Projekt: Algorytmy i struktury danych

Autor: Jakub Chodziński
Prowadzący: prof. dr hab. inż. Władysław Homenda

20 maja 2022 roku

WYŻSZA SZKOŁA
INFORMATYKI I ZARZĄDZANIA
z siedzibą w Rzeszowie

Jakub Chodziński (65530)
IID-P 2021/2022, SW01
Algorytmy i struktury danych
20 maja 2022 roku

Analiza empiryczna złożoności algorytmów sortowania i wyszukiwania

---

Jakub Chodziński & 20 maja 2022 roku

Rozdział 1

Opis projektu

1.1 Założenia projektów zaliczeniowych

- analiza empiryczna złożoności algorytmów sortowania i wyszukiwania,
- zastosowanie struktur danych w algorytmach grafowych: implementacja algorytmu Dijkstry i algorytmów przeszukiwania grafów wszerz i w głąb.

1.2 Analiza empiryczna złożoności algorytmów sortowania i wyszukiwania:

Algorytmy sortowania (wybrane dwa z pierwszych trzech i jeden z pozostałych dwóch algorytmów):

- bąbelkowego (bubble),
- przez wstawianie (insertion),
- przez wybór (selection),
- szybkiego (quick),
- przez scalanie (merge).

Algorytmy wyszukiwania:

- liniowego (linear),
- binarnego (binary).

Trzy typy danych wejściowych:

- posortowane,
- posortowane w kolejności odwrotnej,
- kolejność losowa.

Rozmiary danych wejściowych: n, 2n, 3n, 4n, 5n, 6n, ... gdzie n - rozmiar.

1.3 Sprawozdanie

- dokładna informacja o autorstwie programu (można korzystać z materiałów z Internetu i z pomocy kolegów, ale konieczna będzie dokładna znajomość programu), instrukcja kompilacji i uruchomienia, opis realizacji części źródeł programu odpowiedzialnych analizowane algorytmy i za empiryczną analizę złożoności algorytmów,
- tabelki/wykresy zależności liczb wykonań operacji od rozmiaru zadania,
- programy w C/C++ w tym analizowane algorytmy w czystym C, kompilowane w Visual Studio 2019, oddajemy źródła wszystkich części programu (wyjątek stanowią funkcje pomocnicze, np. użyte do rysowania wykresów),
- oddajemy projekt w 4 folderach: źródła programu z plikiem projektu do kompilacji / budowy programu, program wykonywalny (wersja Release) i ewentualnie potrzebne biblioteki, dane (ten folder pewnie będzie pusty),
- dokumentacja (sprawozdanie wspomniane wyżej, preferowany format pdf),
- program uruchamialny na komputerze z ćzystymśystemem Windows, tzn. bez konieczności instalacji kompilatora, środowiska programistycznego itp.,
- termin przesłania pliku zip z czterema folderami: 20 maja; zachęcam do wcześniejszego oddawania programów.

To są ogólne założenia zaliczenia na ocenę bdb, szczegółowe ustalenia pozostawiam Państwu, wszystkie rozsądne ustalenia zaakceptuję, zaakceptuję również rozsądną modyfikację powyższych założeń. W przypadku wątpliwości zapraszam na konsultacje.

Sprawozdanie musi zawierać następujące elementy

- sformułowanie problemu,
- opis problemu,
- opis rozwiązania,
- prezentacja wyników,
- wnioski.

Sprawozdanie powinno być adresowane do czytelników z ogólną wiedzą informatyczną, opisy muszą być jednoznaczne. Źródła programu należy komentować jednoznacznie, ale oszczędnie.

WYŻSZA SZKOŁA
INFORMATYKI I ZARZĄDZANIA
z siedzibą w Rzeszowie

Jakub Chodziński (65530)
IID-P 2021/2022, SW01
Algorytmy i struktury danych
20 maja 2022 roku

Spis treści

Rozdział 2

Projekt

2.1 Sformułowanie problemu

Analiza empiryczna dla następujących algorytmów sortowania:

- sortowanie bąbelkowe (bubble),
- sortowanie przez wstawianie (insertion),
- sortowanie przez scalanie (merge).

oraz algorytmów wyszukiwania:

- wyszukiwanie liniowe (linear),
- wyszukiwanie binarne (binary).

Przedstawienie otrzymanych danych dla wartości losowych, posortowanych rosnąco oraz malejąco za pomocą wykresów oraz sformułowanie wniosków.

2.2 Opis rozwiązania

Rozwiązywanie problemu zacząłem od utworzenia programu składającego się z:

- rozbudowanego interfejsu użytkownika, pozwalającego obsługiwać program bez znajomości języka programowania lub kodu,
- osobnych klas:
  - generujących i przechowujących odpowiednie tablice (z wartościami losowymi, wartościami posortowanymi rosnąco oraz wartościami posortowanymi malejąco),
  - sortujących za pomocą wybranego algorytmu podaną tablicę i zliczających dla niego operacje dominujące,
  - przeszukujących za pomocą wybranego algorytmu podaną tablicę i zliczających dla niego operacje dominujące,
- biblioteki umożliwiającej utworzenie pliku z zebranymi danymi w formacie umożliwiającym zaimportowanie go bezpośrednio do programu Microsoft Office Excel.

