Algorytmy Sortowania

Damian Jakubowksi

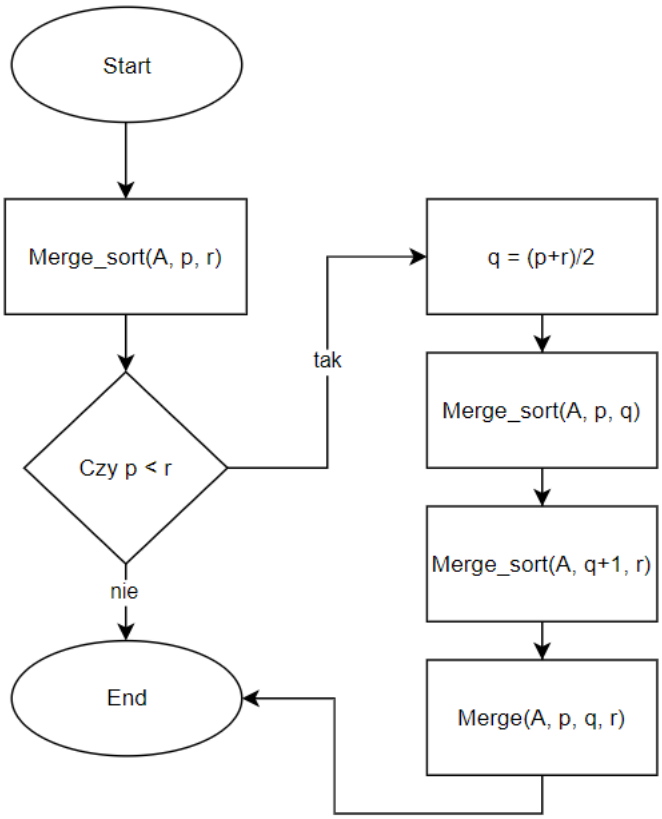
21 kwietnia 2023

Spis treści

1 Przedstawienie omawianych algorytmów

1.1 Sortowanie przez scalanie

Jest to rekurencyjny algorytm sortowania danych implementujacy metode dziel i zwyciężaj. Algorytm wywlołuje się rekurencyjnie na coraz to mnieszym zakresie tablicy, dojdzie do momentu gdy sortuje tylko jeden element następnie metoda Merge odpowiednio wstawia elementy na poprawne miejsca z całego zakresu tablicy.



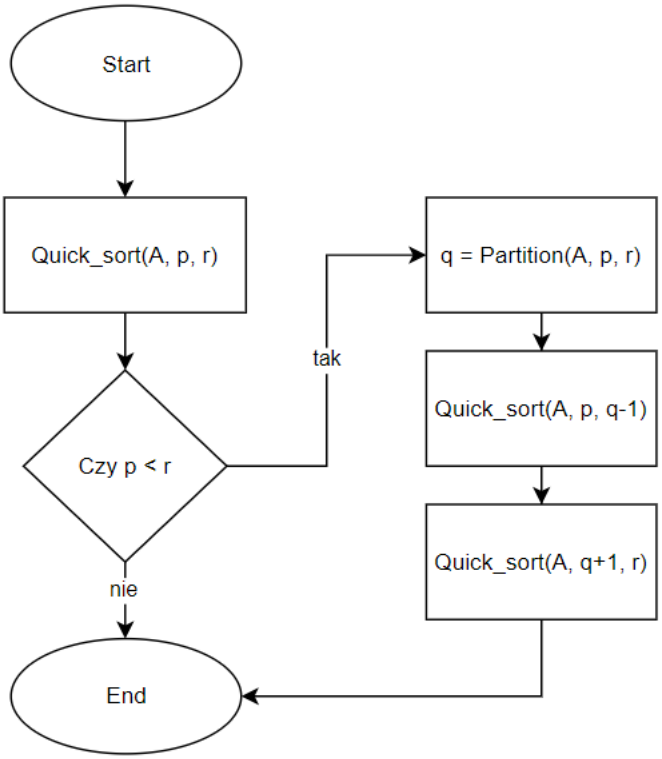
Rysunek 1: Schemat blokowy algorytmu merge sort.

