

# sprawozdanie lista 2

Jakub Knaspi

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:**

Moja modyfikacja Quick Sort polega na użyciu dwóch pivotów, co powoduje powstanie trzech przedziałów: elementy mniejsze od mniejszego pivotu, elementy pomiędzy pivotami oraz elementy większe od większego pivotu. W mojej wersji pivoty dobierane są w taki sposób, aby po zakończeniu pojedynczego przebiegu były one sąsiednie, co ułatwia dalsze podziały. Poniżej przedstawiono trzy przypadki, które determinują sposób umieszczania elementów względem dwóch pivotów:

- **Przypadek 1: element mniejszy lub równy obu pivotom** W tej sytuacji element trafia do części przed mniejszym pivotem — zostaje umieszczony na pozycji  $i$ , natomiast dotychczasowy element z pozycji  $i$  przesuwany jest na pozycję  $j$ . Pod koniec iteracji zmienne  $i$  oraz  $v$  wyznaczają ostateczne miejsca, w których zostaną umieszczone odpowiednio mniejszy i większy pivot. Po wykonaniu operacji zwiększamy  $i$  oraz  $v$  o jeden.

- **Przypadek 2: element znajduje się pomiędzy pivotami**

W tym przypadku element jest najpierw zamieniany z wartością znajdującą się na pozycji  $v-1$ . Następnie trafia na pozycję  $k-1$ , która odpowiada miejscu dla nowego mniejszego pivotu (drugiego od końca w przetwarzanym zakresie). Po wykonaniu operacji zwiększamy zmienne  $v$  oraz  $i$ .

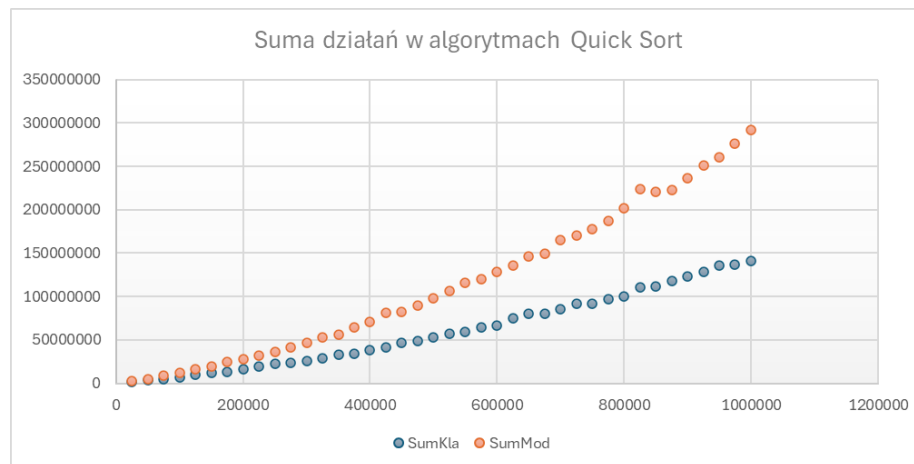
- **Przypadek 3: element większy od obu pivotów**

Jeżeli element jest większy zarówno od mniejszego, jak i większego pivotu, nie wykonujemy żadnej dodatkowej operacji — pozostaje on w części prawej, przeznaczonej dla elementów większych od pivotów.

```
for (int j = p; j < k-1; j++)
{
    if (A[j] <= A[k])
    {
        v+=1;
        if (A[j] <= A[k-1])
        {
            i+=1;
            zamien(A,i,j);
        }
        else
        {
            zamien(A,v-1,j);
            zamien(A,v-1,k-1);
            i+=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.

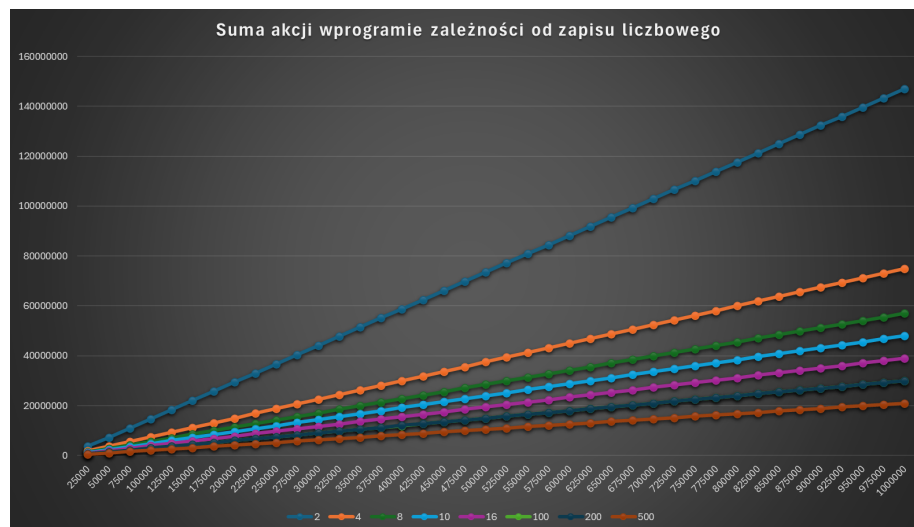
Wiele algorytmów potrafi działać wyłącznie na jednym, określonym typie liczbowym, natomiast Radix Sort może pracować na różnych rodzajach liczb, ponieważ wyodrębnia ich cyfry niezależnie od konkretnego typu danych.

