264511 Ivan Hancharyk

Piątek 9:15 TN

# Algorytmy i Złożoność Obliczeniowa

Sprawozdanie z projektu nr. 1

Temat: *„Badanie efektywności wybranych algorytmów sortowania ze względu na złożoność obliczeniową”*

# 1. Wprowadzenie

W ramach niniejszego sprawozdania badamy wydajność trzech podstawowych algorytmów sortowania: sortowania przez wstawianie (insertion sort), sortowania bąbelkowego (bubble sort) oraz sortowania przez scalanie (merge sort). Te algorytmy są fundamentalne w dziedzinie informatyki i często stanowią punkt wyjścia do nauczania technik sortowania ze względu na ich różnorodne zastosowania oraz charakterystyczne własności złożoności. Poniżej przedstawiono krótki opis każdego z algorytmów wraz z ich teoretycznymi oszacowaniami złożoności.

a) **Insertion Sort:** Cormen opisuje sortowanie przez wstawianie jako metodę, w której elementy wejściowe są konsumowane po jednym za każdym razem, każdy nowy element jest umieszczany w odpowiedniej pozycji wśród wcześniej posortowanych elementów.

**Złożoność czasowa**

**Najlepszy przypadek:**
**O(n):** osiągany, gdy tablica jest już posortowana, co pozwala algorytmowi na zakończenie sprawdzania każdego nowego elementu natychmiast po stwierdzeniu, że jest na właściwym miejscu.
**Średni przypadek:**
**O(n^2):** oczekiwany scenariusz dla losowo rozmieszczonych danych, gdzie prawdopodobieństwo znajdowania elementu blisko jego docelowego miejsca jest niskie.
**Najgorszy przypadek:**
**O(n^2):** występuje, gdy elementy są ułożone w odwrotnej kolejności, co wymaga przesuwania każdego elementu do końca już posortowanego fragmentu tablicy.

**Złożoność pamięciowa**
**O(1):** Insertion Sort jest sortowaniem na miejscu, co oznacza, że nie wymaga dodatkowej pamięci oprócz tej na wejściową tablicę.

b) **Bubble Sort:** Sortowanie bąbelkowe jest jednym z najprostszych algorytmów sortowania, którego główna idea polega na "bąblowaniu" największego niesortowanego elementu na jego odpowiednie miejsce na końcu listy za każdym przejściem przez tablicę. To osiągane jest przez wielokrotne przechodzenie przez listę, porównywanie i zamienianie miejscami sąsiadujących elementów, jeśli nie są w odpowiedniej kolejności.

**Złożoność czasowa**

**Najlepszy przypadek:**
**O(n):** Podobnie jak w Insertion Sort, najlepszy przypadek występuje, gdy tablica jest już posortowana i algorytm może zakończyć działanie po pierwszym przebiegu, jeśli nie wykonano żadnej zamiany.

**Średni przypadek:**
**O(n^2):** dla losowo ułożonych danych, gdzie każdy element prawdopodobnie nie jest blisko swojego docelowego miejsca.
**Najgorszy przypadek:**
**O(n^2):** występuje, gdy elementy są ułożone w odwrotnej kolejności, co wymaga maksymalnej liczby zamian dla każdego elementu.

**Złożoność pamięciowa**
**O(1):** Bubble Sort również jest sortowaniem na miejscu i nie potrzebuje dodatkowej pamięci oprócz przestrzeni na wejściową tablicę.

c) **Merge Sort:** Sortowanie przez scalanie jest przykładem efektywnego algorytmu sortowania wykorzystującego strategię "dziel i zwyciężaj". Ten algorytm dzieli oryginalną tablicę na coraz mniejsze połowy, aż do uzyskania list jednoelementowych, które są naturalnie posortowane. Następnie te małe listy są scalane w większe, posortowane listy, aż cała tablica będzie posortowana.

**Złożoność czasowa**

**Przypadek średni i najlepszy:**
**O(n log n):** Bez względu na początkowy rozkład danych, Merge Sort zawsze dzieli tablicę na dwie równe części i wykonuje rekurencyjne sortowanie scalające. Dzielenie to wymaga logarytmicznej liczby kroków (proporcjonalnie do liczby elementów n), a w każdym kroku, podczas procesu scalania, każdy element jest przetwarzany. Stąd złożoność wynosi **O(n log n).**
**Przypadek najgorszy:**
**O(n log n):** Ta sama złożoność obowiązuje również dla najgorszego przypadku, co czyni Merge Sort szczególnie atrakcyjnym dla danych, które muszą być posortowane w przewidywalnym czasie. Nawet jeśli dane są już posortowane lub są w najgorszym możliwym ułożeniu, czas działania Merge Sort nie ulega zmianie.

**Złożoność pamięciowa**
**O(n):** Merge Sort wymaga dodatkowej pamięci na przechowywanie tymczasowych tablic używanych podczas procesu scalania. To sprawia, że jest to algorytm "out-of-place", ponieważ nie sortuje danych in-place jak np. quicksort lub bubble sort. Z każdym rekurencyjnym wywołaniem funkcji sortującej, konieczne jest alokowanie nowej pamięci dla podtablic, co prowadzi do złożoności pamięciowej O(n).

# 2. Plan eksperymentu

Celem tego badania jest analiza i porównanie wydajności wymienionych algorytmów sortowania w zależności od rozmiaru, typu danych oraz ich początkowego rozkładu. Będziemy badać przypadki tablic całkowicie losowych, posortowanych rosnąco, posortowanych malejąco oraz posortowanych częściowo (33/66%), aby ocenić, jak początkowe ułożenie danych wpływa na efektywność sortowania. Wyniki tych

eksperymentów pozwolą nam zrozumieć, w jakich warunkach każdy z algorytmów może być najbardziej przydatny w praktycznych aplikacjach.

