Algorytmy sortowania

Znalazłaś, znalazłeś błąd?
Coś wymaga lepszego wytłumaczenia?
Wszystkie uwagi proszę zgłaszać tutaj: `dprze@agh.edu.pl`

wersja 2019–09–27

O teorii algorytmów słów kilka

Algorytmy sortujące mają za zadanie ułożyć elementy listy w ustalonym porządku w taki sposób, aby:

- ciąg wynikowy zawierał elementy w niemalejącej kolejności
- ciąg wyjściowy stanowił permutację ciągu wejściowego

Algorytmów sortowania jest wiele, przykładowo: sortowanie szybkie (*quicksort*), przez scalanie, przez kopcowanie, przez wstawianie, introspektywne, przez wybieranie, Shella, bąbelkowe, biblioteczne, koktajlowe, kubełkowe, przez zliczanie... Ponieważ metody te różnią się między sobą m.in. złożonością obliczeniową, stabilnością, możliwością zrównoleglenia obliczeń, wybór konkretnej metody zależy od aplikacji, w której potrzebujemy dokonać sortowania.

Info: W niniejszym ćwiczeniu skupimy się na analizie złożoności czasowej wybranych algorytmów z dwóch powodów – po pierwsze, będzie nam łatwo przeprowadzić eksperymenty i porównać algorytmy; po drugie, często będzie to jedno z najważniejszych kryteriów wyboru algorytmu.

Poniżej przedstawiliśmy wyjaśnienia podstawowych własności algorytmów (w ogólności, nie tylko algorytmów sortowania), w kontekście ich implementacji na dowolnej platformie.

Złożoność obliczeniowa

Złożoność obliczeniowa to funkcja ilość zasobów komputerowych (czasu procesora, pamięci operacyjnej) niezbędnych do wykonania programu realizującego algorytm względem rozmiaru danych wejściowych. Funkcja ta zazwyczaj określona jest z pewnym przybliżeniem.

Złożoność pamięciowa

Złożoność pamięciowa określa ilość zasobów pamięciowych niezbędnych do działania danego algorytmu. Można wyróżnić pamięć potrzebną do przechowywania:

- kodu algorytmu
- danych wejściowych
- danych wyjściowych (przy czym algorytmy działające „w miejscu" nie potrzebują bloku pamięci na dane wyjściowe, ponieważ operują na bloku danych wejściowych)
- danych pośrednich (danych pomocniczych tworzonych i wykorzystywanych w trakcie działania algorytmu)

Złożoność czasowa

Złożoność czasowa określa czas potrzebny do zrealizowania algorytmu. Ponieważ na różnych maszynach ten sam algorytm będzie wykonywał się w różnym czasie, złożoności czasowej nie wyrażamy w standardowych jednostkach czasu (np. sekundach). Zamiast tego zwykle zliczamy *operacje dominujące* przy założeniu, że każda z nich zajmuje jedną hipotetyczną jednostkę czasu (np. jeden cykl procesora).

Złożoność pesymistyczna, oczekiwana i optymistyczna

Dla algorytmów, w przypadku których złożoność obliczeniowa zależy od egzemplarza danych wejściowych, wyróżnia się złożoności:

- pesymistyczną (O) – zużycie zasobów dla najbardziej niekorzystnego zestawu danych
- oczekiwaną / średnią (Θ) – zużycie zasobów dla typowych (tzw. losowych) danych
- optymistyczną (Ω) – zużycie zasobów dla najkorzystniejszego zestawu danych

Przykład (dla złożoności czasowej). Rozważ algorytm sprawdzający, czy w zbiorze danych wejściowych jest liczba ujemna. W przypadku optymistycznym pierwsza ze sprawdzonych wartości okaże się ujemna, więc optymistyczna złożoność czasowa wynosi $\Omega(n) = 1$. W przypadku pesymistycznym w zbiorze nie ma liczb ujemnych bądź jest jedna i algorytm znajdzie ją jako ostatnią sprawdzaną wartość, dlatego złożoność pesymistyczna $O(n) = n$. Złożoność oczekiwana wyniesie natomiast $\Theta(n) = n/2$, gdyż liczba ujemna będzie znajdowana wcześniej bądź później, ale średnio po sprawdzeniu połowy zbioru.

Info: Typowe rzędy złożoności (n – rozmiar danych wejściowych):

- $O(1)$ – stała złożoność (tj. ilość zasobu niezbędnego do działania algorytmu nie zależy od danych wejściowych)
- $O(\log n)$ – złożoność logarytmiczna
- $O(n)$ – złożoność liniowa
- $O(n \log n)$ – złożoność liniowo-logarytmiczna
- $O(n^2)$ – złożoność kwadratowa
- $O(n^X)$ – złożoność wielomianowa (X – dowolna stała)
- $O(X^n)$ – złożoność wykładnicza ($X > 2$ – dowolna stała)

W przypadku złożoności czasowej który rząd odpowiada najwolniejszemu, a który najszybszemu algorytmowi?