1.2 Sortowanie szybkie

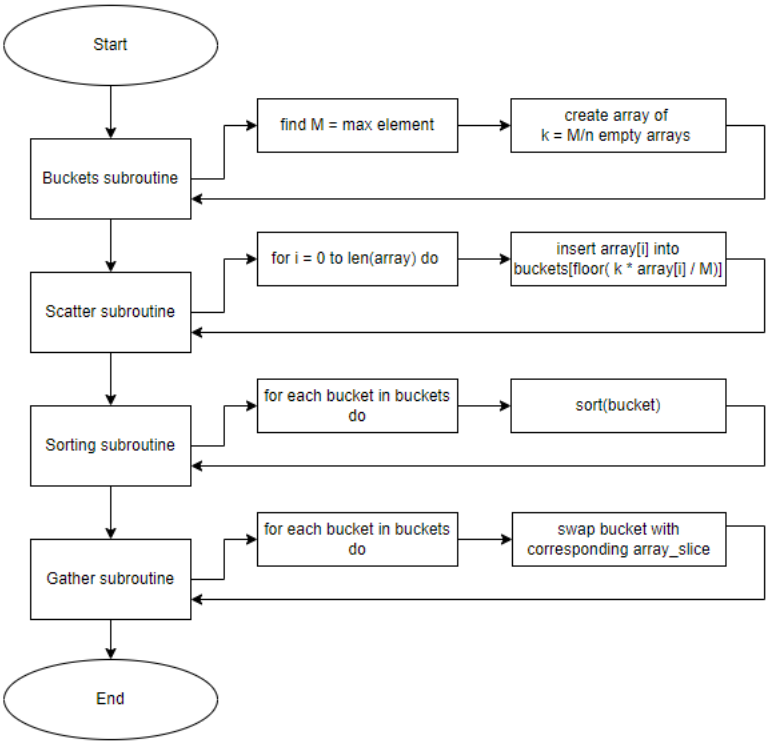
Również jest to algorytm implementujący metode dziel i zwyciężaj. Jest to usprawniony merge sort. Algorytm wywołuje się rekurencyjnie wokół jakiegoś elementu zazwyczaj środkowego jednak istnieją różne metody wyboru tzw. pivota. Za pomocą metody Partition algorytm zamienia stronami elemnety z lewej na prawa i odwrotnie gdy jest on wiekszy lub mniejszy od pivota. Dzieje się to do momentu gdy nie zostanie jeden element do przerzucenia. Jak widać na schemacie (rys.2) takie podejście niweluje potrzebę scalenia tablicy w osobnym przejściu.

1.3 Sortowanie kubełkowe

Zgodnie ze schematem (rys.3) algorytm rozpoczyna swoje działanie od utworzenia tzw. kubłów (ang. bucket), następnie elementy do posortowania są dodawane do konkretnych kubłów zgodnie z zakresem do którego element należy. Działanie metody Scatter subroutine można porównać do tworzenia histogramu danych. Następnie każdy kubeł jest sortowany oddzielnie przez inny algorytm, zazwyczaj dobierany tak aby sprawdzał sie na danych które sortujemy lub sortował szybko odpowiednie ilości elemntów w kubłach. Po wykonaniu sortowania wszytkie kubły są opróżnianie i wpisywane kolejno do tablicy, ponieważ wiemy ,że każdy kubeł jest posortowany i każdy następny zawiera dane odpowiednio większe lub mniejsze od poprzedniego kubła. Złożoność tego algorytmu jest zależna od algorytmu zastosowanego do sortowania poszczególnych kubłów od ich ilości oraz od tego czy dane zachowują rozkład normalny czy nie.



Rysunek 2: Schemat blokowy algorytmu quick sort.



Rysunek 3: Schemat algorytmu bucket sort.

2 Porównanie omawianych algorytmów

2.1 Przypadek pesymistyczny

- merge sort - $O(nlog(n))$,
- quick sort - $O(n^2)$ wystąpi gdy pivotem cały czas będzie element największy,
- bucket sort - $O(n^2)$ wystapi gdy wszystkie elementy trafią do jednego kubła.

2.2 Przypadek średni

- merge sort - $O(nlog(n))$,
- quick sort - $O(nlog(n))$,
- bucket sort - $O(n+k)$.