Przy pomiarze czasu w programie została wykorzystana biblioteka "chrono.h", która jest częścią standardowej biblioteki języka C++. Biblioteka ta operuje na czasie procesora, co pozwala na uzyskanie dokładnego czasu wykonywania kodu. Zostaje pobierany czas przed i po uruchomieniu potrzebnej nam funkcji, a zatem liczy się różnica czasu bazując na ilości okresów prebiegu procesora.

Pomiary zostały wykonane dla następnych rozmiarów tablic: {10 000, 20 000, 40 000, 80 000, 120 000, 160 000, 200 000}. Zakres liczb dla testów [-1000000, 1000000].

Po wykonaniu 100 prób sortowania dla każdego algorytmu wyniki zostały uśrednione i w sprawozdaniu zaprezentowane w formacie tabel (dane dla testów generowane są przez klasę generatorRandom.h).

```cpp
template<typename T>
std::vector<T> generatorRandom<T>::generate(int size, int mode) {
    T min = -1000000;  // Minimalna wartość
    T max = 1000000;   // Maksymalna wartość
    std::vector<T> data(size);
    std::random_device rd;
    std::mt19937 gen( sd: rd());

    if constexpr (std::is_integral<T>::value) {
        std::uniform_int_distribution<T> distrib(min, max);
        std::generate(data.begin(), data.end(), [&]() { return distrib(gen); });
    } else if constexpr (std::is_floating_point<T>::value) {
        std::uniform_real_distribution<T> distrib(min, max);
        std::generate(data.begin(), data.end(), [&]() { return distrib(gen); });
    }
```

```cpp
// Przetwarzanie danych zgodnie z wybranym trybem sortowania.
switch (mode) {
    case 1: // Losowe dane.
        break;
    case 2: // Sortowanie rosnąco.
        std::sort(data.begin(), data.end());
        break;
    case 3: // Sortowanie malejąco.
        std::sort(data.begin(), data.end(), std::greater<T>());
        break;
    case 4: // Częściowo sortowanie - 33% danych.
        std::sort(data.begin(), data.begin() + (size * 33 / 100));
        break;
    case 5: // Częściowo sortowanie - 66% danych.
        std::sort(data.begin(), data.begin() + (size * 66 / 100));
        break;
}
```

*Fragmenty kodu przedstawiające sposób losowania wartości*

Klasa „**generatorRandom**” służy do generowania wektorów z losowymi danymi dla różnych typów danych, takich jak liczby całkowite i zmiennoprzecinkowe, co umożliwia elastyczność w testowaniu algorytmów sortowania na różnorodnych zestawach danych. Użytkownik może określić rozmiar wektora, zakres wartości oraz sposób sortowania danych początkowych, co obejmuje generowanie danych całkowicie losowych, posortowanych rosnąco, malejąco, oraz częściowo posortowanych na poziomie 33% lub 66%. Dzięki zastosowaniu szablonów, „**generatorRandom**” jest w stanie obsłużyć dowolny typ danych, co czyni ją uniwersalnym narzędziem wspierającym zaawansowane scenariusze testowania algorytmów sortowania w różnych warunkach początkowych.

```cpp
// Metoda startująca timer; zapisuje obecny czas jako punkt startowy.
void Time::startTimer() {
    start = chrono::high_resolution_clock::now();
}

// Metoda zatrzymująca timer; zapisuje obecny czas jako punkt końcowy.
void Time::stopTimer() {
    end = chrono::high_resolution_clock::now();
}

// Metoda obliczająca i zwracająca czas trwania pomiędzy startem a stopem.
// Wynik jest konwertowany z mikrosekund dla większej dokładności.
double Time::getDuration() const {
    return chrono::duration_cast<chrono::microseconds>( d: end - start).count() / 1000.0;
}
```

*Klasa „Time.cpp”*

Wynik z klasy „Time” , który jest zwracany przez metodę „getDuration” w mikrosekundach, pozwala to ocenić efektywność implementacji i porównać wydajność poszczególnych algorytmów w bardzo precyzyjny sposób.

```cpp
// Sortuje dane metodą przez wstawianie.
void Sorter::insertionSort() {
    for (int i = 1; i < data.size(); i++) {
        double key = data[i];
        int j = i - 1;
        while (j >= 0 && data[j] > key) {
            data[j + 1] = data[j];
            j--;
        }
        data[j + 1] = key;
    }
}
```

*Implementacja algorytmu przez wstawianie*

W metodzie „insertionSort”, algorytm przegląda elementy wektora danych, począwszy od drugiego elementu. Dla każdego elementu, który jest rozważany, algorytm porównuje go z elementami wcześniejszymi i przesuwa te wcześniejsze elementy o jedną pozycję w prawo, aż znajdzie odpowiednie miejsce dla rozważanego elementu. Następnie wstawia element w znalezione miejsce. Proces ten jest powtarzany dla każdego elementu wektora, co prowadzi do ułożenia danych w porządku rosnącym.

```cpp
// Sortuje dane metodą bąbelkową.
void Sorter::bubbleSort() {
    bool swapped;
    for (int i = 0; i < data.size() - 1; i++) {
        swapped = false;
        for (int j = 0; j < data.size() - i - 1; j++) {
            if (data[j] > data[j + 1]) {
                swap( &: data[j], &: data[j + 1]);
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}
```

