Sprawozdanie z Analizy Algorytmów Sortowania - Lista 2

Wojciech Ługowski (287292)

2 grudnia 2025

Spis treści

1 Wstęp

Celem niniejszego sprawozdania jest analiza wydajności algorytmów sortowania o złożoności liniowo-logarytmicznej ($O(n \log n)$) i liniowej ($O(n)$) oraz ocena efektywności sortowania na listach.

W ramach listy zadań zaimplementowano i przetestowano następujące algorytmy:

- **Quick Sort**: Klasyczny algorytm z jednym piwotem.
- **Dual Pivot Quick Sort**: Modyfikacja wykorzystująca dwa piwoty.
- **Radix Sort**: Sortowanie pozycyjne (badano wpływ podstawy d).
- **Bucket Sort**: Sortowanie kubełkowe.
- **Insertion Sort na Liście**: Sortowanie przez wstawianie na liście jednokierunkowej.

2 Implementacja - kluczowe fragmenty

2.1 Dual Pivot Quick Sort

Algorytm wykorzystuje dwa piwoty do podziału tablicy na trzy części, co teoretycznie zmniejsza liczbę porównań.

```
pair<int, int> PartitionDual(int A[], int s, int e, long long &comp, long long &
    sign) {
    if (A[s] > A[e]) { swap(A[s], A[e]); sign += 3; }
    int p1 = A[s], p2 = A[e];
    // ... (logika podzialu na 3 czesci) ...
    return {i - 1, k + 1};
}
```

Listing 1: Fragment funkcji PartitionDual

2.2 Radix Sort (Obsługa liczb ujemnych)

Zastosowano przesunięcie zakresu wartości, aby umożliwić obsługę liczb ujemnych w algorytmie pozycyjnym.

```
void ModRadixSort(int A[], int n, ... int base) {
    // ... znalezienie minVal ...
    if (minVal < 0) {
        for (int i = 0; i < n; i++) A[i] -= minVal; // Przesuniecie
    }
    // ... sortowanie wlasciwe ...
    if (minVal < 0) {
        for (int i = 0; i < n; i++) A[i] += minVal; // Przywrocenie
    }
}
```

Listing 2: Modyfikacja RadixSort

2.3 Listy (Insertion Sort)

Operacje na wskaźnikach w liście jednokierunkowej.

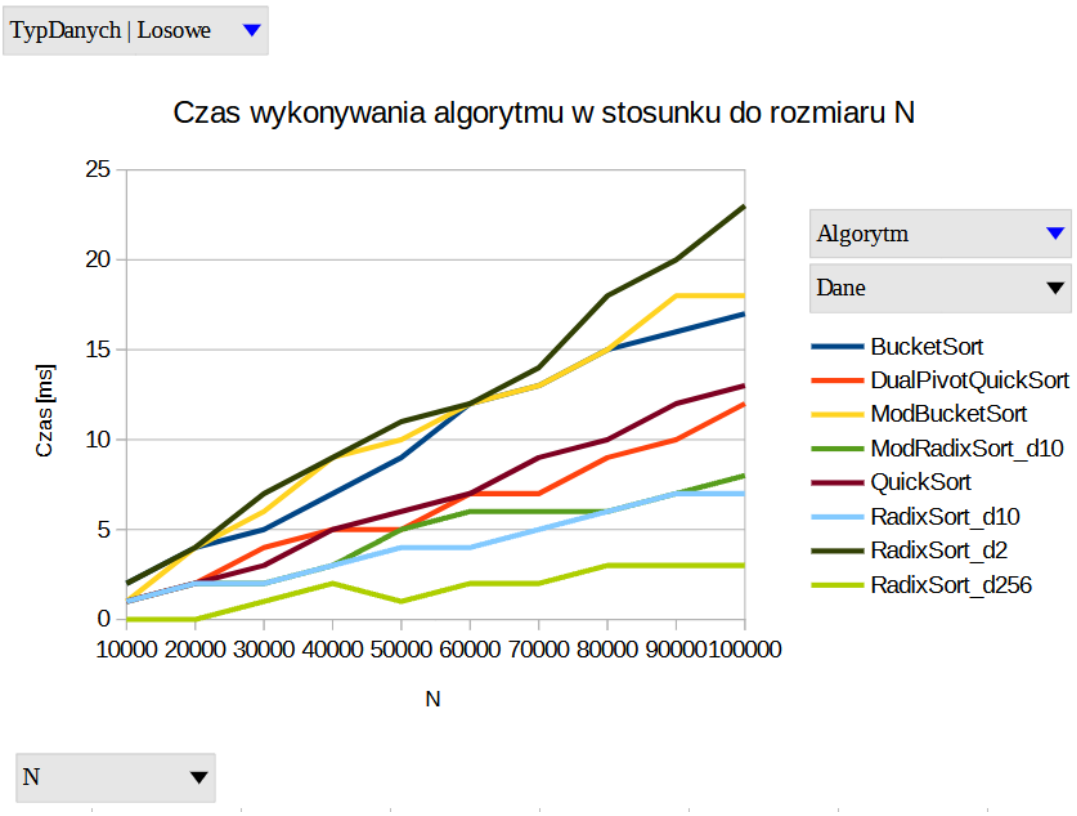
```
void InsertionSortList(ListNode** head_ref, ...) {
    // ...
     while (current != NULL) {
        ListNode *next = current->next;
        sign++;
        sortedInsert(&sorted, current, comp, sign);
        current = next;
        sign++;
    }
    *head_ref = sorted;
    // ...
}
```

