sprawozdanie lista 2

Jakub Knaspki

December 2025

1 Wprowadzenie

Sprawozdanie przedstawia analizę algorytmów sortujących: Quick Sort oraz Bucket Sort. Analiza obejmuje porównanie liczby operacji (przypisań i porównań) oraz ich modyfikacji.

2 QUICK SORT

Quick Sort to algorytm sortowania działający metodą dziel i zwyciężaj. Polega na wyborze elementu pivot z tablicy, a następnie podziale pozostałych elementów na dwie części: mniejsze od pivot i większe od pivot. Następnie algorytm rekurencyjnie sortuje te dwie części, aż cała tablica będzie uporządkowana.

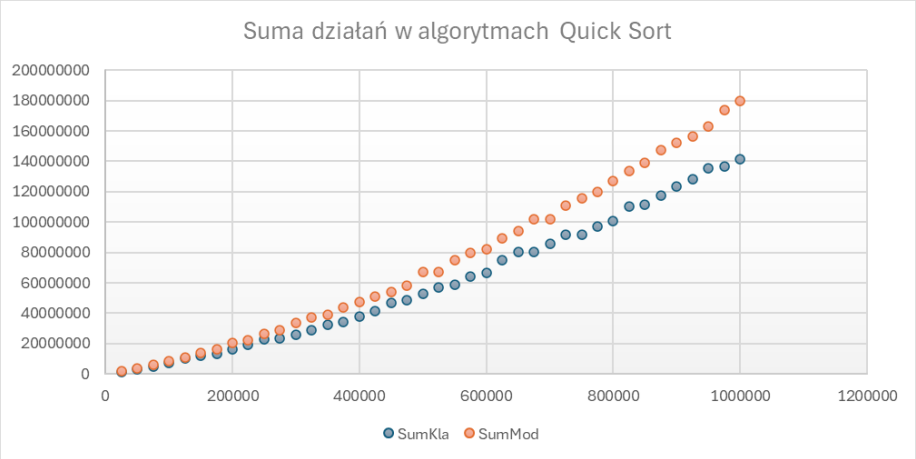
Modyfikacja z podwójnym wyborem 2 pivot'ów:
Wprowadzona modyfikacja algorytmu Quick Sort wykorzystuje dwa pivoty, które dzielą tablicę na trzy części: elementy mniejsze od mniejszego pivota, elementy pomiędzy pivotami oraz elementy większe od większego pivota. W przedstawionej implementacji pivoty są dobierane w taki sposób, aby po wykonaniu pojedynczego przebiegu procedury Partition oba pivoty znajdowały się na swoich końcowych pozycjach, a tablica była podzielona na trzy wymienione przedziały. Poniżej szczegółowo opisano proces partycjonowania dla poszczególnych przypadków.

- **Przypadek 1: element mniejszy lub równy obu pivotom** Element zostaje przesunięty do sekcji lewej, gdzie gromadzone są wszystkie wartości mniejsze od lewego pivota. Ta operacja buduje początkowy segment tablicy, który będzie zawierał najmniejsze elementy. Dzięki temu przygotowujemy tablicę do rekurencyjnego sortowania w kolejnych krokach algorytmu.

- **Przypadek 2: element znajduje się pomiędzy pivotami**
  Element pozostaje w sekcji środkowej, która obejmuje wartości pośrednie pomiędzy pivotami. Brak zamiany oznacza, że element już znajduje się we właściwej części tablicy. Przesunięcie wskaźnika j rozszerza granicę tej sekcji, przygotowując ją do dalszych operacji partycjonowania.

