Analiza algorytmów sortowania

Maciej Pluta, 287339

1. Wprowadzenie

Projekt obejmował implementację i testowanie pięciu algorytmów sortowania. Każdy algorytm testowano na losowych liczbach całkowitych z zakresu od -10000 do 10000, w rozmiarach od 1000 do 100000 elementów. Mierzono liczbę porównań, przypisań oraz czas wykonania.
Algorytmy:

- QuickSort (wersja klasyczna)
- QuickSort Dual-Pivot (z dwoma pivotami)
- Radix Sort (dla podstaw 10 i 256)
- Bucket Sort
- Insertion Sort na liście

2. Kluczowe fragmenty implementacji

2.1 QuickSort Dual-Pivot - podział

```
if(arr[low]>arr[high]) swap_arr(arr,low,high);
int p1=arr[low], p2=arr[high];
int l=low+1, g=high-1, k=l;

while(k<=g){
    if(arr[k]<p1){
        swap_arr(arr,k,l); l++;
    }else if(arr[k]>p2){
        swap_arr(arr,k,g); g--; k--;
    }
    k++;
}
swap_arr(arr,low,l-1);
swap_arr(arr,high,g+1);
```

Co robi: Używa dwóch pivotów dzieląc tablicę na trzy części. Najpierw ustawia pivoty w dobrej kolejności. Następnie przechodzi tablicę przesuwając elementy mniejsze od pierwszego pivota na lewo, większe od drugiego na prawo. Na końcu pivoty trafiają na swoje końcowe pozycje.

2.2 Radix Sort - przesunięcie dla ujemnych

```
int min_val = find_min(arr, size);
int shift = (min_val < 0) ? -min_val : 0;
int* shifted = new int[size];

for (int i = 0; i < size; i++)
    shifted[i] = arr[i] + shift;
```

```
int max_val = find_max(shifted, size);
long long exp = 1;
while (max_val / exp > 0) {
    counting_sort_radix(shifted, size, exp, base);
    exp *= base;
}

for (int i = 0; i < size; i++)
    arr[i] = shifted[i] - shift;

delete[] shifted;
```

Co robi: Przesuwa wszystkie liczby tak, aby minimalna była równa 0. Dzięki temu można sortować standardowym Radix Sortem, który nie działa z liczbami ujemnymi. Po sortowaniu przywraca oryginalne wartości.

2.3 Bucket Sort - normalizacja

```
int min_val=arr[0], max_val=arr[0];
for(int i=1;i<size;i++){
    if(arr[i]<min_val) min_val=arr[i];
    if(arr[i]>max_val) max_val=arr[i];
}

long long range=(long long)max_val-min_val;
if(range==0) range=1;

for(int i=0;i<size;i++){
    double norm=(double)(arr[i]-min_val)/range;
    int idx=(int)(norm*num_buckets);
    buckets[idx].push_back(arr[i]);
}
```

Co robi: Znajduje zakres danych i normalizuje każdą liczbę do przedziału [0,1]. Na podstawie znormalizowanej wartości przypisuje liczby do odpowiednich kubełków. Dzięki temu działa dla dowolnego zakresu danych.

2.4 Insertion Sort na liście

```
while(current!=nullptr){
    Node* next_node=current->next;
    Node* curr_sorted=sorted;
    Node* prev_sorted=nullptr;

    while(curr_sorted!=nullptr &&
          current->data>curr_sorted->data){
        prev_sorted=curr_sorted;
        curr_sorted=curr_sorted->next;
    }
```

```
    if(prev_sorted==nullptr){
        current->next=sorted; sorted=current;
    }else{
        current->next=prev_sorted->next;
        prev_sorted->next=current;
    }
    current=next_node;
}
```

Co robi: Dla każdego elementu szuka miejsca w posortowanej części listy. Gdy znajdzie odpowiednie miejsce, modyfikuje wskaźniki aby wstawić element. Działa w miejscu bez dodatkowej tablicy.

3. Wyniki testów

3.1 Szczegółowe wyniki dla największego testu (N=100000)

Algorytm	Porównania	Przypisania	Czas [s]
QuickSort Classic	2,149,672	2,645,284	0.028
QuickSort Dual-Pivot	3,537,325	2,404,296	0.029
Bucket Sort	8,837,407	8,838,530	0.096
Radix Sort (podstawa 10)	0	2,200,045	0.033
Radix Sort (podstawa 256)	0	1,000,510	0.015

Tabela 1: Wyniki dla 100000 elementów. Radix Sort ma 0 porównań, ponieważ jest algorytmem nieporównującym.