Następnie utworzyłem nowy skoroszyt programu Microsoft Office Excel, zaimportowałem otrzymane z programu dane oraz utworzyłem wykresy dla każdego z algorytmów z osobna przedstawiające różnice w ilości operacji dominujących dla tablic o różnej wielkości oraz zawierających dane losowe, dane posortowane rosnąco lub malejąco. Dla ukazania złożoności obliczeniowej algorytmów dodałem serie pomocnicze składające się z danych najgorszego przypadku.

2.3 Dokładny opis programu

Spis użytych bibliotek wraz z wytłumaczeniem ich zastosowania w programie:

- biblioteka **iomanip** została wykorzystana do ustawienia szerokości kolumny podczas wyświetlania wartości w programie,
- biblioteka **cstdlib** została wykorzystana do operacji czyszczenia ekranu, pauzy oraz do wygenerowania wartości pseudolosowych,
- biblioteka **string** została wykorzystana, aby umożliwić zwracanie wartości typu string przez metody i funkcje,
- biblioteka **fstream** została wykorzystana w celu umożliwienia programowi zapisu danych do pliku,
- biblioteka **cmath** została wykorzystana do obliczenia wartości potęgi oraz logarytmu o podstawie 2,
- biblioteka **ctime** została wykorzystana do zwrócenia aktualnej wartości daty i czasu,
- biblioteka **Windows.h** została wykorzystana w celu użycia funkcji Sleep(), która w tym przypadku umożliwia ponowne generowanie nowego ziarna dla wartości pseudolosowych.

Spis utworzonych klas wraz z opisem ich działania:

- klasa **Arrays** służy do wygenerowania 3 różnych tablic o podanej przez użytkownika długości, a także z określonego przedziału. Przykład obiektu opartego na tej klasie dla tablic o długości 12 elementów, których wartości dla tablicy losowej mają być losowane pseudolosowo z przedziału od 34 do 56 wygląda następująco:

  ```
  Arrays obiekt(12, 56, 34);
  ```

  Domyślnie wystarczy jako parametr podać tylko i wyłącznie długość tablic, wtedy wartością maksymalną będzie 100 a minimalną 1.

  Podana klasa posiada także swoje własne metody takie jak:

  - **fillArrays(size, max, min)** - służąca do ponownego wypełnienia tablic nowymi wartościami generowanymi na podstawie podanych przez użytkownika parametrów:
    - **size** określa długość generowanych przez nas tablic.
    - **max** określa maksymalną wartość dla przedziału z którego program generuje wartości pseudolosowe dla tablicy wartości losowych. Nie musimy podawać tego parametru, przyjmie on wtedy wartość domyślną równą 100.
    - **min** określa minimalną wartość dla przedziału z którego program generuje wartości pseudolosowe dla tablicy wartości losowych. Nie musimy podawać tego parametru, przyjmie on wtedy wartość domyślną równą 1.

  - **getArray(type)** - zwracająca użytkownikowi wybraną tablicę. Jeżeli dla parametru type wpiszemy wartość:
    * **0** - program zwróci tablicę wartości losowych
    * **1** - program zwróci tablicę wartości posortowanych rosnąco (od 1 do size), gdzie size to długość tablicy
    * **2** - program zwróci tablicę wartości posortowanych malejąco (od size do 1), gdzie size to długość tablicy
  - **printArrays()** - wyświetlająca na ekranie wartości wszystkich trzech przechowywanych przez obiekt tablic.

- klasa **SortingAlgorithms** służąca do sortowania przekazanej do obiektu tablicy oraz zliczenia ilości operacji dominujących. Przykład obiektu opartego na tej klasie wygląda następująco:

```
SortingAlgorithms obiekt(tablica, 34, true, 0);
```

  - Parametr **tablica** określa przekazaną do obiektu tablicę.
  - Parametr o wartości **34** określa długość przekazanej do obiektu tablicy.
  - Parametr o wartości **true** określa czy obiekt ma być zaraz po utworzeniu posortowany. W przypadku podania wartości **false** obiekt nie zostanie posortowany.
  - Parametr o wartości **0** wskazuje obiektowi jakim algorytmem ma sortować podaną przez użytkownika tablicę. Powyższy parametr może przyjąć następujące wartości:
    * **0** - algorytm sortowania bąbelkowego
    * **1** - algorytm sortowania przez wstawianie
    * **2** - algorytm sortowania przez scalanie

  Domyślnie wystarczy jako parametr podać tylko tablicę oraz jej długość, wtedy obiekt zostanie posortowany automatycznie algorytmem sortowania przez scalanie.