- **Przypadek 3: element większy od obu pivotów**
  Element trafia do sekcji prawej, przeznaczonej dla wartości większych od prawego pivota. Zamiana przenosi go na koniec przetwarzanego zakresu, gdzie stopniowo formowana jest sekcja największych liczb. Wskaźnik j nie jest przesuwany, co pozwala na ponowną weryfikację nowo wstawionej wartości w następnej iteracji.

```
while (j<=m) {
        if (A[j]<x) {
            zamien(A,i,j);
            i+=1;
            j+=1;
        }
        else {
            if (A[j]>y) {
                zamien(A,j,m);
                m-=1;
            }
            else {
                j+=1;
            }
        }
    }
```



2.1 Porównanie modyfikacji algorytmu z jego oryginałem

Na poniższym wykresie widać, że zmodyfikowana wersja algorytmu *quicksort* wykonuje więcej operacji (przypisań oraz porównań) niż jego wersja podstawowa. Oznacza to, że moja modyfikacja jest mniej optymalna. Średnia złożoność czasowa klasycznego algorytmu quicksort wynosi $O(n \log n)$, natomiast w najgorszym przypadku dla danych już posortowanych osiąga złożoność $O(n^2)$.

3 RADIX SORT

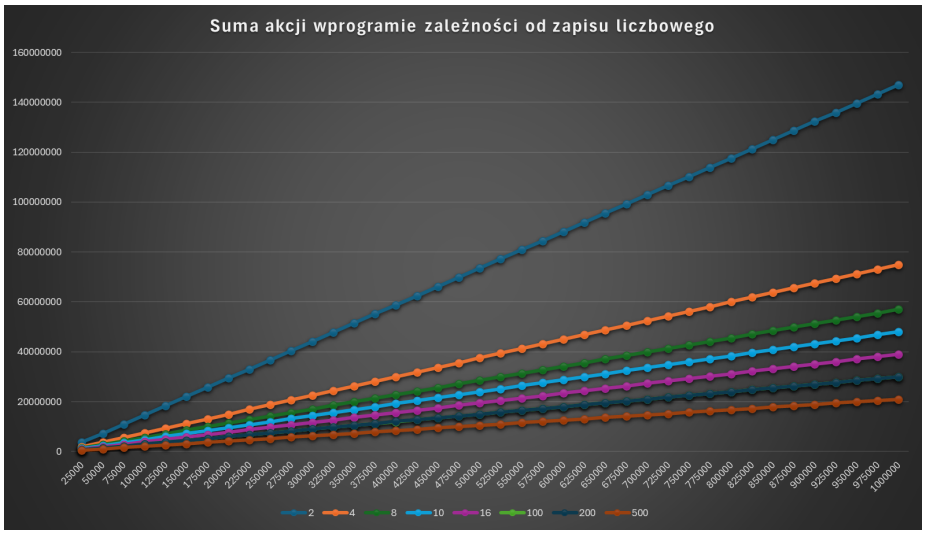
Radix sort polega na porządkowaniu liczb, analizując je cyfra po cyfrze, zwykle zaczynając od najmniej znaczącej cyfry. Na każdym etapie elementy są grupowane według aktualnie rozpatrywanej cyfry, najczęściej przy użyciu sortowania przez zliczanie jako procedury pomocniczej. Po przejściu przez wszystkie cyfry liczby są już całkowicie uporządkowane.
Wiele algorytmów potrafi działać wyłącznie na jednym, określonym typie liczbowym, natomiast Radix Sort może pracować na różnych rodzajach liczb, ponieważ wyodrębnia ich cyfry niezależnie od konkretnego typu danych.

```
for (int i = n - 1; i >= 0; i--)
{
    int cyf = (A[i] / roz) % d;
    B[C[cyf + d - 1] - 1] = A[i];
    C[cyf + d -1]-=1;
    przypisania += 3;
    porownania += 1;
}
```

Ten fragment odpowiada za stabilne układanie elementów w jednym przebiegu Radix Sort. Najpierw wyciągana jest cyfra cyf z elementu `A[i]`, a potem na jej podstawie element zostaje wpisany na właściwe miejsce w tablicy B z pomocą tablicy zliczeń C. Na końcu zmniejszany jest licznik w C, żeby następne elementy z tą samą cyfrą trafiły w kolejne poprawne miejsca.