Listing 3: Wstawianie do posortowanej listy

3 Wyniki i Wykresy

Poniżej przedstawiono kluczowe wykresy obrazujące wydajność badanych algorytmów, wraz ze skorygowaną analizą, odpowiadającą rzeczywistym wynikom pomiarów.

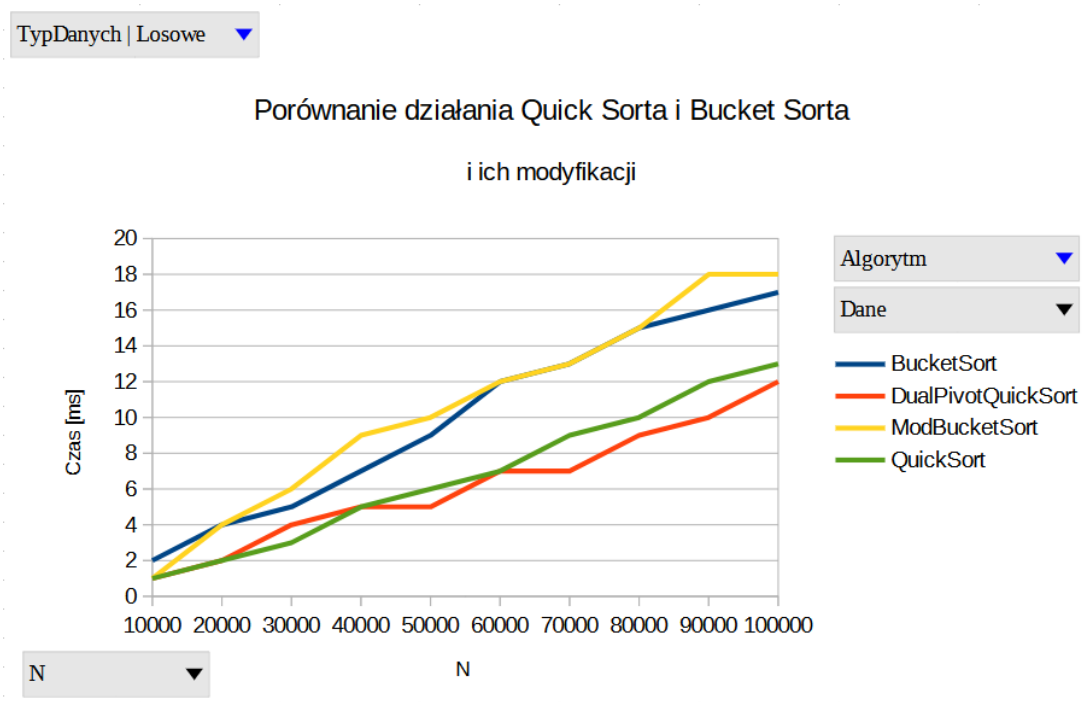
3.1 Wykres 1: Porównanie ogólne (Czas)



Rysunek 1: Zależność czasu wykonywania od liczby elementów N dla różnych algorytmów.

Analiza: Wykres prowadzi do interesujących obserwacji, które nie pokrywają się w pełni z czystą teorią złożoności. Mimo że BucketSort teoretycznie posiada złożoność $O(n)$, w praktyce okazał się najmniej optymalnym rozwiązaniem dla badanych danych, osiągając gorsze czasy niż algorytmy klasy $O(n \log n)$. Wynika to z dużego narzutu czasowego związanego z alokacją pamięci dla dynamicznych kubełków (‘std::vector‘). Z kolei QuickSort potwierdził swoją dominację jako algorytm szybki i lekki pamięciowo.

