Algorytmy sotowania

Bubble sort, Merge sort, Selection sort, Bucket sort

Programowanie dla Fizyków

Adrianna Saribekyan, Ewelina Kowal

18.06.2021

Spis treści

1 Wiadomości wstępne

1.1 Cel pracy

Celem tego ćwiczenia było porównanie czterech algorytmów sortowania: bąbelkowego (bubble sort), przez wybieranie (selection sort), przez scalanie (megrge sort) oraz kubełkowego (bucket sort), poprzez implementację w Pythonie i analizę, pokazując że należy ostrożnie dobrać algorytm do problemu w zależności od różnych parametrów.

1.2 Wstęp teoretyczny - czym są algorytmy sortowania?

Sortowanie danych polega na uporządkowaniu danych, w taki sposób, aby zniwelować problem wyszukiwania danych, aby były one czytelnie przedstawione odbiorcy. Zadaniem algorytmów sortujących jest przedstawienie ciągu liczb n w sposób rosnący.

1.2.1 Przykładowe rodzaje algorytmów sortowania

Możemy je podzielić na **stabilne** i **niestabilne**. Stabilne algorytmy sortowania gwarantują, że kolejność zostanie zachowana, natomiast niestabilne nie dają takiej pewności. Częstym probelemem algorytmów jest zamiana miejscami elementów o takich samych wartościach, co w niektórych przypadkach obniża ich efektywność oraz wydłuża czas akcji.

Stabilne:

- **sortowanie bąbelkowe**
- sortowanie przez wstawianie
- **sortowanie przez scalanie**
- **sortowanie kubełkowe**
- sortowanie przez zliczanie

Niestabilne:

- szybkie sortowanie
- sortowanie przez kopcowanie
- **sortowanie przez wybór**
- sortowanie Shella

1.2.2 Złożoność obliczeniowa

Nie bez powodu istnieje tak wiele algorytmów sortujących. Różnią się one wydajnością i dokładnością w zależności od wielu czynników. Dobierając algorytm do problemu, należy zwrócić uwagębprzede wszystkim na:

- **zastosowanie** - nie wszystkie algorytmy służą do sortowania każdego rodzaju danych;
- **złożoność czasowa** - jest to główne kryterium wydajności, ta operacja jest nazywana analizą asymptotyczną (to pojęcie odnosi się do obliczania czasu dowolnej operacji w jednostkach matematycznych). Wyróżniamy trzy typy: dolna granica, górna granica oraz średnia. Przykładowe wielkości złożoności czasowej, gdzie n to liczba elementów do posortowania: $O(1)$ - stała, $O(logn)$ - logarytmiczna, $O(n)$ - liniowa, $O(nlogn)$, $O(n^2)$ - kwadratowa, $O(n^3)$ - sześcienna, $n^{O(1)}$ - wielomianowa, $2^{O(n)}$ - wykładnicza,. Algorytmy sortujące przy pomocy porównań nie osiągną lepszej wartości dolnego ograniczenia złożoności niż $O(nlogn)$.

- **złożoność pamięciowa** - jest to miara efektywności wykorzystania zasobów przez algorytm. Za pomocą notacji asymptotycznej określa dodatkową ilość pamięci potrzebną algorytmowi na zrealizowanie zadania.
- **stabilność** - algorytm nazywamy stabilnym, jeśli podczas sortowania zachowuje on kolejność występowania elementów o tym samym kluczu
- **inne powody** - np. szczególne przypadki zastosowań, łatwość implementacji, wydajność dla małych rozmiarów danych

2 Implementacja algorytmów w Pyhonie

2.1 Sortowanie bąbelkowe (Bubble sort)

Sortowanie bąbelkowe polega na porównaniu dwóch kolejnych wyrazów danego ciągu i ułożenie ich w odpowiedniej kolejności. Każdy element zbioru jest przesuwany do momentu aż napotka na swojej drodze większy od siebie element. Algorytm ten posiada stosunkowo krótki kod, łatwy w implementacji i zrozumieniu. Sortowanie elementów w miejscu i przechowywanie ich w pamięci pozwala na zmiejszenie złożoności pamięciowej algorytmu. Wykładnicza złożoność czasowa prowadzi jednak do wydłużenia czasu działania metody.

```python
def sort_babelkowe(tab, n):  # tab=liczby
    j = 1
    while n > 1:
        for x in range(0, n-1):
            if tab[x] > tab[x+1]:  # sprawdzanie, czy wybrany element jest mniejszy od następnego
                tab[x], tab[x+1] = tab[x+1], tab[x]  #zamiana miejscami elementów ( min -> max )
                print("\nKrok ", j, " : Zamiana miejscami liczby ", tab[x], " z liczbą ", tab[x+1])
                j += 1
        n -= 1
```

