P O L I T E C H N I K A  R Z E S Z O W S K A

im. Ignacego Łukasiewicza

WYDZIAŁ MATEMATYKI I FIZYKI STOSOWANEJ

Tomasz Nowak

Grupa P05

# Projekt Algorytmy i struktury danych nr. 2

Rzeszów 2022

# 1. Wstęp

Celem projektu było porównanie dwóch algorytmów sortujących, w moim przypadku przyszło mi omówić “Quick Time Sort” oraz “Sortowanie kubełkowe”. Napisany przeze mnie kod zamieszczam na repozytorium Github: https://github.com/Tnovyloo/Projekt-Studia

# 2. Teoria

QuickSort jest algorytmem sortowania opartym na podziale i scalaniu. Jego złożoność obliczeniowa wynosi optymistyczna oraz typowa O(n log n), pesymistyczna zaś O($n^2$). Oznacza to, że czas wykonania algorytmu jest wprost proporcjonalny do ilości elementów do posortowania, ale z logarytmicznym współczynnikiem. QuickSort jest szybki i prosty w implementacji, a także skaluje się dobrze w przypadku dużych zbiorów danych. Jest to jeden z najbardziej popularnych algorytmów sortowania i najlepszy wybór do zastosowań zwykłych.

Sortowanie kubełkowe jest algorytmem sortowania, w którym elementy danych są przydzielane do „kubłów” według wartości. Następnie każdy kubeł jest posortowany za pomocą innego algorytmu sortowania. Złożoność tego algorytmu wynosi O(n + k), gdzie k jest liczbą kubłów. Jest to algorytm skalujący się dobrze w przypadku dużych zbiorów danych. Jest to szczególnie przydatne w przypadku danych, które są wyrażone w jednostkach skalarnych, takich jak liczby całkowite lub zmiennoprzecinkowe. Sortowanie kubełkowe jest czasem lepszy niż QuickSort, gdy elementy danych są “blisko siebie”.

# 3. Schematy blokowe oraz pseudokod

Pseudokod oraz schemat blokowy dla sortowania kubełkowego:

1. Utwórz listę do przechowywania danych.

2. Wybierz wartość podziału jako pierwszy element listy.

3. Utwórz puste koszyki, aby przechowywać elementy mniejsze i większe niż wartość podziału.

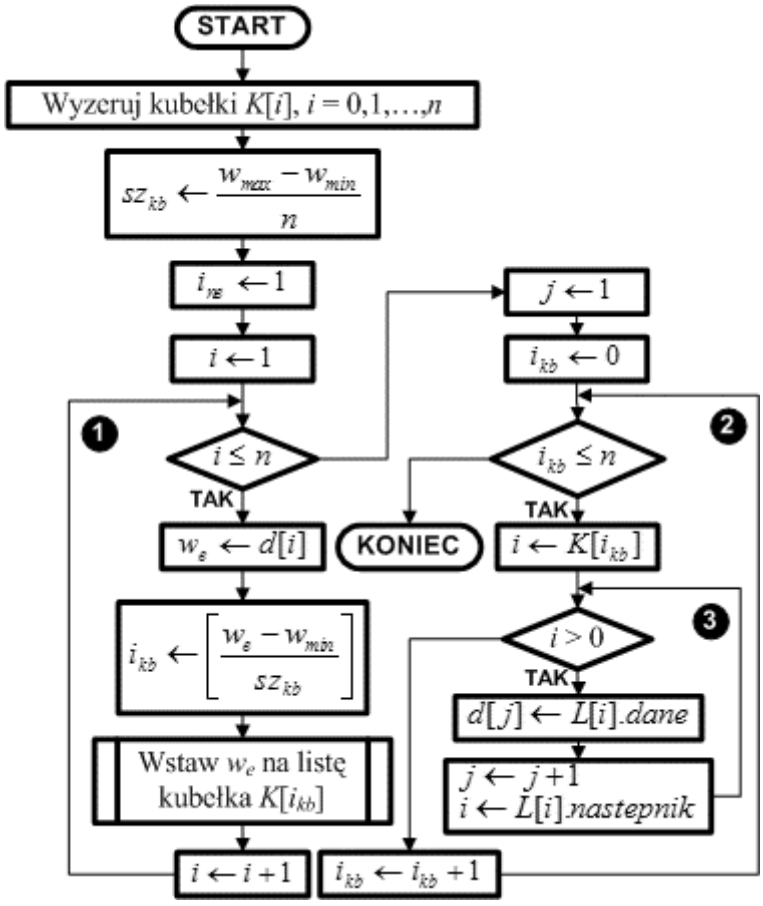
4. Przeiteruj przez listę.