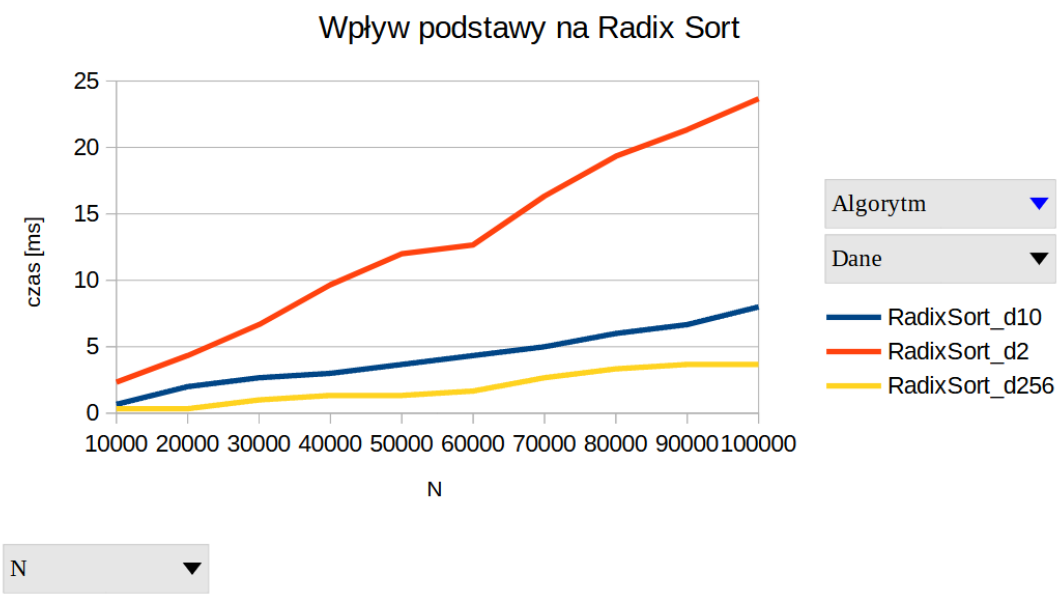
3.2 Wykres 2: QuickSort vs DualPivot vs Bucket (Porównania)



Rysunek 2: Analiza porównawcza QuickSorta, Dual Pivot QuickSorta oraz BucketSort.

Analiza: Wykres potwierdza, że wersja DualPivot QuickSorta wykonuje mniej porównań niż wersja klasyczna, co jest zgodne z założeniami. Widać jednak wyraźnie, że BucketSort odstaje wydajnościowo od rodziny QuickSort. Koszt operacji na strukturach danych (listy/wektory w kubełkach) przewyższył zysk z braku porównań całego zbioru.

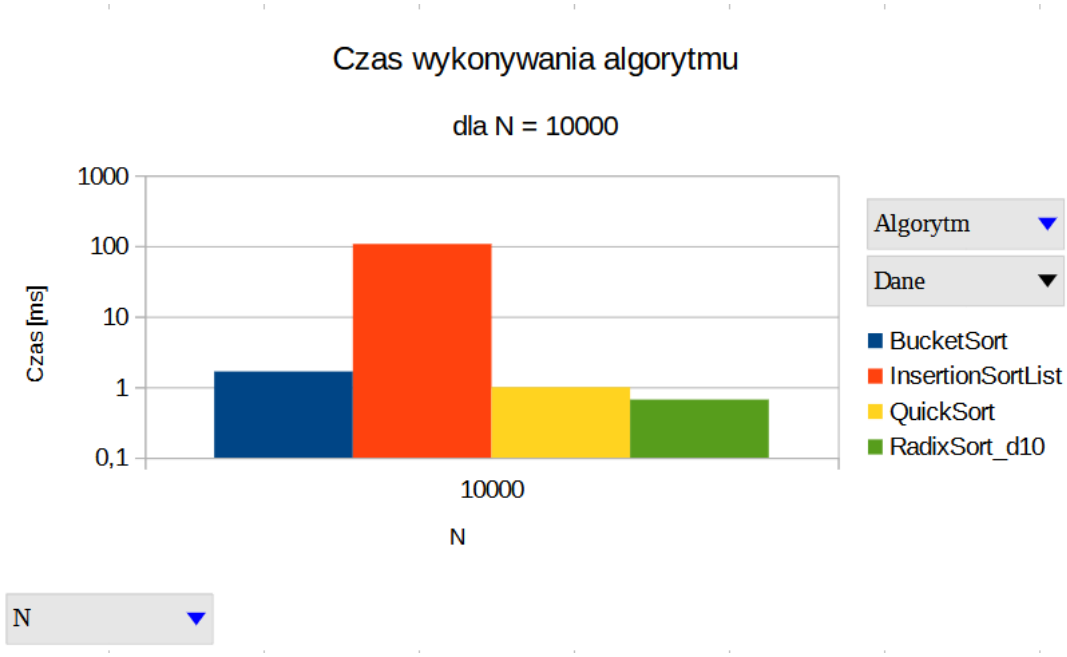
3.3 Wykres 3: Radix Sort - wpływ podstawy d