Rysunek 1: Implementacja algorytmu bąbelkowego.

```python
while True:
    print("Podaj ilość elementów tablicy: ")
    a = int(input())
    tablica = []
    for i in range(a):
        print("Podaj element nr ", (i + 1))
        tablica.append(int(input()))
    print("Twoja tablica: ")
    wypisz(tablica)
    print("""\nWybierz metodę sortowania:
    1 - bąbelkowe,
    2 - przez wybór,
    3 - przez scalanie,
    4 - kubełkowe""")
    metoda = input("\nWybieram: ")
```

(a)

```python
    if metoda == "1":
        sort_babelkowe(tablica, a)
        print("\nTwoja tablica po sortowaniu bąbelkowym: ")
        wypisz(tablica)
    elif metoda == "2":
        sort_wyb(tablica, a)
        print("\nTwoja tablica po sortowaniu przez wybór: ")
        wypisz(tablica)
    elif metoda == "3":
        sort_scal(tablica, 0, a - 1)
        print("\nTwoja tablica po sortowaniu przez scalanie: ")
        wypisz(tablica)
    elif metoda == "4":
        sort_kub(tablica, a)
        print("\nTwoja tablica po sortowaniu kubełkowym: ")
        wypisz(tablica)
    else:
        print("Wybierz poprawnie opcję 1, 2 lub 3.")

    czy_menu = input("\nCzy chcesz wrócić do menu? (Tak/Nie) ")
    if czy_menu.lower() == "nie":
        break
```

(b)

Rysunek 2: Kod proszący użytkownika o dane i wywołujący funkcje.

2.1.1 Działanie

```
Podaj ilość elementów tablicy:
4
Podaj element nr  1
5
Podaj element nr  2
2
Podaj element nr  3
8
Podaj element nr  4
1
Twoja tablica:
5 | 2 | 8 | 1 |
Wybierz metodę sortowania:
    1 - bąbelkowe,
    2 - przez wybór,
    3 - przez scalanie,
    4 - kubełkowe

Wybieram: 1

Krok  1  : Zamiana miejscami liczby  2  z liczbą  5

Krok  2  : Zamiana miejscami liczby  1  z liczbą  8

Krok  3  : Zamiana miejscami liczby  1  z liczbą  5

Krok  4  : Zamiana miejscami liczby  1  z liczbą  2

Twoja tablica po sortowaniu bąbelkowym:
1 | 2 | 5 | 8 |
Czy chcesz wrócić do menu? (Tak/Nie) Nie
```

Rysunek 3: Wynik użycia algorytmu sortowania bąbelkowego.

2.2 Sortowanie przez scalanie (Merge sort)

Algorytm sortowania przez scalanie polega na rekuryncyjnym dzieleniu tablicy wejściowej na podtablice, aż do momentu uzyskania jednoelementowych tablic. W ostatniej fazie sortowania, wartości elementów znajdujących się w podtablicy są porównywane, a następnie łączone według ustalonej zasady. Jest to jeden z logarytmów szybkich - złożonośc kwardatowa została zastapiona logarytniczną, co w dużej mierze wpływa na skrócenie czasu pracy. Merge sort wykorzystuje dodatkową tablice o takim samym rozmiarze jak tablica wejściowa, co znacznie wpływa na jego złożoność pamięciową.

```
30  def scal(tab, n, m, r):
31      lsize = m - n + 1
32      rsize = r - m
33
34      left = [0] * lsize
35      right = [0] * rsize
36
37      for x in range(lsize):
38          left[x] = tab[n + x]
39      for y in range(rsize):
40          right[y] = tab[m + y + 1]
```