Algorytm 1: sortowanie szybkie - *quicksort*

Quicksort jest algorytmem z grupy „dziel i zwyciężaj". Oznacza to, że zamiast rozwiązywać oryginalny (złożony) problem dzielimy go na mniejsze podproblemy (które jesteśmy w stanie rozwiązać), a na koniec scalamy rozwiązania podproblemów, aby uzyskać rozwiązanie oryginalnego problemu.

Quicksort to algorytm rekurencyjny, działający w następujący sposób: Wybieramy „element rozstrzygający" (ang. *pivot*) i porządkujemy listę w taki sposób, aby wartości mniejsze przesunąć na lewo od pivota, a większe – na prawo. Dla każdej z tak utworzonych grup (tj. dla grupy położonej na lewo od pivota oraz dla grupy położonej na prawo od pivota) ponownie wykonujemy powyższą procedurę – o ile grupa zawiera choć dwa elementy. Pivot może zostać wybrany na kilka sposobów, np. jako pierwszy, środkowy, ostatni bądź losowy element listy. Najważniejszym składnikiem algorytmu jest sposób porządkowania wartości względem elementu rozstrzygającego.

Oto pseudokod algorytmu quicksort:

Algorithm 1: `Quicksort`

Input: (I – nieposortowana lista, *start* – początek grupy, *stop* – koniec grupy)

```
i ← start
j ← stop
pivot ←? (wybór zależny od metody)
while i < j do
    i ← i + 1 dopóki I[i] < pivot
    j ← j − 1 dopóki I[j] > pivot
    if i ⩽ j then
        zamień miejscami I[i] z I[j]
        i ← i + 1
        j ← j − 1
    end
end
if start < j then
    Quicksort(I, start, j)
end
if i < stop then
    Quicksort(I, i, stop)
end
```

Zilustrujmy działanie algorytmu *Quicksort* dla przykładowego zestawu liczb $I = [2, 25, 9, 14, 1, 3, 10]$:

1. $i = 0$, $j = 6$, $pivot = 14$ (środkowy element)
2. Szukam i: $i = 1$, bo $I[1] == 25 \geqslant 14$
3. Szukam j: $j = 6$, bo $I[6] == 10 \leqslant 14$
4. $i \overset{?}{\leqslant} j$: zamieniam $I[1]$ z $I[6]$ i inkrementuję i oraz dekrementuję j
   $I = [2, \underline{10}, 9, 14, 1, 3, \underline{25}]$, $i = 2$, $j = 5$
5. Szukam i: $i = 3$, bo $I[3] == 14 \geqslant 14$
6. Szukam j: $j = 5$, bo $I[5] == 3 \leqslant 14$
7. $i \overset{?}{\leqslant} j$: zamieniam $I[3]$ z $I[5]$ i inkrementuję i oraz dekrementuję j
   $I = [2, 10, 9, \underline{3}, 1, \underline{14}, 25]$, $i = 4$, $j = 4$
8. Szukam i: $i = 5$, bo $I[5] == 14 \geqslant 14$
9. Szukam j: $j = 4$, bo $I[4] == 1 \leqslant 14$
10. $i \not< j$: koniec
11. REKURENCJA:
    - $start == 0 < j == 4$: Quicksort(I, 0, 4) (...)
    - $i == 5 < stop == 6$: Quicksort(I, 5, 6) (...)

Info: W pokazanym przykładzie zaproponowano wybór elementu rozstrzygającego jako wyraz środkowy listy (grupy). W przypadku listy o parzystej liczbie elementów n można wybrać np. element o indeksie $\lfloor n/2 \rfloor$.

Czasowa złożoność algorytmu Quicksort

- przypadek optymistyczny: $\Omega(n \log n)$
  – za każdym razem jako pivot wybrana zostaje mediana
- przypadek oczekiwany: $\Theta(n \log n)$
  – równomierny rozkład prawdopodobieństwa wyboru mediany jako pivotu
- przypadek pesymistyczny: $O(n^2)$
  – za każdym razem jako pivot wybierany zostaje element najmniejszy lub największy

Algorytm 2: sortowanie bąbelkowe – *bubble sort*

Sortowanie bąbelkowe to prosty algorytm sortujący, który przechodzi przez listę porównując dwa sąsiednie elementy i zamienia je miejscami, jeśli są w złej kolejności. Przejście jest powtarzane dopóki zmiany nie są już potrzebne – czyli do momentu posortowania listy.

