Příklad ke zkoušce #42 – Insertion sort

Úvod do programování

Martin Řanda

Úvod

Tento dokument slouží jako technická zpráva k programům `insert_sort_alg.py` a `data_manip.py`.

První zmíněný program se zabývá setříděním libovolné posloupnosti reálných čísel. Soubor `data_manip.py` pak obsahuje podpůrné funkce k načtení posloupnosti čísel z textového souboru a k exportu výsledku, a proto je jeho obsah rozebírán jen okrajově.

Oba programy byly otestovány a shledány funkčními na verzích 3.8 a 3.9 programovacího jazyka Python.

Zadání

Navrhněte program v jazyku Python, který z textového souboru načte nesetříděnou posloupnost reálných čísel tvořenou n prvky, provede jejich vzestupné či sestupné řazení podle daného vstupního parametru a výsledek uloží do textového souboru.

Definice problému a insertion sort

Cormen et al. (2009) definuje třídící úlohu následujícím způsobem. Nechť $\{Y_1, Y_2, ..., Y_n\}$ je posloupnost reálných čísel, kde n je přirozené číslo. Vzestupným setříděním posloupnosti $\{Y_i\}_1^n$ budeme rozumět posloupnost prvků $\{Y_1', Y_2', ..., Y_n'\}$, pro které platí

$$Y_1' \leq Y_2' \leq \ ... \ \leq Y_n'.$$

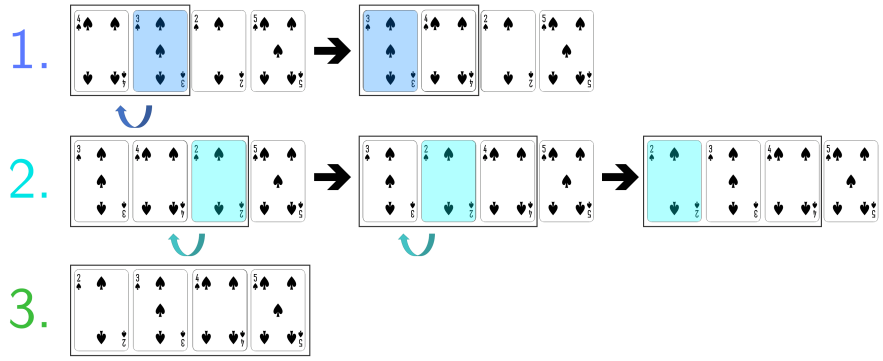
Setřídíme-li posloupnost $\{Y_i\}_1^n$ sestupně, obdržíme výše uvedenou posloupnost v opačném pořadí, tedy $\{Y_n', Y_{n-1}', ..., Y_1'\}$, a platí

$$Y_n' \geq Y_{n-1}' \geq \ ... \ \geq Y_1'.$$

Jednoduše řečeno, naším cílem bude seřadit prvky posloupnosti v sestupném nebo vzestupném pořadí. Problém lze například vyřešit pomocí skupiny řadících algoritmů, kterou nazýváme *řazení vkládáním* neboli *insertion sort*. Tato metoda může být člověku povědomá, jelikož, jak Sedgewick (1998) podotýká, mnoho lidí tento způsob třídění používá při řazení karet.

Existující algoritmy

Knuth (1998) jako nejjednodušší implementaci řazení vkládáním považuje *straight insertion*, kde se jednotlivě prochází klíče (prvky) posloupnosti a postupně se hodnota prvku Y_j porovnává s hodnotami $Y_{j-1}, Y_{j-2}, ...$ Prvek je vkládán na pozice $(j-1), (j-2), ...$, dokud není na správném místě. Postupně tak vytváříme setříděné podposloupnosti, jejichž počet prvků v každém dalším kroku narůstá. Obrázek níže ilustruje tento proces na čtyřech kartách:



Mezi další způsoby implementace algoritmu insertion sort lze považovat metodu *binary insertion*, která používá tzv. *binary search* algoritmus ke setřídění posloupnosti.