Rysunek 4: Implementacja algorytmu sortowania przez scalanie.

```
42      left_idx = 0
43      right_idx = 0
44      curr_idx = n
45      while left_idx < lsize and right_idx < rsize:
46          if left[left_idx] <= right[right_idx]:
47              tab[curr_idx] = left[left_idx]
48              left_idx += 1
49          else:
50              tab[curr_idx] = right[right_idx]
51              print("Przesunięcie liczby", tab[curr_idx], "na mniejsce", curr_idx)
52              right_idx += 1
53          curr_idx += 1
```

Rysunek 5: Implementacja algorytmu sortowania przez scalanie.

```
55      while left_idx < lsize:
56          tab[curr_idx] = left[left_idx]
57          curr_idx += 1
58          left_idx += 1
59      while right_idx < rsize:
60          tab[curr_idx] = right[right_idx]
61          curr_idx += 1
62          right_idx += 1
```

Rysunek 6: Implementacja algorytmu sortowania przez scalanie.

```
65  def sort_scal(tab, l, r):
66      if r > l:
67          m = int((l + r) / 2)
68          sort_scal(tab, l, m)
69          sort_scal(tab, m + 1, r)
70          scal(tab, l, m, r)
```

Rysunek 7: Implementacja algorytmu sortowania przez scalanie.

2.2.1 Działanie

```
Twoja tablica:
5 | 2 | 8 | 1 |
Wybierz metodę sortowania:
    1 - bąbelkowe,
    2 - przez wybór,
    3 - przez scalanie,
    4 - kubełkowe

Wybieram: 3
Przesunięcie liczby 2 na mniejsce 0
Przesunięcie liczby 1 na mniejsce 2
Przesunięcie liczby 1 na mniejsce 0

Twoja tablica po sortowaniu przez scalanie:
1 | 2 | 5 | 8 |
```

Rysunek 8: Wynik użycia algorytmu sortowania przez scalanie.

2.3 Sortowanie przez wybór (Selection sort)

Algorytm selection sort zaczyna pracę od wyszukania elementu minimalnego zbioru. Następnie element minimalny zostaje umiejscowiony na początku tablicy. W dalszym etapie działania, algorytm w taki sam sposób sprawdza minimalne elementy i ustawia je na kolejnych miejscach tablicy wyjściowej. Algorytm nie korzysta z dodatkowych tablic oraz struktur (sortowanie w miejscu), dzięki czemu nie wymaga dużych zasobów pamięciowych podczas pracy. Nie sprawdza się jednak w przypadku tablic z większą ilością danych.

```
17  def sort_wyb(tab, n):  # n=ilość elementów, tab=lista/tablica
18      for x in range(n-1):
19          minimum = x
20          for j in range(x+1, n):
21              if tab[j] < tab[minimum]:
22                  minimum = j
23          if x != minimum:
24              print("\nKrok ", x+1, ": wstawianie liczby ", tab[minimum], " na pozycje ", x)
25              pom = tab[x]
26              tab[x] = tab[minimum]
27              tab[minimum] = pom
```

Rysunek 9: Implementacja algorytmu sortowania przez wybór.

2.3.1 Działanie

```
Twoja tablica:
5 | 2 | 8 | 1 |
Wybierz metodę sortowania:
    1 - bąbelkowe,
    2 - przez wybór,
    3 - przez scalanie,
    4 - kubełkowe

Wybieram: 2

Krok  1 : wstawianie liczby  1  na pozycje  0

Krok  3 : wstawianie liczby  5  na pozycje  2

Twoja tablica po sortowaniu przez wybór:
1 | 2 | 5 | 8 |
```

Rysunek 10: Wynik użycia algorytmu sortowania przez wybór.