*Implementacja algorytmu sortowania bąbelkowego*

Metoda “bubbleSort” sortuje wektor danych poprzez wielokrotne przeglądanie listy i zamianę miejscami sąsiednich elementów, jeśli występują w nieprawidłowej kolejności. Pętla zewnętrzna iteruje przez listę od początku do końca, za każdym razem zmniejszając zakres sprawdzania o jeden, ponieważ największe elementy "wypływają" na koniec listy w kolejnych iteracjach. Pętla wewnętrzna porównuje kolejne elementy i zamienia je miejscami, jeśli są w niewłaściwej kolejności. Jeżeli podczas jednej z pełnych iteracji pętli nie wykonano żadnej zamiany, algorytm kończy działanie, co oznacza, że lista jest już posortowana.

```cpp
// Metody do realizacji sortowania przez scalanie.
void Sorter::merge(vector<double>& left, vector<double>& right, vector<double>& bars) {
    int nL = left.size(), nR = right.size();
    int i = 0, j = 0, k = 0;
    while (j < nL && k < nR) {
        if (left[j] < right[k]) bars[i++] = left[j++];
        else bars[i++] = right[k++];
    }
    while (j < nL) bars[i++] = left[j++];
    while (k < nR) bars[i++] = right[k++];
}

void Sorter::mergeSortRecursive(vector<double>& arr) {
    if (arr.size() > 1) {
        int mid = arr.size() / 2;
        vector<double> left( first: arr.begin(), last: arr.begin() + mid);
        vector<double> right( first: arr.begin() + mid, last: arr.end());

        mergeSortRecursive( &: left);
        mergeSortRecursive( &: right);
        merge( &: left, &: right, &: arr);
    }
}

void Sorter::mergeSort() {
    mergeSortRecursive( &: data);
}
```

*Implementacja algorytmu przez scalanie*

Metoda „mergeSort” inicjuje proces sortowania przez wywołanie rekurencyjnej funkcji „mergeSortRecursive”, która dzieli wektor na coraz mniejsze podzbiory aż do osiągnięcia jednoelementowych segmentów. Następnie, w procesie zwanym scalaniem, metoda „merge” łączy te podzbiory w sposób uporządkowany, efektywnie budując posortowany wektor z powrotem.

Podział wektora na dwie części odbywa się w „mergeSortRecursive”, gdzie mid jest środkiem wektora. Funkcja tworzy nowe wektory left i right, które reprezentują lewą i prawą połowę oryginalnego wektora. Po rekursywnym posortowaniu tych połówek, następuje scalenie.

Scalanie, realizowane przez metodę „merge”, porównuje elementy z obu połówek i umieszcza mniejszy z nich w wynikowym wektorze bars. Proces ten kontynuowany jest aż do wyczerpania elementów w obu wektorach połówek.

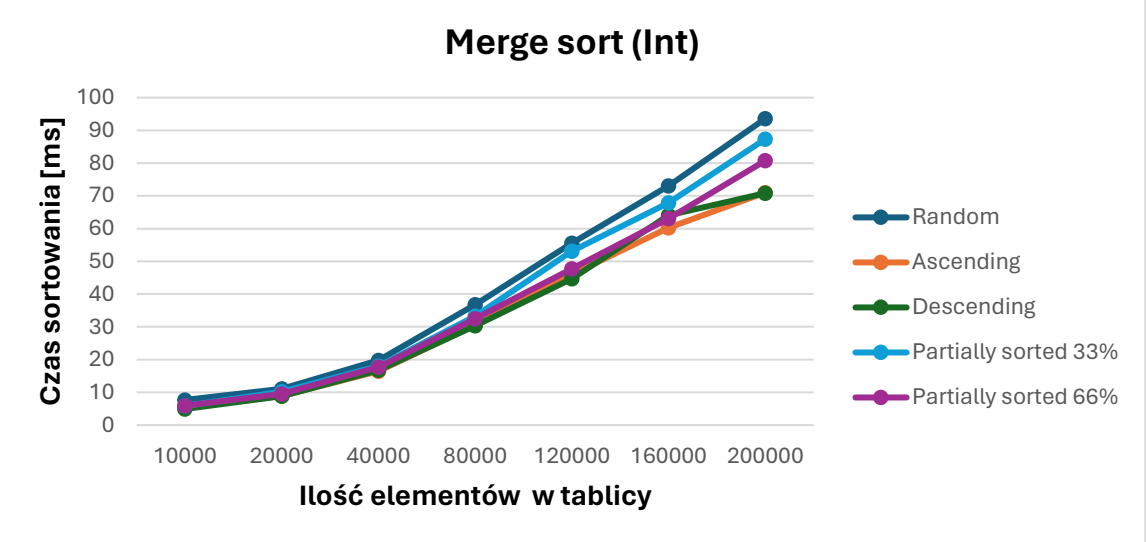
# 3. Wyniki (w postaci tabel i wykresów)

## Merge Sort dla typu danych int

*Tabela 1:*

| Rozmiar Tablicy | Random | Ascending | Descending | Partially sorted 33% | Partially sorted 66% |
|---|---|---|---|---|---|
| 10000 | 7,686 | 5,584 | 4,924 | 5,86 | 5,928 |
| 20000 | 11,112 | 8,903 | 8,792 | 10,17 | 9,549 |
| 40000 | 19,744 | 16,472 | 16,754 | 18,057 | 17,712 |
| 80000 | 36,764 | 30,552 | 30,366 | 33,314 | 32,502 |
| 120000 | 55,522 | 46,016 | 44,717 | 53,084 | 47,784 |
| 160000 | 73,037 | 60,265 | 64,031 | 67,916 | 63,108 |
| 200000 | 93,58 | 70,998 | 70,826 | 87,344 | 80,76 |

