Raport AiSD lista 2

Amelia Dorożko

15 grudnia 2024

1 Sortowanie szybkie

1.1 `QUICK_SORT`

Algorytm `QUICK_SORT` polega na wyborze elementu zwanego pivotem, który jest ostatnim elementem tablicy. Następnie za pomocą `PARITION` dzieli tablicę na dwie części: elementy mniejsze od pivota znajdują się po jego lewej stronie, a większe po prawej. Po podziale rekurencyjnie sortuje obie podtablice. Średnia złożoność czasowa tego algorytmu wynosi $O(n \log n)$. Jednak w najgorszym przypadku, gdy podział jest bardzo nierówny (np. dla tablicy posortowanej), czas działania wzrasta do $O(n^2)$.

1.2 `QUICK_SORT`-z dwoma pivotami

`QUICK_SORT2` to modyfikacja, która wykorzystuje dwa pivoty zamiast jednego. Tablica dzielona jest na trzy części: elementy mniejsze od pierwszego pivota, elementy pomiędzy dwoma pivotami oraz elementy większe od drugiego pivota. Podczas `PARTITION2` algorytm najpierw porównuje oba pivoty i w razie potrzeby zamienia je miejscami, aby pierwszy pivot był mniejszy od drugiego. Następnie elementy są przypisywane do odpowiednich części: na początek (mniejsze niż pierwszy pivot), na koniec (większe niż drugi pivot) lub pozostają między pivotami. Po partycjonowaniu algorytm rekurencyjnie sortuje każdą z trzech części tablicy.

`QUICK_SORT2` to modyfikacja, która wykorzystuje dwa pivoty zamiast jednego. Tablica dzielona jest na trzy części: elementy mniejsze od pierwszego pivota, elementy pomiędzy dwoma pivotami oraz elementy większe od drugiego pivota. Podczas `PARTITION2` algorytm najpierw porównuje oba pivoty i w razie potrzeby zamienia je miejscami, aby pierwszy pivot był mniejszy od drugiego. Następnie elementy są przypisywane do odpowiednich części: na początek (mniejsze niż pierwszy pivot), na koniec (większe niż drugi pivot) lub pozostają między pivotami. Po partycjonowaniu algorytm rekurencyjnie sortuje każdą z trzech części tablicy.

Wykorzystanie dwóch pivotów zmniejsza liczbę poziomów rekurencji. Dzięki temu algorytm staje się bardziej wydajny pod względem liczby porównań i przypisań, szczególnie w przypadku dużych, zróżnicowanych zbiorów danych.

Długość tablicy	QS:porównania	QS: przypisania	QS2: porównania	QS2: przypisania
10	38	36	39	58
100	802	744	530	682
200	1725	1794	1149	1558
400	4002	3762	2659	3264
600	6621	6866	4088	5424
800	9262	9674	6325	8524
1000	11873	11104	7942	11026

Tabela 1: Liczba porównań i przypisań dla `QUICK_SORT` i `QUICK_SORT2` dla liczb z zakresu [-100,100]

Wprowadzenie podwójnego pivota optymalizuje proces sortowania i zmniejsza ryzyko degeneracji algorytmu, co czyni go bardziej wszechstronnym w praktyce.

2 Sortowanie przez zliczanie i sortowanie pozycyjne

2.1 `COUNTING_SORT`

Algorytm ten wykorzystuje tablicę pomocniczą do zliczania wystąpień poszczególnych elementów w zbiorze danych. Jest to algorytm stabilny i działa w czasie liniowym *O(n)*. Dla każdego elementu w tablicy A obliczana jest cyfra na określonej pozycji. Cyfra ta jest obliczana przy użyciu wyrażenia:

$$\left(\frac{A[i]}{k}\right)\%d$$

gdzie:

- k to miejsce,
- d to liczba cyfr (w systemie dziesiętnym jest to 10),
- $A[i]$ to element tablicy A.

Wartości cyfr są zliczane w tablicy pomocniczej C.

2.2 `RADIX_SORT`

`RADIX_SORT` to algorytm sortowania oparty na metodzie sortowania pozycyjnego. Zamiast porównywać liczby bezpośrednio, sortuje dane w kolejnych krokach, zaczynając od najmniej znaczącej cyfry (cyfrach jedności), a kończąc na najbardziej znaczącej cyfrze.Używa `COUNTING_SORT` do sortowania cyfr na kolejnych miejscach w liczbach