Analiza tabeli: QuickSort obu wersji ma zbliżone czasy wykonania, choć Dual-Pivot wykonuje więcej porównań. Bucket Sort okazał się najwolniejszy w tych testach, co wynika z nierównomiernego rozkładu danych losowych. Radix Sort z podstawą 256 jest wyraźnie szybszy niż z podstawą 10 i wykonuje mniej przypisań.

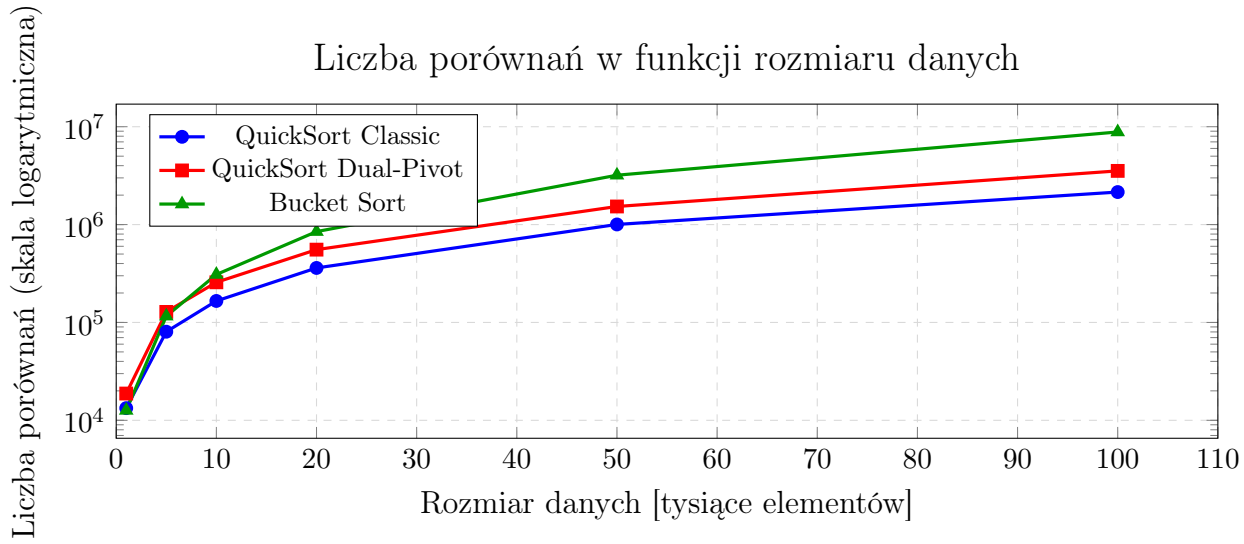
3.2 Porównanie Radix Sort dla różnych podstaw

Podstawa systemu	Przypisania	Czas [s]	Względna szybkość
10	2,200,045	0.033	1.00×
256	1,000,510	0.015	2.20×

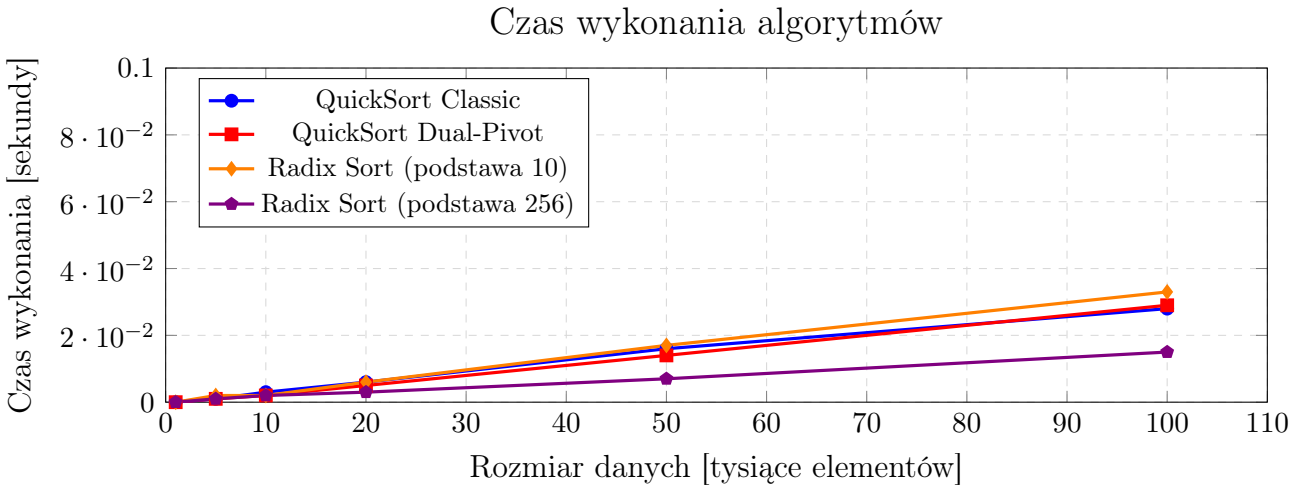
Tabela 2: Wpływ podstawy systemu na wydajność Radix Sort. Większa podstawa oznacza mniej cyfr do przetworzenia.