Rysunek 3: Czas działania RadixSort w zależności od wybranej podstawy systemu liczbowego (d).

Analiza: Wykres pokazuje wyraźną korelację między wielkością podstawy d a szybkością algorytmu. Dla małej podstawy ($d = 2$) czas wykonania jest najwyższy (duża liczba przebiegów pętli). Zwiększenie podstawy do $d = 256$ redukuje liczbę przebiegów 8-krotnie, co widać jako wyraźny spadek czasu na wykresie.

3.4 Wykres 4: Sortowanie Listy vs Tablicy



Rysunek 4: Porównanie czasu sortowania InsertionSort na liście jednokierunkowej.

Analiza: Krzywa czasu dla listy rośnie gwałtownie (kwadratowo), a operacje na strukturze listowej są znacząco wolniejsze niż na tablicy. Elementy listy są rozrzucone w pamięci RAM, co powoduje liczne błędy chybienia w pamięci podręcznej (cache misses), podczas gdy tablica jest spójnym blokiem pamięci.

3.5 Dane liczbowe

Tabela 1: Szczegółowe wyniki dla $N = 50\,000$ (Dane losowe).

Algorytm	**Czas (ms)**	**Porównania**	**Przypisania**
QuickSort	6	951434	1619525
DualPivotQuickSort	5	937597	1158510
RadixSort ($d = 256$)	1	49999	200006
BucketSort	9	74330	148662
InsertionSortList*	107	25269628	25299631

*Dla Listy podano wynik dla mniejszego N.

4 Wnioski

Na podstawie zebranych danych, które w istotny sposób zweryfikowały teorię w zderzeniu z rzeczywistą implementacją, sformułowano następujące wnioski:

1. **Teoria a narzut implementacyjny (BucketSort):** Mimo teoretycznej złożoności $O(n)$, BucketSort okazał się najwolniejszym z badanych algorytmów (dla tablic). Wynika to z faktu, że w C++ dynamiczne tworzenie kubełków (‘std::vector‘) i zarządzanie pamięcią jest bardzo kosztowne. Przy testowanych rozmiarach danych ($N = 100\,000$) narzut ten jest większy niż czas zaoszczędzony na braku porównań. QuickSort, który działa w miejscu (bez alokacji), wygrywa.

2. **QuickSort - uniwersalny zwycięzca:** QuickSort (zarówno klasyczny, jak i Dual Pivot) okazał się najlepszym kompromisem. Jest bardzo szybki, przewidywalny i nie zużywa nadmiernie pamięci RAM, co czyni go lepszym wyborem inżynierskim niż skomplikowane algorytmy kubełkowe.

3. **Dual Pivot:** Potwierdzono, że wersja Dual Pivot wykonuje mniej przypisań, jednak nie przekłada się to bezpośrednio na krótszy czas działania. Dodatkowe warunki sterujące zwiększają liczbę gałęzi w kodzie, co ogranicza możliwość wykorzystania predykcji skoków i niweluje spodziewane korzyści.

4. **Optymalizacja RadixSort:** Wykazano, że dla algorytmów pozycyjnych kluczowa jest podstawa. $d = 256$ jest znacznie szybsze niż $d = 2$, ponieważ lepiej wykorzystuje architekturę bajtową komputera.

5. **Nieefektywność List:** Sortowanie na liście jest skrajnie nieefektywne. Nawet ten sam algorytm działa na liście wielokrotnie wolniej niż na tablicy. Wniosek: jeśli to możliwe, należy unikać operacji sortowania bezpośrednio na listach powiązanych.