2.4 Sortowanie kubełkowe (Bucket sort)

Sortowanie kubełkowe polega na utworzeniu kubełków w których zliczane są poszczególne elementy tablicy wejściowej. Ilość kubełków jest uzależniona od róznicy wartości największego i najmniejszego elementu zbioru. W końcowej fazie nowa tablica jest tworzona z elementów znajdujących się w kubełkach po posortowaniu. Algorytm działa szybko i sprawnie oraz cechuje go wysoka stabilność, ale nie sprawdza się w przypadku nierównomiernego rozłożenia damych. Dodatkową wadą jest też fakt, że do przeprowadzenia działania potrzebuje określonej wartości minimalnej oraz maksymalnej.

```
73  def minmmax(tab):  # początkowa faza - wyszukiwanie najmniejszego elementu zbioru
74      mmin = tab[0]  # tworzymy zmienne pomocnicze
75      mmax = tab[0]
76      for el in tab:
77          if el > mmax:  # sprawdzamy, zależność zmienenej z pozostałymi elementami zbioru
78              mmax = el
79          if el < mmin:
80              mmin = el
81      array = []
82      array.append(mmin)  # tworzymy tablice ze zmienną pomocniczą
83      array.append(mmax)
84      return array
```

Rysunek 11: Implementacja algorytmu kubełkowego (definiowanie wartości minimalnej/maksymalnej).

```python
def sort_kub(tab, n):
    minmax = minmmax(tab)
    ymin = minmax[0]  # wyszukujemy wartość minimalną i maxymalną zbioru
    ymax = minmax[1]
    buckets = [0] * (ymax - ymin + 1)  # tworzymy kubełki ( ilość kubełków = wartość_max - wartość_min + 1 )
    print("Stworzono kubelki:")
    for x in range(ymax - ymin + 1):
        print("B", (x + ymin), end=" | ")
    print()
    for x in range(n):
        buckets[tab[x] - ymin] += 1  # zliczamy, ile razy dany elment pojawił się w kubełku
    print("Zliczono elementy:")
    for x in range(ymax - ymin + 1):
        print("B", (x + ymin), "=", buckets[x], end=" | ")
    print()
    lastindex = 0
    for x in range(ymax - ymin + 1):  # ustawiamy elemnty kubełków w określonej kolejności (min -> max)
        y = lastindex
        while y < buckets[x] + lastindex:
            tab[y] = x + ymin
            y += 1
        lastindex = y
```

Rysunek 12: Implementacja algorytmu kubełkowego.

2.4.1 Działanie

```
Twoja tablica:
4 | 2 | 1 |
Wybierz metodę sortowania:
    1 - bąbelkowe,
    2 - przez wybór,
    3 - przez scalanie,
    4 - kubełkowe

Wybieram: 4
Stworzono kubelki:
B 1 | B 2 | B 3 | B 4 |
Zliczono elementy:
B 1 = 1 | B 2 = 1 | B 3 = 0 | B 4 = 1 |

Twoja tablica po sortowaniu kubełkowym:
1 | 2 | 4 |
```

Rysunek 13: Wynik użycia algorytmu sortowania kubełkowego.

2.5 Porównanie złożoności obliczeniowej

Zestawienie nazw wyżej wymienionych algorytmów sortowania z ich złożonością obliczeniową w tabeli.

	Najlepszy	Średni	Najgorszy	Pamięć	Czy stabilny
Bubble sort	n	n^2	n^2	1	Tak
Selection sort	n^2	n^2	n^2	1	Nie
Merge sort	$nlogn$	$nlogn$	$nlogn$	n	Nie
Bucket sort	—	$n+r$	$n+r$	$n+r$	Tak

Tabela 1: Zestawienie sortowań: bubble, merge, selection, bucket.

3 Podsumowanie

Udało nam się wykonać założony cel. Zaimplementowałyśmy skutecznie cztery algorytmy sortowań - bąbelkowy, przez scalanie, przez wybór i kubełkowy. Porównałyśmy działanie i złożoność tych algorytmów, potwierdzając początkowe założenia dotyczące nieuniwersalności agorytmów. Należy odpowiednio dobrać model działania w zależności od problemu, z którym mamy do czynienia.

4 Bibliografia

1. https://en.wikipedia.org/wiki/Sorting$_algorithmRadix_sort$

2. https://www.tutorialspoint.com/python$_data_structure/python_big_on$otation.html

3. https://pl.wikipedia.org/wiki/Sortowanie

4. http://math.uni.wroc.pl/ jagiella/p2python/skrypt$_html/wyklad5 - 1.html intro_r$oznice