2.3 RADIX_SORT-dla liczb ujemnych

Ta modyfikacja algorytmu Radix Sort umożliwia sortowanie liczb zarówno dodatnich, jak i ujemnych. Przed rozpoczęciem sortowania, dane są dzielone na liczby dodatnie i ujemne. Liczby ujemne są przechowywane w osobnej tablicy, a następnie przekształcone na liczby dodatnie. Następnie, takie dwie tablice są sortowane przy użyciu klasycznego RADIX_SORT.

```
1
2  void RADIX_SORT_WITH_NEGATIVES(int A[], int n, int d) {
3      int negative_count = 0, positive_count = 0;
4      for (int i = 0; i <= n; i++){
5          if (A[i]<0){
6              negative_count++;
7              liczbaPorownan++;
8          }
9          if (A[i]>0){
10             positive_count++;
11             liczbaPorownan++;
12         }
13     }
14     int negatives[negative_count], positives[positive_count];
15     int neg_index = 0, pos_index = 0;
16
17     for (int i = 0; i < n; i++) {
18         if (A[i] < 0) {
19             negatives[neg_index++] = -A[i];
20             liczbaPrzypisan++;
21         } else {
22             positives[pos_index++] = A[i];
23             liczbaPrzypisan++;
24         }
25         liczbaPorownan++;
26     }
```

Podstawa d wpływa również na rozmiar bucketów, do których będą trafiały elementy w trakcie sortowania. Zwiększenie liczby grup (w przypadku większej podstawy d) może pomóc w szybszym rozdzieleniu elementów, ale z drugiej strony wiąże się z większymi wymaganiami pamięciowymi. Zmiana podstawy może wpłynąć na efektywność przestrzenną i czasową algorytmu. Większa podstawa może zmniejszyć liczbę iteracji. Idzie zauważyć, że wzrost ten jest liniowy.

Długość tablicy	Podstawa 2	Podstawa 8	Podstawa 10	Podstawa 16
5	Porównania: 12 Przypisania: 220	Porównania: 13 Przypisania: 144	Porównania: 13 Przypisania: 137	Porównania: 13 Przypisania: 173
10	Porównania: 23 Przypisania: 1061	Porównania: 23 Przypisania: 276	Porównania: 23 Przypisania: 207	Porównania: 18 Przypisania: 151
100	Porównania: 203 Przypisania: 4261	Porównania: 203 Przypisania: 1873	Porównania: 202 Przypisania: 1454	Porównania: 202 Przypisania: 1221
200	Porównania: 403 Przypisania: 8461	Porównania: 403 Przypisania: 3673	Porównania: 403 Przypisania: 2867	Porównania: 402 Przypisania: 2331
400	Porównania: 802 Przypisania: 16820	Porównania: 803 Przypisania: 7273	Porównania: 803 Przypisania: 5667	Porównania: 803 Przypisania: 4654
600	Porównania: 1203 Przypisania: 22261	Porównania: 1203 Przypisania: 10873	Porównania: 1202 Przypisania: 8454	Porównania: 1202 Przypisania: 6987
800	Porównania: 1603 Przypisania: 33661	Porównania: 1602 Przypisania: 23851	Porównania: 1603 Przypisania: 12848	Porównania: 1602 Przypisania: 9662

Tabela 2: Porównanie wyników dla różnych podstaw w Radix Sort z danymi dodatnimi i ujemnymi

3 Sortowanie kubełkowe

3.1 `BUCKET_SORT`

Algorytm `BUCKET_SORT` jest sortowaniem rozdzielającym, które polega na podzieleniu danych wejściowych na "kubełki", a następnie sortowaniu ich indywidualnie za pomocą innego algorytmu sortowania (zwykle `INSERTION_SORT`). Po posortowaniu elementów w kubełkach, wyniki są scalane z powrotem w jedną posortowaną tablicę.
Pierwsza wersja algorytmu zakłada, że dane wejściowe są liczbami zmiennoprzecinkowymi mniejszymi od 1 i większymi, bądź równymi zero.

3.1.1 BUCKET_SORT dla dowolnych liczb

Druga wersja algorytmu została zmodyfikowana, aby działała dla liczb **dowolnego przedziału**. Zakłada się, że dane mogą pochodzić z dowolnego przedziału liczbowego.

- **Znalezienie min i max:** Pierwszym krokiem jest znalezienie minimalnej (`minValue`) i maksymalnej (`maxValue`) wartości w tablicy. Dzięki temu możemy obliczyć zakres danych.
- **Przypisanie do kubełków:** Następnie dane są rozdzielane na kubełki w sposób, który uwzględnia minimalną i maksymalną wartość.

  gdzie range to różnica między największą a najmniejszą wartością w tablicy. Dzięki temu każdemu elementowi przypisywana jest odpowiednia pozycja w jednym z kubełków.

```
1
2
3       void BUCKET_SORT2(double A[], int n) {
4           double minValue = A[0];
5           double maxValue = A[0];
6
7           for (int i = 1; i < n; i++) {
8               if (A[i] < minValue) {
9                   minValue = A[i];
10              }
11
12              if (A[i] > maxValue) {
13                  maxValue = A[i];
14              }
15          }
16
17          double range = maxValue - minValue;
18          List B[n];
19          for (int j = 0; j < n; j++) {
20              B[j] = List();
21
22          }
23
24          for (int i = 0; i < n; ++i) {
25              int bucketIndex = static_cast<int>(((A[i] -
                    minValue) / range) * n);
26              if (bucketIndex == n) bucketIndex--;
27              LIST_INSERT(B[bucketIndex], new Node(A[i]));
```

- Dalej kod działa tak jak dla "normalnego" `BUCKET_SORT`, czyli sortujemy liczby w kubełkach używając `INSERTION_SORT`, a następnie scalając.