5. Jeśli element jest mniejszy lub równy wartości podziału, dodaj go do koszyka mniejszych elementów.

6. Jeśli element jest większy od wartości podziału, dodaj go do koszyka większych elementów.

7. Uruchom ponownie powyższy algorytm dla każdego koszyka osobno, wybierając następną wartość podziału z listy.

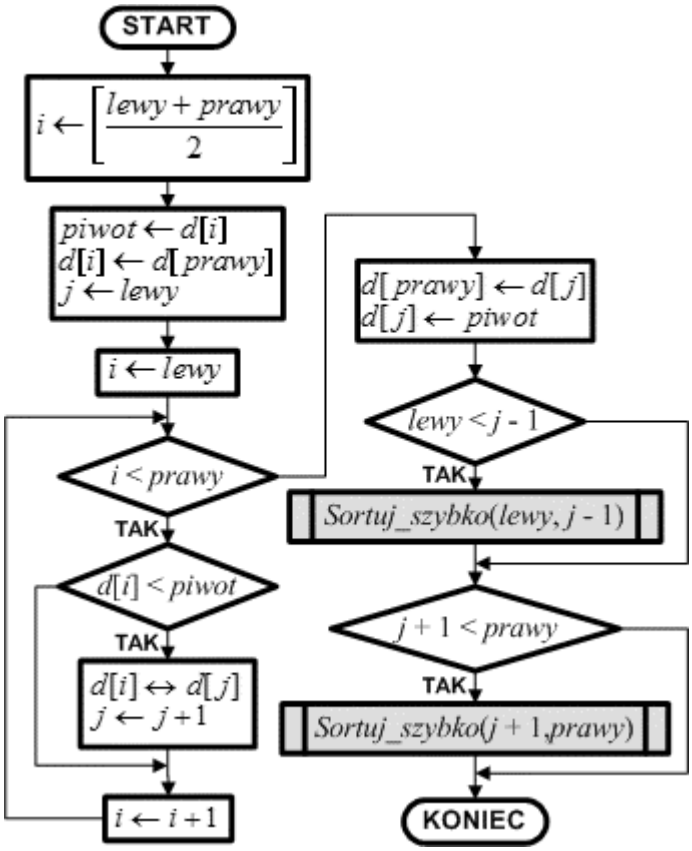
8. Po zakończeniu sortowania kubełkowego, połącz wszystkie elementy w posortowanej liście.




https://eduinf.waw.pl/inf/alg/003_sort/0020.php (mgr Jerzy Wałaszek)

Pseudokod oraz schemat blokowy dla sortowania “QuickSort”:

1. Sprawdź, czy lista jest pusta.
2. Jeśli tak, zakończ działanie.
3. Wybierz element bazowy jako pierwszy element listy.
4. Ustaw dwa wskaźniki na początek i koniec listy.
5. Ustaw wskaźnik pivot na wybrany element bazowy.
6. Ustaw wskaźnik i na pozycji początkowej.
7. Ustaw wskaźnik j na pozycji końcowej.
8. Porównaj elementy od i i j.
    a. Jeśli element i jest mniejszy lub równy od pivot, zwiększ wskaźnik i o 1.
    b. Jeśli element j jest większy l




https://eduinf.waw.pl/inf/alg/003_sort/0018.php (mgr Jerzy Wałaszek)