2.3 Przypadek optymistyczny

- merge sort - $O(nlog(n))$,
- quick sort - $O(nlog(n))$ wystąpi gdy pivotem cały czas będzie element środkowy,
- bucket sort - $O(n+k)$ elementy będą równomiernie rozłożone między kubły.

3 Omówienie badanych danych

Dane które bądą sortowane to baza danych z ocenami filmów ze strony Kaggle.com. Od razu możemy założyc, że będzie zachowany rozkład normalny więc bucket sort powinien sprawdzić się przynajmniej dobrze. Wartość maksymalna to 10 a minimalna wystepująca w zbiorze to 1.

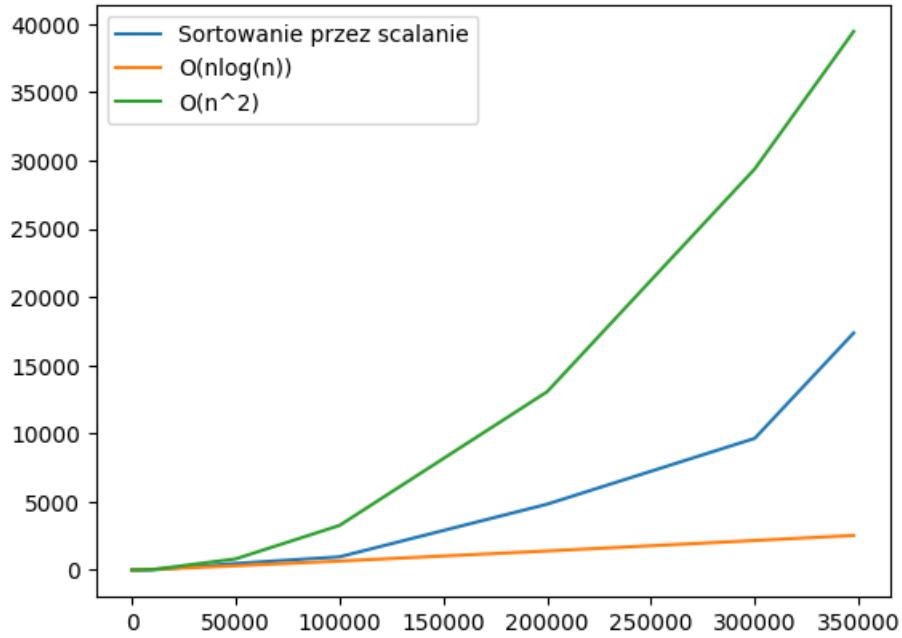
lp.	movie	rating
10	Closure (I) (2018)	9.0
11	Unstoppable (2010)	8.0
12	Dangerous Lies (2020)	7.0
13	Beastie Boys Story (2020)	7.0
14	"Ruben Brandt Collector (2018)"	7.0
15	Some Kind of Hate (2015)	3.0
16	Cube Zero (2004)	8.0
17	Carne (1991)	7.0
18	500 Days of Summer (2009)	5.0

Tabela 1: Fragment danych przeznaczonych do sortowania.

4 Badanie poszczególnych algorytmów

4.1 Sortowanie przez scalanie

Złożoność czasowa



Rysunek 4: Złożoność optymistyczna i pesymistyczna w porównaniu do rzeczywistej

Złożoność pamięciowa

```
while (i < mid + 1 && j < end + 1) {
    if (list[i] <= list[j]) {
      i++;
    } else {
      int val = list.removeAt(j);
      list.insertAt(i, val);
      i++;
      j++;
      mid++;
    }
```