3.1 Jaka jest zależność pomiędzy typami cyfr

Jak już wspomniałem, Radix Sort może sortować wartości zapisane w różnych systemach liczbowych, nie tylko dziesiętnym. Poniższy wykres pokazuje, jak zmienia się efektywność algorytmu w zależności od wielkości podstawy (typu cyfr).



Z wykresu widać, że im większa podstawa systemu, tym algorytm działa bardziej optymalnie. Wiemy, że ta zależność utrzymuje się co najmniej do wartości około 500, więc jeśli chcemy zminimalizować liczbę przypisań i porównań, opłaca się wybrać większy typ cyfr używany w sortowaniu.
Jeśli chcemy, aby algorytm działał również dla liczb ujemnych, konieczne jest utworzenie dwa razy większego zakresu cyfr (typów zapisu) -1 (bo zero się nie powtarza), czyli jednej dłuższej tablicy obejmującej zarówno wartości ujemne, jak i dodatnie. W klasycznym Radix Sort indeksowanie zaczyna się od 0, natomiast w tej wersji startujemy od $-d+1$, aby poprawnie działały cyfry reprezentujące liczby ujemne.
Średni i najgorszy czas działania Radix Sort to w zasadzie to samo: $O(k \cdot (n+d))$, bo algorytm zawsze robi dokładnie k przejść, niezależnie od ułożenia danych.

4 Insertion Sort na liście jednokierunkowej

Poniższy fragment kodu pokazuje sortowanie listy jednokierunkowej metodą Insertion Sort:

```
Wezel* insertion_sort_lista(Wezel* poczatek) {
    Wezel* posortowana_czesc = nullptr;
    while (poczatek != nullptr) {
        Wezel* biezacy = poczatek;
        poczatek = poczatek->nastepny;
        // wstawiamy biezacy w odpowiednie miejsce
    }
    return posortowana_czesc;
}
```

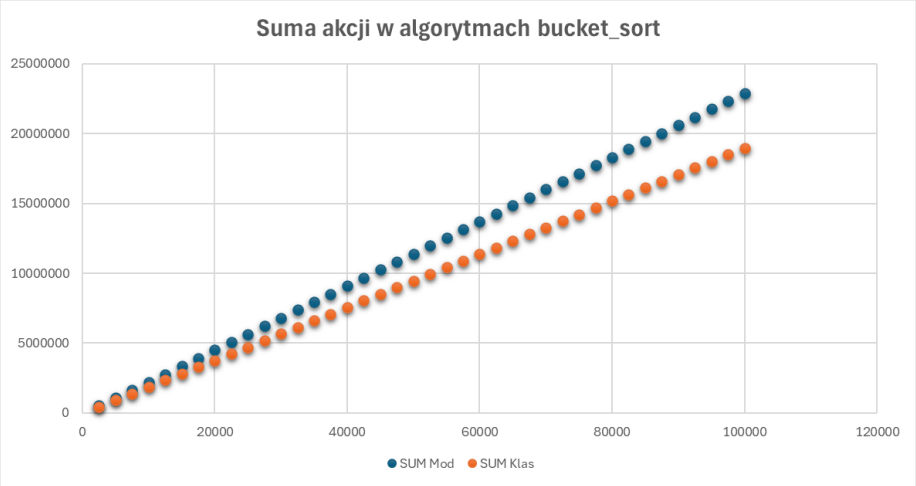
Ten fragment kodu realizuje sortowanie listy jednokierunkowej metodą Insertion Sort. Algorytm bierze po kolei każdy element z oryginalnej listy i wstawia go w odpowiednie miejsce w nowej, posortowanej części. W przeciwieństwie do sortowania tablic, tutaj nie przesuwamy całych elementów, tylko zmieniamy wskaźniki, co sprawia, że operacja jest bardziej elastyczna i wydajna dla list.