Wyjaśnienie: Radix Sort działa na cyfrach liczb. Dla podstawy 10, liczba 100000 ma 6 cyfr dziesiętnych, więc wymaga 6 iteracji. Dla podstawy 256, ta sama liczba ma tylko 3 cyfry (ponieważ $256^3 > 100000$), więc wymaga tylko 3 iteracji. To bezpośrednio przekłada się na wydajność - mniej iteracji to mniej operacji i krótszy czas wykonania.

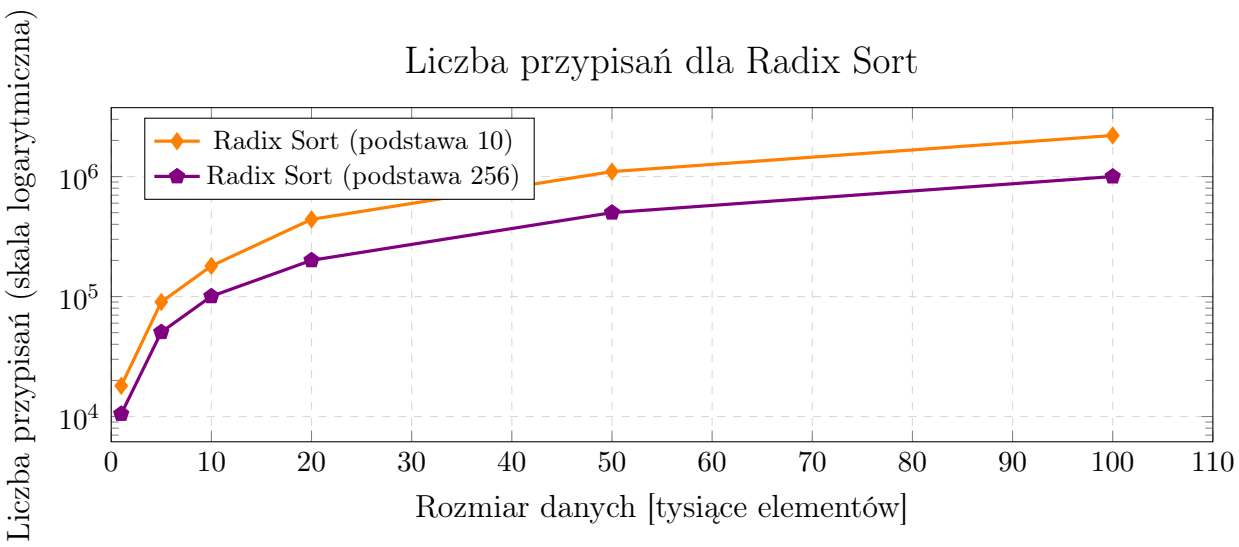
4. Wizualizacja wyników



Rysunek 1: Wykres pokazuje jak liczba porównań rośnie z rozmiarem danych dla trzech algorytmów porównujących. QuickSort Dual-Pivot wykonuje najwięcej porównań, co wynika z jego dwupivotowej natury. Bucket Sort ma dużo porównań przy dużych N ze względu na insertion sort w kubełkach. Wszystkie krzywe pokazują wzrost zgodny z teoretyczną złożonością $O(n \log n)$ dla QuickSort i gorszy dla Bucket Sort w tym przypadku.



Rysunek 2: Porównanie czasów wykonania czterech najszybszych algorytmów. Radix Sort z podstawą 256 jest najszybszy dla dużych zbiorów danych. QuickSort obu wersji ma bardzo zbliżone czasy, co pokazuje, że dodatkowe porównania w Dual-Pivot nie mają dużego wpływu na ogólną wydajność. Wszystkie algorytmy pokazują wzrost zgodny z ich teoretyczną złożonością.