*Rysunek 1:*

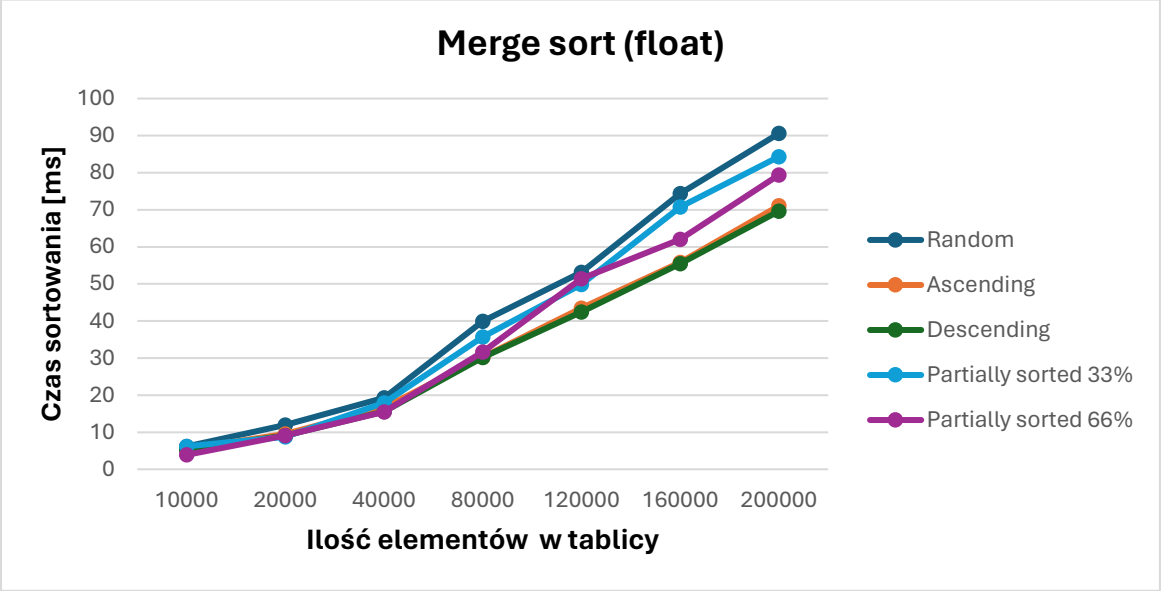


## Merge Sort dla typu danych float

*Tabela 2:*

| Rozmiar Tablicy | Random | Ascending | Descending | Partially sorted 33% | Partially sorted 66% |
|---|---|---|---|---|---|
| 10000 | 6,151 | 5,482 | 5,29 | 6,183 | 3,873 |
| 20000 | 11,986 | 9,525 | 9,103 | 8,736 | 9,057 |
| 40000 | 19,274 | 16,647 | 15,581 | 17,888 | 15,444 |
| 80000 | 39,904 | 30,241 | 30,182 | 35,713 | 31,672 |
| 120000 | 53,135 | 43,426 | 42,443 | 49,887 | 51,41 |
| 160000 | 74,356 | 55,821 | 55,464 | 70,71 | 62,039 |
| 200000 | 90,575 | 71,063 | 69,638 | 84,327 | 79,401 |

*Rysunek 2:*

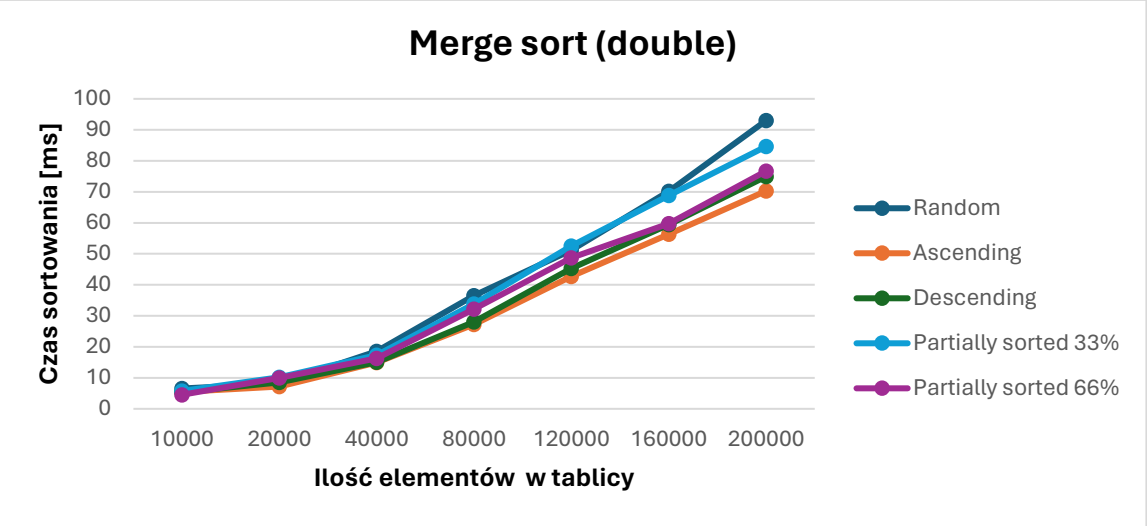



## Merge Sort dla typu danych double

*Tabela 3:*

| Rozmiar Tablicy | Random | Ascending | Descending | Partially sorted 33% | Partially sorted 66% |
|---|---|---|---|---|---|
| 10000 | 6,564 | 5,455 | 5,322 | 5,762 | 4,541 |
| 20000 | 8,089 | 7,195 | 8,478 | 10,192 | 10,016 |
| 40000 | 18,496 | 14,937 | 15,047 | 17,31 | 16,287 |
| 80000 | 36,466 | 27,212 | 27,999 | 33,834 | 32,133 |
| 120000 | 51,156 | 42,662 | 45,289 | 52,54 | 48,74 |
| 160000 | 70,173 | 56,356 | 59,491 | 68,882 | 59,75 |
| 200000 | 92,984 | 70,285 | 74,943 | 84,647 | 76,704 |