# 4. Praktyka oraz testy

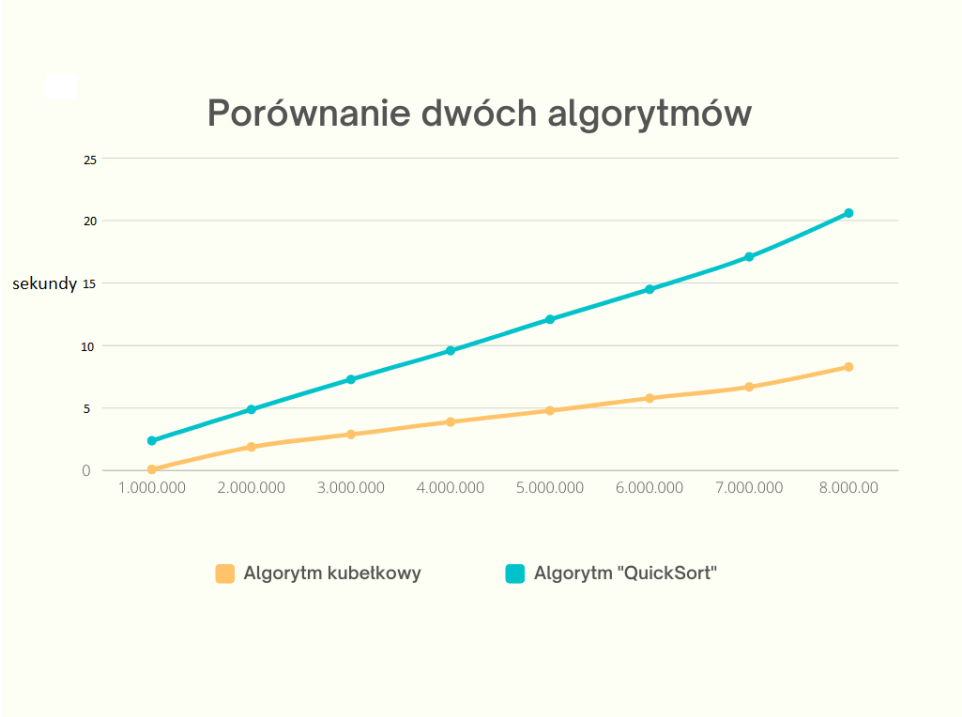
Testy napisanego kodu na githubie. (Funkcja “test2()”)

```
[39, 1, 25, 35, 6] - Przykładowa tablica
[1, 6, 25, 35, 39] - sortowanie BucketSort
[1, 6, 25, 35, 39] - sortowanie QuickSort
```

Testowanie złożoności czasowej dwóch algorytmów (Funkcja “test()”):

```
Czas sortowania dla 1000000 elementów
Kubełkowy: 0.1s
Quick Sort: 2.48s
Czas sortowania dla 2000000 elementów
Kubełkowy: 0.19s
Quick Sort: 4.9s
Czas sortowania dla 3000000 elementów
Kubełkowy: 0.29s
Quick Sort: 7.3s
Czas sortowania dla 4000000 elementów
Kubełkowy: 0.39s
Quick Sort: 9.68s
Czas sortowania dla 5000000 elementów
Kubełkowy: 0.48s
Quick Sort: 12.16s
```

Przedstawienie wyników na wykresie:




Jak widać na załączonym obrazku oraz wykresie, algorytm sortujący Quick Sort jest dużo wolniejszy na większej ilości danych w tablicy. Wychodzi to z jego natury czyli złożoności obliczeniowej $O(n \log n)$.

# 5. Podsumowanie

Algorytm kubełkowy sprawdził się dużo lepiej w sortowaniu liczb całkowitych w tablicy. Spostrzegam, że jedną z wad Quick Sortu jest rekurencyjność, ponieważ rekurencja nie daję sobie tak dobrze rady przy dużej ilości danych, gdyż wynika to z natury funkcji rekurencyjnych.