```
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]--;
    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 można wyobrazić sobie to jako kubelki. Każdy element trafia do odpowiedniego kubelka na podstawie swojej wartości. Następnie każdy kubek jest sortowany wewnętrznie (np. insertion sortem). Na końcu zawartości kubeków 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:

```

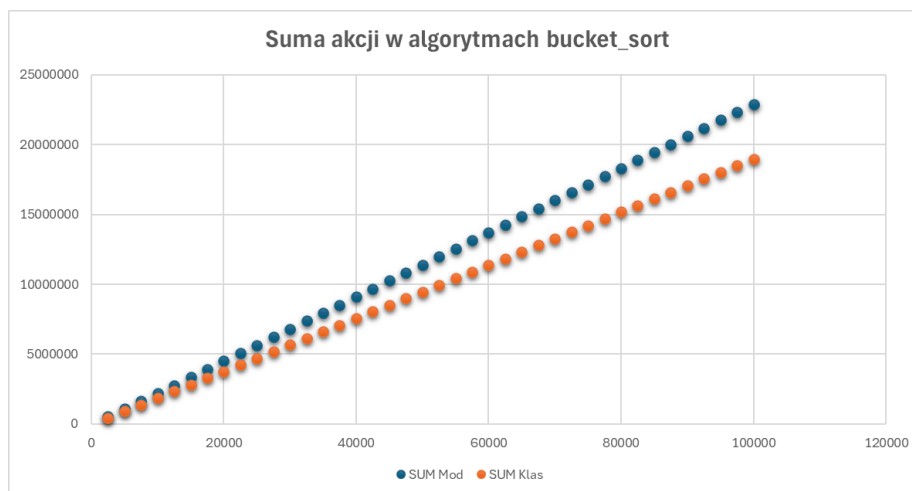
for (double x : lista)
{
    tablica[i] = x;
    i += 1;
    przypisania += 2;
}
insertion_sort(tablica, size);

```

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

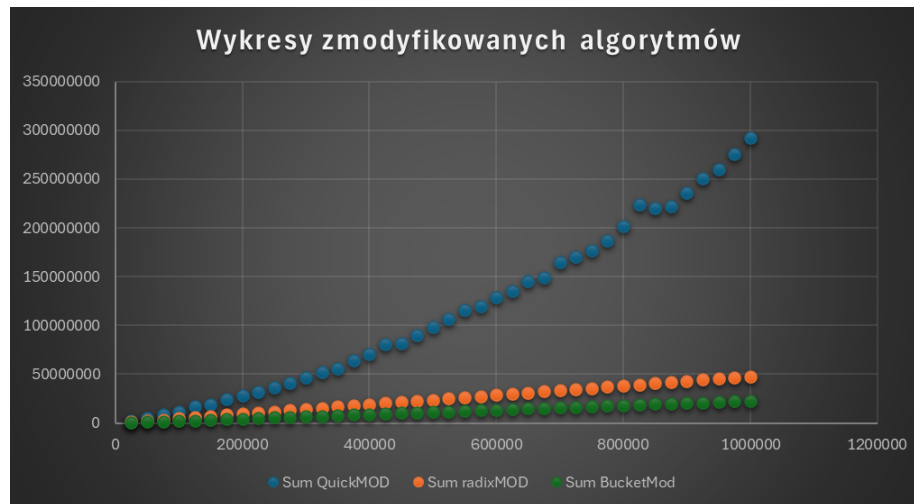
## 5.1 Porównanie Modyfikacji do klasycznego algorytmu

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 skracaając 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.

## 6 Podsumowanie



Rysunek 1: Suma porównań i przypisań dla zmodyfikowanych algorytmów

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).