*Rysunek 3:*

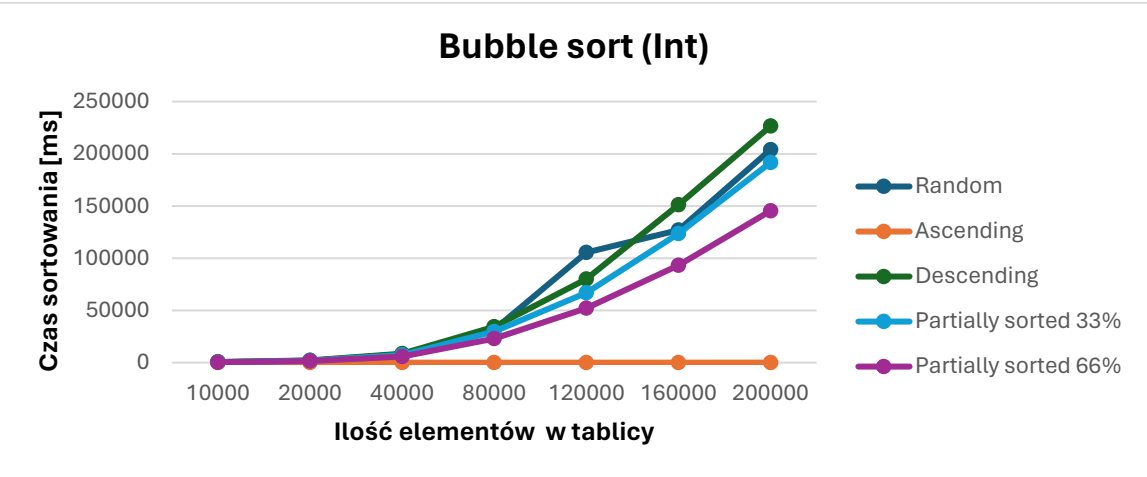



## Bubble Sort

*Tabela 4:*

| Rozmiar Tablicy | Random | Ascending | Descending | Partially sorted 33% | Partially sorted 66% |
|---|---|---|---|---|---|
| 10000 | 486,395 | 0,065 | 528,657 | 451,037 | 343,013 |
| 20000 | 1955,485 | 0,127 | 2112,233 | 1776,175 | 1441,312 |
| 40000 | 8038,938 | 0,254 | 8464,424 | 7338,749 | 5659,318 |
| 80000 | 31675,208 | 0,533 | 34274,353 | 29655,833 | 22930,36 |
| 120000 | 105449,091 | 0,79 | 80112,455 | 66946,27 | 52129,303 |
| 160000 | 126793,92 | 1,048 | 151286,611 | 123683,318 | 93105,723 |
| 200000 | 203987,522 | 1,336 | 226464,515 | 191721,064 | 145401,899 |

*Rysunek 4:*

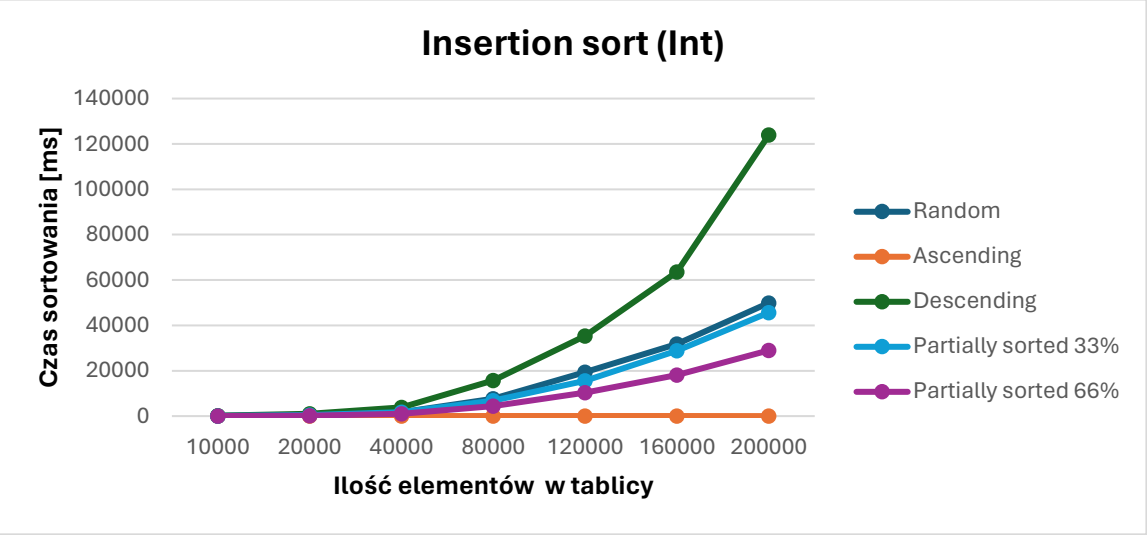


### Insertion Sort

*Tabela 5:*

| Rozmiar Tablicy | Random | Ascending | Descending | Partially sorted 33% | Partially sorted 66% |
|---|---|---|---|---|---|
| 10000 | 99,906 | 0,09 | 199,508 | 89,934 | 58,049 |
| 20000 | 498,358 | 0,225 | 1047,403 | 477,126 | 231,685 |
| 40000 | 1801,067 | 0,294 | 3880,171 | 1769,088 | 1077,541 |
| 80000 | 7618,394 | 0,43 | 15674,798 | 6849,476 | 4395,082 |
| 120000 | 19432,719 | 0,943 | 35270,792 | 15618,756 | 10315,601 |
| 160000 | 31842,342 | 1,609 | 63604,921 | 28796,67 | 18086,848 |
| 200000 | 49905,658 | 1,984 | 123913,302 | 45647,088 | 28980,548 |

*Rysunek 5:*




# 4. Podsumowanie i wnioski

## Efektywność algorytmów w różnych scenariuszach:

Sortowanie przez scalanie wykazuje konsekwentną wydajność niezależnie od porządku i typu danych, co jest zgodne z jego teoretyczną złożonością czasową **O(n log n)**. Jednak wydajność sortowania bąbelkowego znacznie spada wraz ze wzrostem rozmiaru tablicy, co podkreśla jego niepraktyczność dla większych zestawów danych z powodu złożoności **O(n^2)**.