Dále lze zmínit *shellsort*, který dle Knuth (1998) zásadně vylepšuje *straight insertion* tím způsobem, že dovoluje provádět "dlouhé skoky" namísto "krátkých kroků".

V neposlední řadě také existuje metoda *address calculation sorting*, při které se odhaduje přibližná finální pozice jednotlivých prvků, aby se optimalizoval konečný počet srovnávání.

Popis zvoleného algoritmu

Budeme se pokoušet implementovat první ze zmíněných přístupů, tedy *straight insertion*. Po vzoru článku "Řazení vkládáním" (Wikipedia 2019) si rozdělme postup do 5 kroků. Předpokládejme, že nesetříděnou posloupnost máme patřičným způsobem načtenou:

1. *Krok*
   Posloupnost rozdělme na 2 podposloupnosti – seřazenou a neseřazenou. Seřazená podposloupnost obsahuje první dva prvky a neseřazená podposloupnost obsahuje zbylých $n - 2$ prvků.

2. *Krok*
   Z neseřazené podposloupnosti vyberme nejvzdálenější prvek od prvního prvku. Tento prvek postupně porovnávejme a vyměňujme s předchozími prvky, dokud nenarazíme na prvek se stejnou nebo menší hodnotou nebo dokud se nedostaneme na první pozici.

3. *Krok*
   Zvětšíme seřazenou posloupnost o jeden prvek, což zároveň odpovídajícím způsobem zmenší neseřazenou posloupnost. Pokračujme na *Krok 2*, dokud není neseřazená posloupnost prázdná.

4. *Krok*
   Je-li neseřazená posloupnost prázdná, potom seřazená posloupnost obsahuje všechny prvky ve vzestupném pořadí.

Pro seřazení prvků od největšího po nejmenší pouze změníme způsob porovnávání a analogicky nalezneme seřazenou posloupnost.

Rozbor algoritmu a problematické situace

V první řadě je nutné si uvědomit, že Python označuje první prvek posloupnosti indexem 0, což ale není nijak zvlášť problematické. V praxi to pouze znamená, že pomocí cyklu `for` budeme muset postupovat od prvku s indexem 1 až po prvek s indexem n.

Z předchozí podkapitoly si lze všimnou, že *Krok 3* odkazuje na *Krok 2*, který po dokončení opět pokračuje na *Krok 3*. Tohoto "zacyklení" lze docílit například pomocí `while` cyklu. Musíme se však zamyslet, podle jaké podmínky či podmínek se tento cyklus bude řídit, abychom z něj ve správnou dobu "vyskočili" a neskončili v nekonečném cyklu. V *Kroku 2* lze identifikovat 2 podmínky:

- "postupně porovnávejme [...] s předchozími prvky, dokud nenarazíme na prvek se stejnou nebo menší hodnotou"
- "dokud se nedostaneme na první pozici"

Pokud bychom tedy tyto podmínky měly nějakým způsobem zapsat v podobě kódu, dospěli bychom k něčemu jako (`pos` jakožto `position`):

```
while sequence[pos - 1] > sequence[pos] and pos > 0:
    ...
```

V tuto chvíli je pravděpodobně ta nejtěžší část tohoto problému hotova a nyní je pouze potřeba projít všechny prvky posloupnosti a adekvátně je zařadit. Na

procházení prvků použijeme zmíněný cyklu `for`:

```
for index in range(1, len(sequence)):
    ...
```

Nyní se opět vraťme k cyklu `while`, ve kterém musíme dořešit správné zařazení prvků. Z *Kroku 2* vyplývá, že prvek vyměňujeme s předchozím prvkem, dokud je hodnota předcházejícího prvku ostře větší. Toho lze docílit následujícím způsobem (daný zápis použijeme pouze pro přehlednost):

```
while sequence[pos - 1] > sequence[pos] and pos > 0:
    previous = sequence[pos - 1]
    current = sequence[pos]
    previous, current = current, previous
    ...
```