5 BUCKET SORT

Bucket sort dzieli przedział wartości na równe przedziały mozna wyobrazic sobie to jako kubełki. Każdy element trafia do odpowiedniego kubełka na podstawie swojej wartości. Następnie każdy kubełek jest sortowany wewnętrznie (np. insertion sortem). Na końcu zawartości kubełków są łączone w posortowaną całość. Normalnie Bucket Sort działa tylko dla liczb z przedziału od 0 do 1, co jest pewnym ograniczeniem. Moja modyfikacja polega na usunięciu tego ograniczenia, dzięki czemu algorytm może sortować liczby w dowolnym zakresie. Dzięki temu można go zastosować znacznie szerzej, nie tracąc przy tym idei dzielenia na koszyki i sortowania ich osobno. Udaje nam się to zrobić dzięki poniższemu fragmentowi:

```
double norm = (A[i]-min)/(max-min);
int indeks = int(norm * n);
```

Tworzymy w ten sposób „normę", która zamienia każdą liczbę na ułamek z przedziału [0,1). Innym ważnym elementem jest:

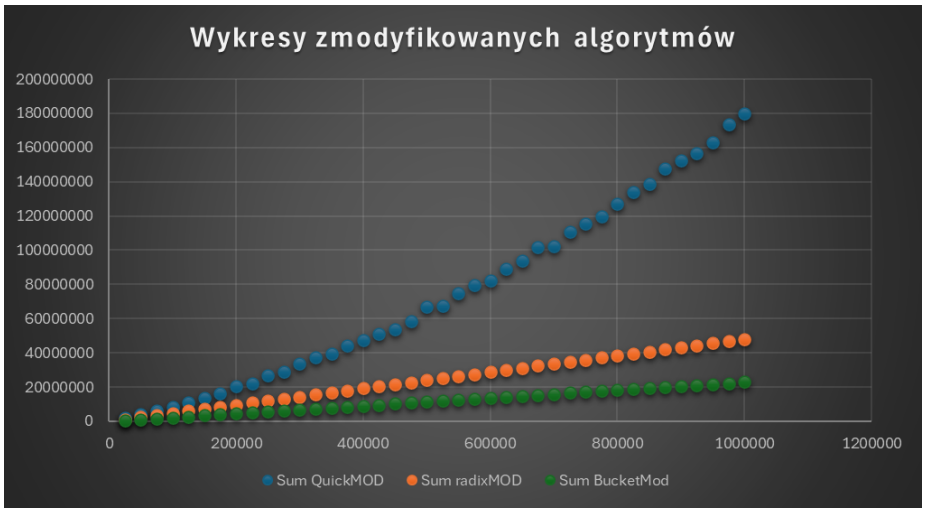



```
for (double x : lista)
{
    tablica[i] = x;
    i += 1;
    przypisania += 2;
}
insertion_sort(tablica, size);
```

Dzięki temu możemy zamienić listę na tablicę, z którą łatwiej i szybciej przeprowadzić sortowanie.

5.1 Porównanie Modyfikacji do klasycznego algorytmu

Na poniższym wykresie widać, że suma porównań i przypisań w wersji zmodyfikowanej (kolor niebieski) jest większa. Ma to sens, ponieważ modyfikacja wprowadziła dodatkowe warunki, nie skracając przy tym samego procesu sortowania. Średnia oraz pesymistyczna złożoność czasowa Bucket Sort wynosi w przybliżeniu $O(n + k)$, gdzie n to liczba elementów, a k to liczba koszyków, zakładając równomierny rozkład danych.



Rysunek 1: Suma porównan i przypisań dla zmodyfikowanych algorytmów

6 Podsumowanie

Jak widać na powyższym wykresie, zmodyfikowany Quick Sort (kolor niebieski) wypada dość słabo. Na drugim miejscu pod względem liczby operacji jest Radix Sort (kolor pomarańczowy) z $d = 10$, a najmniej działań wymaga Bucket Sort (kolor zielony).