## Znaczenie typu danych:

Z danych załączonych w sprawozdaniu nie ma podstaw, aby powiedzieć, że typ danych odgrywa znaczącą rolę w wydajności algorytmów sortowania. **Merge sort** został wybrany w celu oszczędzenia czasu, bo jest szybszy od dwóch innych badanych algorytmów. Chociaż dało nam to możliwość upewnić się w poprawnej implementacji algorytmu w projekcie, ponieważ typ danych nie za bardzo wpływa na jego wydajność.

Co do algorytmów **insertion** i **buble sort**, poza sprawozdaniem przeprowadziłem kilka pomiarów czasu dla tych algorytmów na innych typach danych niż **[Int]**. Pokazało to, że sortowanie liczb całkowitych jest zwykle szybsze w porównaniu z liczbami zmiennoprzecinkowymi i podwójnej precyzji, co jest zgodne z rozumieniem, że mniejsze typy danych wymagają mniejszego obciążenia obliczeniowego.

## Stabilność algorytmów:

Stabilność algorytmu, czyli jego zdolność do zachowania względnej kolejności równych elementów, nie jest wyraźnie zaznaczona w danych. Jednak stabilne algorytmy sortowania są kluczowe, gdy pierwotna sekwencja ma pewne istotne znaczenie, na przykład podczas sortowania rekordów według wtórnego klucza.

## Wydajność implementacji:

Stosowanie narzędzia <std::chrono::high_resolution_clock> z biblioteki C++ umożliwiło dokładne pomiarowanie czasów sortowania, co jest niezbędne dla wiarygodności przeprowadzonych eksperymentów.
Projekt dał nam wgląd w to, jak optymalizacja kodu może wpływać na wydajność algorytmów, podkreślając wagę pisania efektywnego kodu oraz wykorzystania zaawansowanych funkcji języków programowania do uzyskiwania maksymalnej wydajności.

## Potwierdzenie teoretycznych założeń:

Eksperymenty ogólnie potwierdzają teoretyczne oczekiwania dotyczące złożoności różnych algorytmów. Na przykład, sortowanie przez wstawianie pokazuje lepszą wydajność na danych częściowo posortowanych, co jest zgodne z jego najlepszym scenariuszem. Sortowanie przez scalanie konsekwentnie przewyższa sortowanie bąbelkowe dla większych rozmiarów danych, potwierdzając jego złożoność **O(n log n)** wobec złożoności **O(n^2)** sortowania bąbelkowego.

## Podsumowanie:

Eksperymenty przeprowadzone w ramach projektu dostarczyły cennych informacji o efektywności różnych algorytmów sortowania, wskazując na ich mocne i słabe strony w zależności od rozmiaru i typu danych, a także ich początkowego uporządkowania. Przyglądając się wynikom, możemy lepiej zrozumieć, w jakich warunkach wybrane algorytmy będą działać najszybciej, a kiedy ich użycie może nie być optymalne. Projekt pokazał także, jak ważne jest stosowanie właściwych narzędzi do pomiaru czasu i optymalizacji kodu, co przekłada się na dokładność i wiarygodność eksperymentalnych ocen wydajności algorytmów. Pozwoliło to na głębsze zrozumienie teoretycznych założeń oraz na ich praktyczną weryfikację. Wiedza ta jest nieoceniona przy projektowaniu nowych rozwiązań oraz optymalizacji istniejących algorytmów sortowania, co może przynieść znaczące korzyści w różnych obszarach informatyki.

# 5. Literatura


- Cormen T., Leiserson C.E., Rivest R.L., Stein C., Wprowadzenie do algorytmów, WNT
- https://www.geeksforgeeks.org/
- https://pl.wikipedia.org/wiki/Sortowanie_b%C4%85belkowe
- https://en.wikipedia.org/wiki/Merge_sort
- https://en.wikipedia.org/wiki/Insertion_sort