Aby byl výše zapsaný cyklus funkční, je třeba jej vnořit do našeho `for` cyklu. Bez závažné újmy na efektivitě algoritmu deklarujme `pos = index` pro přehlednost. Zbývá už jen postupně snižovat poziční argument, abychom se mohli postupně posouvat. Toho lze docílit například pomocí `pos -= 1`, tedy pozice se nám každým během cyklu `while` sníží o 1, dokud nenarazí na nulu.

Posloupnost je seřazena vzestupně a zbývá jen vyřešit případ, kdy bychom chtěli opačné pořadí. Pokud jsme načetli naši posloupnost jako seznam, můžeme použít metodu `reverse()`, která, jak název napovídá, obrátí pořadí seznamu (starší verze programu). Také je možno vytvořit obdobnou funkci jako signum, která na základě vstupu přenásobí aktuálně evaluované prvky číslem 1 či -1 a v druhém případě tak otočí pořadí (nová verze programu).

Protože je export dat řešen v jiném souboru, pro předávání vstupního argumentu způsobu řazení (vzestupně či sestupně) je použita funkce `lambda`. Vytvoří se tak tzv. *anonymní funkce*, ve které se již nespecifikuje způsob setřídění.

Možná vylepšení

Uživatel na vstupu vybírá, jestli chce posloupnost seřadit vzestupně nebo sestupně. Program ale vždy nejdříve posloupnost seřadí vzestupně a až poté případně pořadí otáčí (starší verze programu). Výhodou tohoto přístupu je jednoduchost implementace. Zásadní nevýhodou je však fakt, že v případě sestupné posloupnosti probíhají dvě řazení, což může mít negativní dopad na výkonnost algoritmu pokud je n dostatečně velké.

Vstupní a výstupní data

Z textového souboru načteme nesetříděnou posloupnost reálných čísel, která je tvořena n prvky. Hodnoty by měly být od sebe nějakým způsobem odděleny,

například s využitím zalomení řádku (line break). Když se každý prvek nachází na samostatné lince textového souboru, je pak možné jednoduše pomocí metody `readlines()` načíst jednotlivé řádky do seznamu. V repozitáři je k dispozici soubor `sequence.txt`, který obsahuje ilustrační vstupní data.

Výstupem programu je textový soubor, který obsahuje vzestupně nebo sestupně seřazenou posloupnost.

Závěrečné shrnutí

V tomto dokumentu bylo popsáno, jakým způsobem lze implementovat řadící algoritmus *insertion sort* na nesetříděnou posloupnost reálných čísel.

Klíčovým krokem implementace algoritmu je identifikace podmínek vnořeného `while` cyklu a vyvarování se nekonečnému "zacyklení". Pro sestupně seřazenou posloupnost lze použít metodu `reverse()`, což ale zároveň představuje možné riziko ztráty výkonnosti, pokud posloupnost obsahuje velký počet prvků.

Program `insert_sort_alg.py` obsahuje konkrétní řešení tohoto příkladu a s pomocí `data_manip.py` jsou načítána a exportována relevantní data.

Seznam literatury

Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., 2009. *Introduction to algorithms*. MIT press.

Hetland, M.L., 2014. *Python Algorithms: mastering basic algorithms in the Python Language*. Apress.

Knuth, D.E., 1998. *The Art of Computer Programming, volume 3: Sorting and Searching*. Addison-Wesley Professional.

Sedgewick, R., 1998. *Algorithms in c++, parts 1-4: fundamentals, data structure, sorting, searching (Vol. 1)*. Pearson Education.

Wikipedia, 2019. *Řazení vkládáním* [Online]. Dostupné z: https://cs.wikipedia.org/wiki/Řazení_vkládáním [k 19.01.2021].