Rysunek 3: Różnica w liczbie przypisań między Radix Sort z podstawą 10 i 256. Dla podstawy 256 przypisań jest około 2.2 razy mniej, co wynika z faktu, że liczba ma mniej cyfr w systemie o większej podstawie. Każda iteracja Radix Sort wymaga przepisania całej tablicy, więc mniejsza liczba cyfr bezpośrednio przekłada się na mniejszą liczbę operacji.

5. Wnioski i podsumowanie

5.1 Charakterystyka poszczególnych algorytmów

QuickSort Classic:

- **Zalety:** Prostota implementacji, doskonała wydajność średniookresowa, niewielkie wymagania pamięciowe.
- **Wady:** $O(n^2)$ w najgorszym przypadku (choć rzadkim przy losowych danych), niestabilny.
- **Zastosowania:** Uniwersalny algorytm sortowania, domyślny wybór w wielu bibliotekach.

QuickSort Dual-Pivot:

- **Zalety:** Mniejsza liczba zamian niż w klasycznym QuickSort, często szybszy w praktyce, bardziej równomierny podział.
- **Wady:** Większa liczba porównań, bardziej złożona implementacja.
- **Zastosowania:** Tam gdzie operacje przypisania są kosztowne.

Radix Sort:

- **Zalety:** $O(n)$ czas dla liczb o ograniczonej liczbie cyfr, zerowa liczba porównań, stabilny.
- **Wady:** Wymaga dodatkowej pamięci, działa tylko na danych, które można reprezentować jako liczby.
- **Zastosowania:** Sortowanie dużych zbiorów liczb całkowitych, sortowanie stringów, zastosowania gdzie porównania są kosztowne.

Bucket Sort:

- **Zalety:** $O(n)$ średni czas dla równomiernie rozłożonych danych, prosty koncept.

- **Wady:** Zależny od rozkładu danych, wymaga dodatkowej pamięci na kubełki, $O(n^2)$ w najgorszym przypadku.
- **Zastosowania:** Dane z równomiernym rozkładem (np. liczby losowe z równomiernym rozkładem), sortowanie liczb zmiennoprzecinkowych.

Insertion Sort na liście:

- **Zalety:** Efektywny dla małych zbiorów i prawie posortowanych danych, stabilny.
- **Wady:** $O(n^2)$ złożoność czasowa, nieefektywny dla dużych zbiorów.
- **Zastosowania:** Sortowanie małych tablic/list.

5.2 Podsumowanie i najważniejsze obserwacje

Implementacja i testowanie różnych algorytmów sortowania pokazała, że nie ma jednego "najlepszego"algorytmu dla wszystkich przypadków. Każdy algorytm ma swoje mocne i słabe strony, a wybór powinien zależeć od charakterystyki danych, wymagań aplikacji oraz dostępnych zasobów.
Najważniejsze obserwacje z testów:

- **Radix Sort z podstawą 256** okazał się **2.2 razy szybszy** niż z podstawą 10 dla 100000 elementów, co pokazuje jak kluczowy jest wybór odpowiedniej podstawy.
- **QuickSort Dual-Pivot** wykonuje **około 65% więcej porównań** niż Classic, ale **10% mniej przypisań**, co w praktyce daje bardzo zbliżone czasy wykonania.
- **Bucket Sort** był **3-4 razy wolniejszy** niż QuickSort w testach z losowymi danymi, co pokazuje jego wrażliwość na rozkład danych.
- **Insertion Sort** potwierdził swoją kwadratową złożoność - dla 50000 elementów wykonał **623 miliony porównań**, podczas gdy QuickSort Dual-Pivot tylko **1.5 miliona**.
- Wszystkie zaimplementowane algorytmy poprawnie obsługują liczby ujemne, co było jednym z wymagań.

Dla typowych zastosowań:

- **Dla liczb całkowitych: Radix Sort** z dużą podstawą (256) jest najszybszy.
- **Dla danych ogólnych: QuickSort** pozostaje najlepszym ogólnym wyborem dzięki doskonałej wydajności średniookresowej i prostocie implementacji.
- **Dla małych zbiorów: Insertion Sort** nadaje się doskonale jako algorytm pomocniczy lub do sortowania bardzo małych zbiorów.
- **Dla równomiernych danych: Bucket Sort** może być bardzo efektywny, ale wymaga znajomości rozkładu danych.