# 6. Kod źródłowy

**Main.cpp:**

```
#include <conio.h>
#include <iostream>
#include <string>
#include <chrono>
#include "Time.h"
#include "Sorter.h"
#include "generatorRandom.h"

using namespace std;
using namespace std::chrono;

// Funkcja wyświetlająca menu główne programu
void displayMenu(const string& info) {
    cout << endl;
    cout << info << endl;
    cout << "1. Load from file" << endl;
    cout << "2. Generate randomly" << endl;
    cout << "3. Display array" << endl;
    cout << "4. Sort by insertion" << endl;
    cout << "5. Sort by merging" << endl;
    cout << "6. Bubble sort" << endl;
    cout << "7. Test (measurements)" << endl;
    cout << "0. Return to menu" << endl;
    cout << "Choose an option: ";
}

Sorter sorter; // Instancja klasy Sorter do zarządzania danymi i sortowaniem

// Główna funkcja zarządzająca interakcją z użytkownikiem
void menu_sorting() {
    char opt;           // Opcja wybrana przez użytkownika
    string fileName;    // Nazwa pliku do wczytania
    int size;           // Rozmiar tablicy do generacji
    int dataType;       // Typ danych do generacji

    do {
        displayMenu("--- SORTING MENU ---");
        cin.ignore(numeric_limits<streamsize>::max(), '\n');  // Ignoruje
pozostałości w buforze
        opt = _getche();
        cout << endl;
```

```
        switch (opt) {
            case '1':
                cout << "Enter file name: ";
                cin >> fileName;
                sorter.loadFromFile(fileName);
                sorter.display();
                break;
            case '2':
                cout << "Enter array size: ";
                cin >> size;
                cout << "Insert data type (1 for int, 2 for float, 3 for
double): ";
                cin >> dataType;
                cout << "Choose data kind (1-Random, 2-Ascending, 3-
Descending, 4-Partially sorted 33%, 5-Partially sorted 66%): ";
                int mode;
                cin >> mode;
                int min, max;
                cout << "Enter minimum value: ";
                cin >> min;
                cout << "Enter maximum value: ";
                cin >> max;
                switch (dataType) {
                    case 1:
                        sorter.load(generatorRandom<int>::generate(size, min,
max, mode));
                        break;
                    case 2:
                        sorter.load(generatorRandom<float>::generate(size,
static_cast<float>(min), static_cast<float>(max), mode));
                        break;
                    case 3:
                        sorter.load(generatorRandom<double>::generate(size,
static_cast<double>(min), static_cast<double>(max), mode));
                        break;
                }
                sorter.display();
                break;
                sorter.display();
                break;
            case '3':
                sorter.display();
                break;
            case '4': // Sortowanie przez wstawianie
                cout << "Sorting array by insertion sort.\n";
                sorter.insertionSort();
                sorter.displaySorted();
                cout << (sorter.isSorted() ? "Array is sorted." : "Array is
NOT sorted.") << endl;
                break;
            case '5': // Sortowanie przez scalanie
                cout << "Sorting array by merge sort.\n";
                sorter.mergeSort();
                sorter.displaySorted();
                cout << (sorter.isSorted() ? "Array is sorted." : "Array is
NOT sorted.") << endl;
                break;
            case '6': // Sortowanie bąbelkowe
                cout << "Sorting array by bubble sort.\n";
                sorter.bubbleSort();
                sorter.displaySorted();
                cout << (sorter.isSorted() ? "Array is sorted." : "Array is
```

```
NOT sorted.") << endl;
                break;
            case '7': {
                cout << "Choose sort type (insertion, bubble, merge): ";
                string sortType;
                cin >> sortType;
                Time timer;
                timer.startTimer();

                if (sortType == "insertion") {
                    sorter.insertionSort();
                } else if (sortType == "bubble") {
                    sorter.bubbleSort();
                } else if (sortType == "merge") {
                    sorter.mergeSort();
                }

                timer.stopTimer();
                cout << "Time taken for " << sortType << " sort: " <<
timer.getDuration() << " ms" << endl;
                break;
            }

            case '0':
                break;
            default:
                cout << "Invalid option." << endl;
                break;
        }
    } while (opt != '0');
}

// Główna funkcja programu, która uruchamia menu
int main() {
    char option;
    do {
        cout << "==== MAIN MENU ====" << endl;
        cout << "1. Sorting" << endl;
        cout << "0. Exit" << endl;
        cout << "Choose an option: ";
        option = _getche();
        cout << endl;

        if (option == '1') {
            menu_sorting();
        }
    } while (option != '0');

    return 0;
}
```

## Sorter.cpp:

```
#include "Sorter.h"
#include <fstream>
#include <iostream>
#include <iomanip>
#include <vector>
#include "Time.h"

// Wyświetla aktualne dane w konsoli.
void Sorter::display() const {
```

```
    if (data.empty()) {
        cout << "Data vector is empty." << endl;
    } else {
        for (const auto& value : data) {
            cout << fixed << setprecision(3) << value << " ";
        }
        cout << endl;
    }
}

// Sortuje dane metodą przez wstawianie.
void Sorter::insertionSort() {
    for (int i = 1; i < data.size(); i++) {
        double key = data[i];
        int j = i - 1;
        while (j >= 0 && data[j] > key) {
            data[j + 1] = data[j];
            j--;
        }
        data[j + 1] = key;
    }
}

// Sortuje dane metodą bąbelkową.
void Sorter::bubbleSort() {
    bool swapped;
    for (int i = 0; i < data.size() - 1; i++) {
        swapped = false;
        for (int j = 0; j < data.size() - i - 1; j++) {
            if (data[j] > data[j + 1]) {
                swap(data[j], data[j + 1]);
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}

// Metody do realizacji sortowania przez scalanie.
void Sorter::merge(vector<double>& left, vector<double>& right,
vector<double>& bars) {
    int nL = left.size(), nR = right.size();
    int i = 0, j = 0, k = 0;
    while (j < nL && k < nR) {
        if (left[j] < right[k]) bars[i++] = left[j++];
        else bars[i++] = right[k++];
    }
    while (j < nL) bars[i++] = left[j++];
    while (k < nR) bars[i++] = right[k++];
}

void Sorter::mergeSortRecursive(vector<double>& arr) {
    if (arr.size() > 1) {
        int mid = arr.size() / 2;
        vector<double> left(arr.begin(), arr.begin() + mid);
        vector<double> right(arr.begin() + mid, arr.end());

        mergeSortRecursive(left);
        mergeSortRecursive(right);
        merge(left, right, arr);
    }
}
```

```cpp
void Sorter::mergeSort() {
    mergeSortRecursive(data);
}

// Metoda sprawdzająca, czy dane są posortowane.
bool Sorter::isSorted() const {
    for (int i = 0; i < data.size() - 1; i++) {
        if (data[i] > data[i + 1]) return false;
    }
    return true;
}

// Wczytywanie danych z pliku.
void Sorter::loadFromFile(const string& filename) {
    ifstream file(filename);
    if (!file) {
        cout << "File not found." << endl;
        return;
    }
    data.clear();
    double value;
    while (file >> value) {
        data.push_back(value);
    }
    file.close();
}

// Testowanie i mierzenie czasu sortowania.
void Sorter::testSort(const string& sortType) {
    Time timer;
    timer.startTimer(); // Rozpoczęcie pomiaru czasu

    if (sortType == "insertion") {
        insertionSort();
    } else if (sortType == "bubble") {
        bubbleSort();
    } else if (sortType == "merge") {
        mergeSort();
    }

    timer.stopTimer(); // Zakończenie pomiaru czasu
    cout << "Time taken for " << sortType << " sort: " << timer.getDuration()
<< " ms" << endl;
}
```