3.2 Złożoność algorytmu

Złożoność czasowa algorytmu zależy od kilku czynników, takich jak liczba kubełków, liczba elementów w tablicy oraz wybrany algorytm do sortowania danych w kubełkach. Złożoność dla `BUCKET_SORT` w najlepszym przypadku wynosi

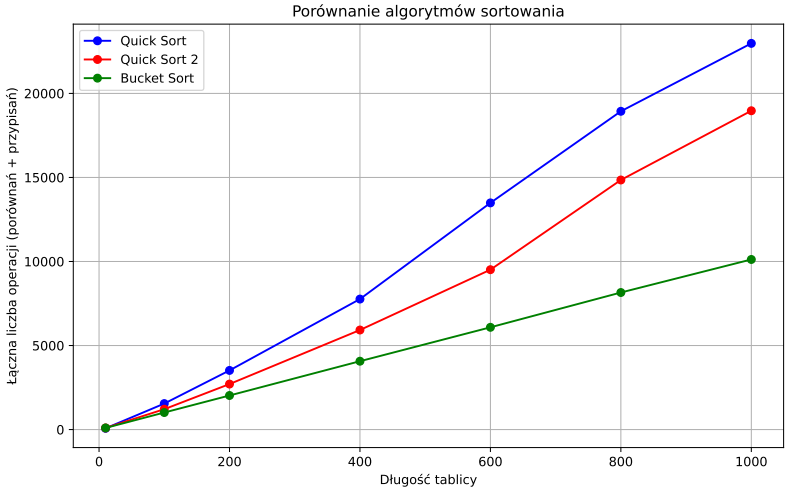
$O(n)$, ale w najgorszym przypadku może osiągnąć $O(n^2)$, szczególnie gdy dane są nierównomiernie rozłożone w kubełkach.

4 `INSERTION_SORT` na listach

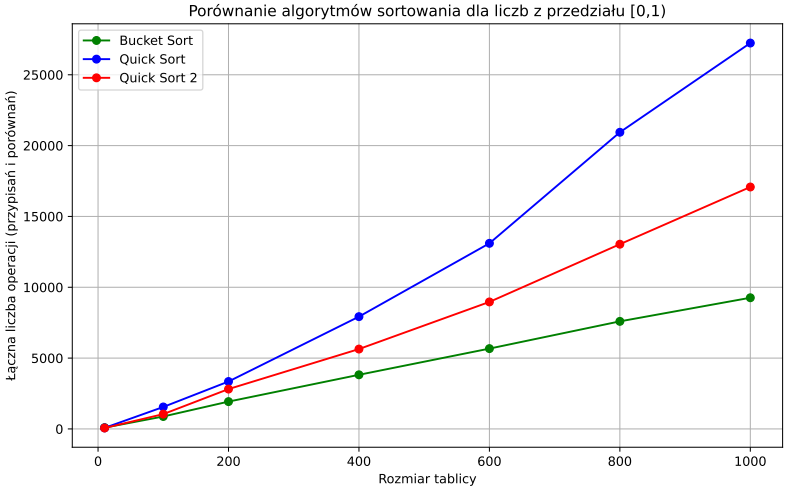
Warto zwrócić uwagę, że algorytm wykorzystany przez `BUCKET_SORT`, który używany jest do posortowania elementów wewnątrz poszczególnych bucketów. W kontekście implementacji na listach dwukierunkowych, `INSERTION_SORT` operuje bezpośrednio na węzłach, co pozwala na sprawne zarządzanie strukturą listy.Jest on skuteczny dla małych zbiorów danych, w szczególności w kubełkach.

4.1 Porównanie wydajności algorytmów

`QUCIK_SORT` jest bardziej wszechstronny, natomiast `BUCKET_SORT` działa najlepiej, gdy dane są równomiernie rozłożone w małym zakresie. Tutaj porównujemy działanie algorytmów dla liczb z przedziału [0,1):



Porównując dane z mniej rozłożonego zakresu liczb, widzimy, że `QUICK_SORT` jest również w tym przypadku najmniej efektywnym algorytmem.



`BUCKET_SORT` działa najlepiej, gdy dane są równomiernie rozłożone. Jeśli dane są bardzo skupione w jednym przedziale , czas działania zbliża się do $O(n^2)$, ponieważ sortowanie w poszczególnych bucketach wymaga więcej operacji. Wymaga on pamięci na dodatkowe struktury. Tworzenie, zarządzanie oraz usuwanie tych struktur jest kosztowne czasowo. Dla danych o nieprzewidywalnym rozkładzie, `QUICK_SORT2` jest bardziej niezawodnym i szybszym wyborem. Dlatego w tym przypadku, `BUCKET_SORT` jest najbardziej efektywnym algorytmem.