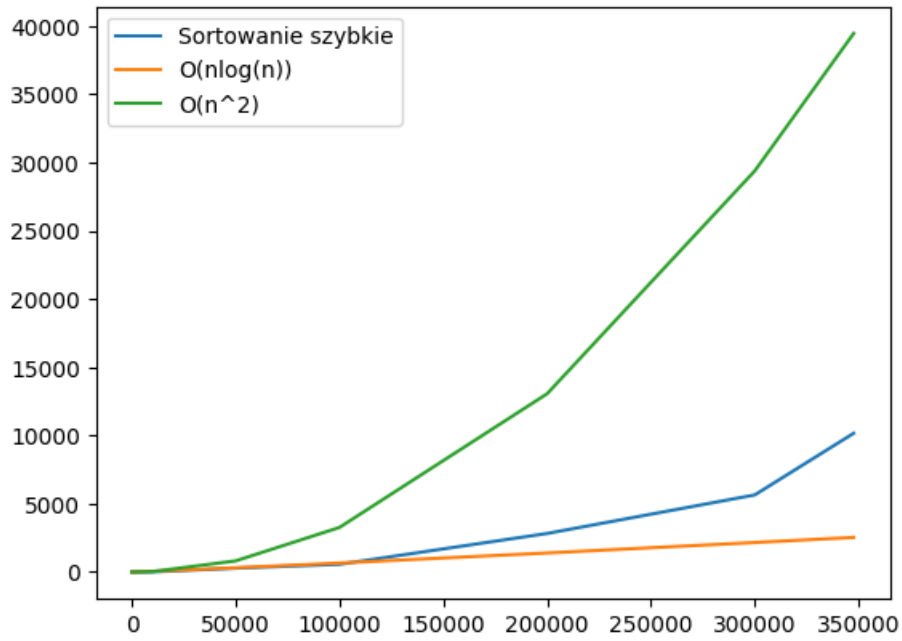
Algorytm jest zaimplementowany w miejscu więc fragment odpowiedzialny za przepisywanie elementów na odpowiednie pozycje pracuje na oryginale (nie alokuje dodatkowych zasobów) co daje złożoność pamięciową O(1).

Wnioski

Mergesort z pośród trzech badanych algorytmów poradził sobie najgorzej co też wynika z teorii mimo, że złożoności obliczeniowe są te same. W związku, że algorytm jest rekurencyjny pojawiło się duże zurzycie pamięci.

4.2 Sortowanie szybkie

Złożoność czasowa



Rysunek 5: Złożoność optymistyczna i pesymistyczna w porównaniu do rzeczywistej

Złożoność pamięciowa

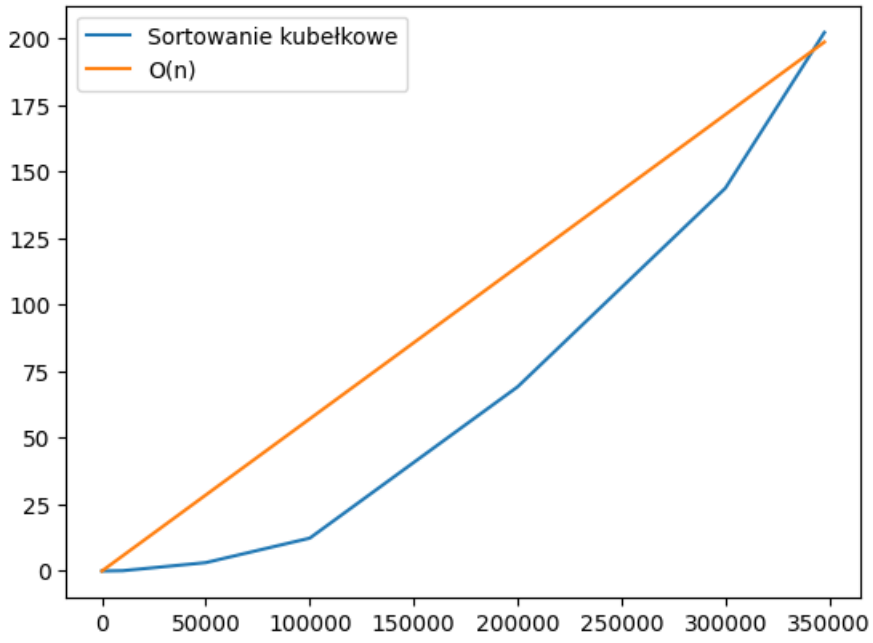
```
int _partition(DoubleLinkedList<int> list, int left, int right){
  int x = list[right];
  int i = left-1;
  for (int j = left; j <= right-1; j++){
    if (list[j] < x){
      i++;
      list.swapValue(i, j);
    }
  }
  list.swapValue(i+1, right);
  return i+1;
}
```

Metoda partition zaimplementowana jest również w miejscu i można to zauważyć po fragmencie kodu wyżej. W funckji używany jest oryginał sortowanej listy co nie powoduje zbędnej alokacji pamięci więc złożoność pamięciowa algoprytmu wynosi O(1).