## Sorter.h:

```cpp
#ifndef SORTER_H
#define SORTER_H

#include <vector>
#include <string>
#include <iostream>
#include <algorithm>
using namespace std;

// Klasa Sorter służy do przechowywania i sortowania danych.
// Wspiera różne metody sortowania.
class Sorter {
private:
    vector<double> data;  // Przechowuje dane do sortowania.
```

```
public:
    // Wczytuje dane z wektora.
    template<typename T>
    void load(const vector<T>& input_data) {
        data.clear();
        data.reserve(input_data.size());
        for (const T& value : input_data) {
            data.push_back(static_cast<double>(value));
        }
    }

    // Wyświetla posortowane dane.
    void displaySorted() const {
        display();
    }

    void insertionSort();
    void bubbleSort();
    void mergeSort();
    bool isSorted() const; // Sprawdza, czy dane są posortowane.

    // Metody do realizacji sortowania przez scalanie.
    void merge(vector<double>& left, vector<double>& right, vector<double>&
bars);
    void mergeSortRecursive(vector<double>& arr);

    void loadFromFile(const string &filename);
    void display() const;
    void testSort(const string &sortType);
};

#endif // SORTER_H
```

## generatorRandom.h:

```
#ifndef AIZO_GENERATORRANDOM_H
#define AIZO_GENERATORRANDOM_H

#include <vector>
#include <random>
#include <algorithm>
#include <type_traits>

// Klasa generatorRandom służy do generowania wektorów z losowymi danymi.
// Obsługuje różne typy danych, takie jak liczby całkowite i
zmiennoprzecinkowe.
template<typename T>
class generatorRandom {
public:
    // Funkcja generate tworzy wektor z losowo wygenerowanymi danymi.
    // Parametr size określa rozmiar wektora.
    // Parametr mode określa sposób sortowania danych: losowy, rosnący,
malejący, częściowo posortowany (33% lub 66%).
    static std::vector<T> generate(int size, T min, T max, int mode = 0);
};

template<typename T>
std::vector<T> generatorRandom<T>::generate(int size, T min, T max, int mode)
{
    std::vector<T> data(size);
    std::random_device rd;
    std::mt19937 gen(rd());
```

```
    // Inicjalizacja danych w zależności od typu danych.
    if constexpr (std::is_integral<T>::value) {
        std::uniform_int_distribution<T> distrib(min, max);
        std::generate(data.begin(), data.end(), [&]() { return distrib(gen);
});
    } else if constexpr (std::is_floating_point<T>::value) {
        std::uniform_real_distribution<T> distrib(min, max);
        std::generate(data.begin(), data.end(), [&]() { return distrib(gen);
});
    }

    // Przetwarzanie danych zgodnie z wybranym trybem sortowania.
    switch (mode) {
        case 1: // Losowe dane.
            break;
        case 2: // Sortowanie rosnąco.
            std::sort(data.begin(), data.end());
            break;
        case 3: // Sortowanie malejąco.
            std::sort(data.begin(), data.end(), std::greater<T>());
            break;
        case 4: // Częściowo sortowanie - 33% danych.
            std::sort(data.begin(), data.begin() + (size * 33 / 100));
            break;
        case 5: // Częściowo sortowanie - 66% danych.
            std::sort(data.begin(), data.begin() + (size * 66 / 100));
            break;
    }

    return data;
}

#endif //AIZO_GENERATORRANDOM_H
```

## Time.cpp:

```
#include "Time.h"

// Metoda startująca timer; zapisuje obecny czas jako punkt startowy.
void Time::startTimer() {
    start = chrono::high_resolution_clock::now();
}

// Metoda zatrzymująca timer; zapisuje obecny czas jako punkt końcowy.
void Time::stopTimer() {
    end = chrono::high_resolution_clock::now();
}

// Metoda obliczająca i zwracająca czas trwania pomiędzy startem a stopem.
// Wynik jest konwertowany z mikrosekund dla większej dokładności.
double Time::getDuration() const {
    return chrono::duration_cast<chrono::microseconds>(end - start).count() /
1000.0;
}
```

## Time.h:

```
#ifndef TIME_H
#define TIME_H
```

```
#include <chrono>
using namespace std;

// Klasa Time wykorzystuje mechanizmy biblioteki chrono do mierzenia czasu.
// Pozwala na precyzyjne pomiary czasu wykonania określonych fragmentów kodu.
class Time {
private:
    // Zmienne do przechowywania punktów czasowych - początku i końca
pomiaru.
    chrono::time_point<chrono::high_resolution_clock> start, end;

public:
    void startTimer();
    void stopTimer();
    double getDuration() const;
};

#endif // TIME_H
```