Zilustrujmy to przykładem listy $I = [2, 25, 9, 14, 1]$ (pogrubiono pary porównywanych elementów):

- $[\mathbf{2,25}, 9, 14, 10] \rightarrow [\mathbf{2,25}, 9, 14, 10]$
- $[2, \mathbf{25,9}, 14, 10] \rightarrow [2, \mathbf{9,25}, 14, 10]$
- $[2, 9, \mathbf{25,14}, 10] \rightarrow [2, 9, \mathbf{14,25}, 10]$
- $[2, 9, 14, \mathbf{25,10}] \rightarrow [2, 9, 14, \mathbf{10,25}]$
- $[\mathbf{2,9}, 14, 10, 25] \rightarrow [\mathbf{2,9}, 14, 10, 25]$
- $[2, \mathbf{9,14}, 10, 25] \rightarrow [2, \mathbf{9,14}, 10, 25]$
- $[2, 9, \mathbf{14,10}, 25] \rightarrow [2, 9, \mathbf{10,14}, 25]$
- $[2, 9, 10, \mathbf{14,25}] \rightarrow [2, 9, 10, \mathbf{14,25}]$

Algorytm można zapisać w następujący sposób:

Algorithm 2: `Bubblesort`

Input: (I – nieposortowana lista)

```
n ← length(I)
while n > 1 do
    for i ← 1 : n do
        if I[i − 1] > I[i] then
            zamień I[i − 1] z I[i]
        end
    end
    n ← n − 1
end
```

Uwaga: Zoptymalizuj algorytm w taki sposób, aby w przypadku posortowania listy nie generować niepotrzebnych, kolejnych przejść przez listę.

Czasowa złożoność algorytmu sortowania bąbelkowego

- przypadek optymistyczny: $\Omega(n)$
  – gdy lista wejściowa jest posortowana
- przypadek oczekiwany: $\Theta(n^2)$
- przypadek pesymistyczny: $O(n^2)$

Część praktyczna – przebieg laboratorium

Ćwiczenie polega na implementacji dwóch opisanych wyżej metod sortowania i wykonaniu testów porównujących ich złożoność czasową dla zróżnicowanych, reprezentatywnych, danych wejściowych.

Info: Po co te testy? Przeprowadzenie kilku testów czasowych pozwoli Ci lepiej zrozumieć pokazane algorytmy oraz teorię o złożoności obliczeniowej i czasowej (którą zanudzaliśmy Cię powyżej...). Kolejno: porównasz działanie dwóch algorytmów dla różnych danych wejściowych; sprawdzisz jakie kombinacje wykonują się „najszybciej", a na których wyniki trzeba poczekać dłużej; a na końcu skonfrontujesz rezultaty z teorią.

Żeby zrealizować laboratorium musisz wykonać zadania opisane niżej – zadania 1 i 2 możesz wykonać w dowolnej kolejności. Plotka głosi, że implementacja algorytmu bąbelkowego sprawia mniej trudności. Zadania 3 i 4 możesz zrealizować w nowym pliku .py, importując pakiet `sort`. Skorzystaj z szablonu przygotowanego przez prowadzących (nie zmieniaj ani nazw funkcji, ani nazwy pliku!), pamiętaj o uzupełnieniu nagłówka imieniem, nazwiskiem i numerem legitymacji.

Zadanie 1

Zaimplementuj algorytm quicksort. Funkcja powinna zwrócić posortowaną listę, podczas gdy lista wejściowa powinna pozostać niezmieniona.

Zadanie 2

Zaimplementuj algorytm bubblesort. Funkcja powinna zwrócić krotkę postaci: (posortowana lista, liczba wykonanych porównań). Pamiętaj o zoptymalizowaniu algorytmu w taki sposób, aby w przypadku uzyskania posortowanej listy nie generować dalszych, niepotrzebnych już przejść przez listę. Lista wejściowa powinna pozostać niezmieniona.

Zadanie 3

Przygotuj następujący zestaw danych wejściowych do testów:

1. Posortowana lista
2. Lista posortowana odwrotnie (elementy kolejno od największego do najmniejszego)
3. Lista składająca się z elementów o równych wartościach
4. Lista z losowymi wartościami elementów

Aby wygenerować listę losowych wartości możesz użyć:

```
import random
random.sample(range(a, b), n)
```

Długość list jest dowolna – ważne, żeby nie były ani „za krótkie", ani „za długie" (1000 elementów w zupełności wystarczy).

Zadanie 4

Wykonaj testy czasowe każdego algorytmu dla każdej z list wejściowych. Do pomiaru czasu wykonania fragmentu kodu możesz użyć funkcji `timeit`:

```
from timeit import timeit
t1_bubble = timeit("bubblesort(I)", number=1, globals=globals())
```

Pamiętaj, że aby wyniki eksperymentu miały jakikolwiek sens, porównanie działania algorytmów należy przeprowadzić z użyciem **takich samych list**. Co więcej, aby wyniki były reprezentatywne należy powtórzyć pomiar wielokrotnie (np. 1000 razy dla każdego przypadku) i uśrednić wyniki.

Zadanie 5

Czy wyniki eksperymentu zgadzają się z teorią? Przedstaw rezultaty prowadzącej/prowadzącemu.

Uwaga: Rozwiązanie umieść na platformie UPeL, w odpowiednim miejscu. Pamiętaj aby zamieścić wszystkie potrzebne pliki: sort.py, __init__.py oraz (jeżeli stworzyłaś/eś taki plik) plik z testami czasowymi. Nie pakuj plików do archiwum. Spełnienie tych wymagań jest bardzo ważne, ponieważ zadanie jest oceniane **automatycznie** (oprócz kilku Szczęśliwców, których rozwiązania zostaną „wzięte pod lupę" przez prowadzących...).