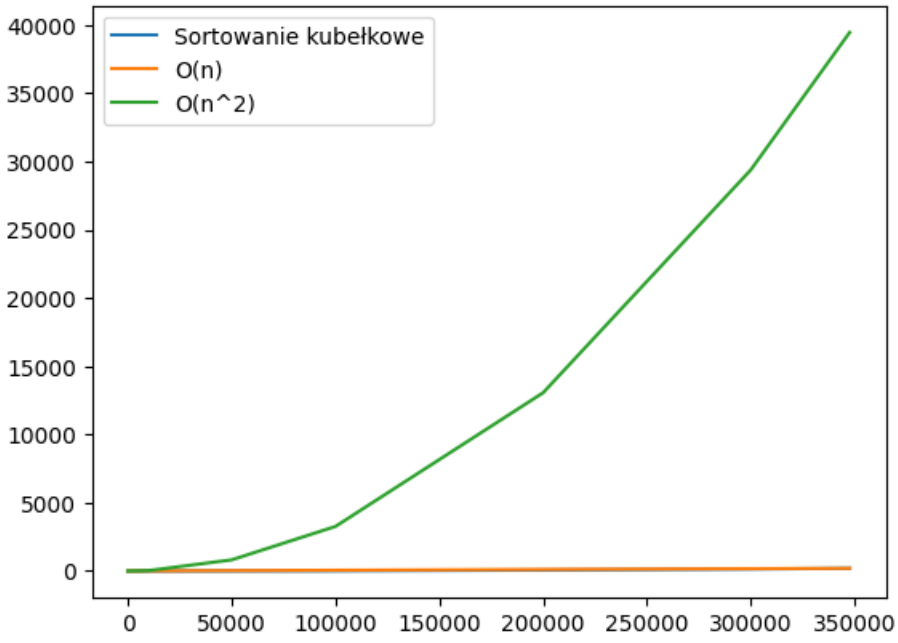
Wnioski

Alogrytm sortowania szybkiego poradził sobie lepiej niż sortowanie przez scalanie co również jest wynikiem przewidywanym, ponieważ quick sort to tak naprawde usprawniony merge sort.

4.3 Sortowanie kubełkowe



Rysunek 6: Złożoność optymistyczna w porównaniu do rzeczywistej



Rysunek 7: Złożoność pesymistyczna w porównaniu do rzeczywistej

Złożoność pamięciowa

```
for (int i = 0; i < startSize; i++) {
    var popedItem = list.popBack();
    var index = (k * popedItem ~/ m).floor();
    if (index >= k) {
      buckets.last.pushBack(popedItem);
      continue;
    }
    buckets.elementAt(index)!.pushBack(popedItem);
  }
```

Zgodnie z fragmentem który jest odpowiedzialny za rozrzucenie elementów do odpowiednich kubłów element najpierw jest usuwany z oryginału a następnie przerzucany do kubła w związku z tym elementy nie są kopiowane więc nie jest potrzebne więcej miejsca aby je przechować co daje złożoność pamięciową O(1).

```
T? popBack() {
    Node<T> popedNode = this._tail!;
    this._tail = _tail!.prev;
    this._tail!.next = null;
    this._size--;
    return popedNode.value;
  }
```

Metoda popBack() przepina wskaźnik tail na przedostatni element i zwraca jego wartość oraz usuwa referencje na ostatni element tak aby trash collector mógł zwolnic jego zasoby.

Wnioski

Tak jak zakładano w Punkcie 3 sortowanie kubełkowe poradziło sobie z taką ilością danych zgodnie z przewidywaną złożonością i najlepiej wśród badanych algorytmów. Jednym z powodów może być fakt, że sortowane dane spełniają warunki możliwości wytąpienia najlepszego przypadku bucket sorta.

5 Podsumowanie

Osiągnięte wyniki są zgodne z teoretycznymi obliczeniami złożoności i również udało się osiągnąć złożonośc pamięciową równą O(1). Cały projekt bardzo dobrze obrazuje, że ważne jest by dopasować algorytm do rodzaju danych lub ich ilości.