v čtení pomáhají komentáře.

Jaká je složitost tohoto kódu?

```
def partition(data, left, right, pivotIndex, n):
    pivotValue = data[pivotIndex]
    data[pivotIndex], data[right] = data[right], data[pivotIndex]  # Move pivot to end
    storeIndex = left
    # Move all elements smaller than the pivot to the left of the pivot
    for i in range(left, right):
        if data[i] < pivotValue:
            data[storeIndex], data[i] = data[i], data[storeIndex]
            storeIndex += 1
    # Move all elements equal to the pivot right after
    # the smaller elements
    storeIndexEq = storeIndex
    for i in range(storeIndex, right):
        if data[i] == pivotValue:
            data[storeIndexEq], data[i] = data[i], data[storeIndexEq]
            storeIndexEq += 1
    data[right], data[storeIndexEq] = data[storeIndexEq], data[right]  # Move pivot to its final place
    # Return location of pivot considering the desired location n
    if n < storeIndex:
        return storeIndex  # n is in the group of smaller elements
    if n <= storeIndexEq:
        return n  # n is in the group equal to pivot
    return storeIndexEq  # n is in the group of larger elements

data = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
print(data)
pivotIndex = 2  # 3rd element is 4
n = 6        # 6th smallest element is 6
result = partition(data, 0, len(data)-1, pivotIndex, n)
```

```
print("Partitioned index:", result)
print("Data after partitioning:", data)

```

Counting sort

Pro celá čísla nebo podobné objekty s omezeným rozsahem hodnot máme přirozenou metodu třídění v lineárním čase:

- pro každou možnou hodnotu objektu si zřídíme příhrádku
- roztřídíme data podle přihrádek
- sesypeme přihrádky do výsledného pole. Pomůžeme si kumulativními součty obsazeností přihrádek, které nám dají pozice pro jednotlivé hodnoty ve výsledném setříděném poli. Postupujeme od konce pole a "vyprázdňujeme" přihrádky do správných pozic. Toto zaručí, že třídění bude stabilní - tedy stejné prvky budou ve stejném pořadí jako ve vstupním poli.

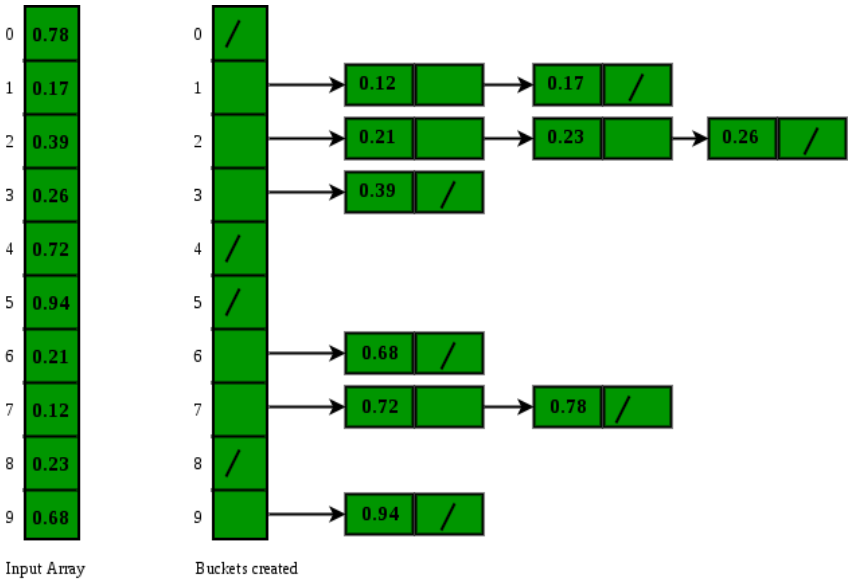
Implementace může vypadaat takto:

```
from random import randint

def count_sort(b:list[int], rmax: int) -> list[int]:
    bins = [0] * rmax   # vytvoříme přihrádky
    # roztřídíme
    for elem in b:
        bins[elem] += 1
    # kumulativní součty obsazeností
    for i in range(rmax):
        if i==0:
            continue
        bins[i] += bins[i-1]
    # naplníme výsledné pole
    output = [0] * len(b)
    for i in reversed(range(len(b))):
        j = b[i]
        bins[j] -= 1
        output[bins[j]] = j
    return output

r = 10
data = [randint(0,r-1) for _ in range(100)]

print(*data)
print(*count_sort(data, r))
```

Kritická hodnota u tohoto algoritmu je počet možných hodnot ve vstupním poli. Pokud tuto hodnotu známe a je podstatně menší než počet hodnot ve vstupním poli, je třídění lehké. Nezapomínejme také na to, že pokud možné hodnoty nejsou čísly, budeme je možná muset setřídit.

Bucket sort

- Nahrubo si setřídíme čísla do přihrádek
- Setřídíme obsah přihrádek
- Spojíme do výsledného seznamu



- Jednoduchý algoritmus
- Občas musíme popřemýšlet, jak vytvořit přihrádky (viz domácí úkol)
- Pro jednotlivé kroky můžeme použít counting sort
- Funguje krásně mysticky.

Varianta: deque

Namísto toho, abychom pokaždé sesypali kyblíky do pole, prostě průběžně dělíme věci do z kyblíků do jiných kyblíků. Potřebujeme ale zarážky, abychom věděli, co přišlo nově.

```python
from collections import deque

words: list[str] = []

while "-end-" not in (line := input()):
    words.append(line.strip())


def get_bucket(word: str, order: int) -> int:
    if order < len(word):
        return ord(word[order]) - ord("a") + 1
    return 0


def queue_bucketsort(words: list[str]) -> list[str]:
    buckets = [deque([""]) for _ in range(ord("z") - ord("a") + 1)]
    # 1. Fill buckets
    buckets[0].extendleft(words)
    max_length = len(max(words, key=len))
    for order in reversed(range(max_length)):
        for bucket in buckets:
            while (w := bucket.popleft()) != "":
```

```
                buckets[get_bucket(w, order)].append(w)
        for bucket in buckets:
            bucket.append("")
        print(order)
        for i, bucket in enumerate(buckets):
            print(chr(i + ord("a")-1), ": ", *bucket)

    return [w for bucket in buckets for w in bucket if w != ""]


words = queue_bucketsort(words)
print(*words, sep="\n")
print("-end-")
```

Domácí úkoly

Bude to samý Bucket sort, ale doufám, že se vám úlohy budou líbit.

1. **Klasický bucketsort** - třídíte čísla, ale vypisujete jednotlivá stadia.
2. **Bucketsort pro seznam slov** - třídíte slova a musíte upravit algoritmus podle toho.