  Podana klasa posiada także swoje własne metody takie jak:

  - **printArray()** - wyświetlająca na ekranie wartości przechowywanej przez obiekt tablicy.
  - **printDominantOperations()** - wyświetlająca na ekranie liczbę operacji dominujących.
  - **printAlgorithm()** - wyświetlająca na ekranie słowną nazwę wybranego algorytmu sortującego.
  - **getArray()** - zwracająca przekazaną do obiektu tablicę.
  - **getDominantOperations()** - zwracająca liczbę operacji dominujących.
  - **getAlgorithm()** - zwracającą słowną nazwę wybranego algorytmu sortującego.
  - **setArray(array, size)** - przypisującą obiektowi nową tablicę - array o podanej długości - size.

  - **sort(algorithm)** - sortującą przechowywaną przez obiekt tablicę za pomocą wskazanego w algorytmu sortującego - algorithm. Powyższy parametr może przyjąć następujące wartości:
    * **0** - algorytm sortowania bąbelkowego
    * **1** - algorytm sortowania przez wstawianie
    * **2** - algorytm sortowania przez scalanie

    Domyślnie nie musimy podawać żadnego parametru, wtedy obiekt zostanie posortowany automatycznie algorytmem sortowania przez scalanie.

- klasa **SearchingAlgorithms** służąca do przeszukania przekazanej do obiektu tablicy oraz zliczenia ilości operacji dominujących. Przykład obiektu opartego na tej klasie wygląda następująco:

```
SearchingAlgorithms obiekt(tablica, 34, 12, true, 0);
```

  - Parametr **tablica** określa przekazaną do obiektu tablicę.
  - Parametr o wartości **34** określa długość przekazanej do obiektu tablicy.
  - Parametr o wartości **12** określa szukaną wartość w tablicy przez algorytm wyszukujący.
  - Parametr o wartości **true** określa czy obiekt ma zostać przeszukany pod względem szukanej wartości zaraz po jego utworzeniu. W przypadku podania wartości **false** obiekt nie zostanie przeszukany.
  - Parametr o wartości **0** wskazuje obiektowi jakim algorytmem ma przeszukiwać podaną przez użytkownika tablicę. Powyższy parametr może przyjąć następujące wartości:
    * **0** - algorytm wyszukiwania liniowego
    * **1** - algorytm wyszukiwania binarnego

  Domyślnie wystarczy jako parametr podać tylko tablicę, jej długość oraz szukaną wartość, wtedy obiekt zostanie przeszukany automatycznie algorytmem wyszukiwania binarnego.

  Podana klasa posiada także swoje własne metody takie jak:

  - **printArray()** - wyświetlająca na ekranie wartości przechowywanej przez obiekt tablicy.
  - **printPosition()** - wyświetlająca na ekranie pozycję na której algorytm wyszukujący znalazł podaną wartość szukaną.
  - **printDominantOperations()** - wyświetlająca na ekranie liczbę operacji dominujących.
  - **printAlgorithm()** - wyświetlająca na ekranie słowną nazwę wybranego algorytmu wyszukującego.
  - **getArray()** - zwracająca przekazaną do obiektu tablicę.
  - **getPosition()** - zwracającą pozycję na której algorytm wyszukujący znalazł podaną wartość szukaną.
  - **getDominantOperations()** - zwracająca liczbę operacji dominujących.
  - **getAlgorithm()** - zwracającą słowną nazwę wybranego algorytmu wyszukującego.
  - **setArray(array, size)** - przypisującą obiektowi nową tablicę - array o podanej długości - size.

– **setSearchingValue(searchingValue)** - przypisującą obiektowi nową wartość szukaną - searchingValue.

– **search(algorithm)** - przeszukującą przechowywaną przez obiekt tablicę za pomocą wskazanego w algorytmu wyszukującego - algorithm. Powyższy parametr może przyjąć następujące wartości:

  * **0** - algorytm wyszukiwania liniowego
  * **1** - algorytm wyszukiwania binarnego

  Domyślnie nie musimy podawać żadnego parametru, wtedy obiekt zostanie przeszukany automatycznie algorytmem wyszukiwania binarnego.

2.4 Opis algorytmów oraz ich implementacji

2.4.1 Algorytm sortowania bąbelkowego [2]

Algorytm sortowania bąbelkowego jest najprostszym algorytmem sortowania, który polega na wielokrotnej zamianie sąsiednich elementów, jeśli są w nieprawidłowej kolejności. Algorytm ten w podstawowej wersji posiada bardzo dużą złożoność czasową ($O(n^2)$) dla średniego i najgorszego przypadku przez co nie jest on odpowiednim wyborem dla dużych zbiorów danych. Złożoność tę można jednak nieco zredukować używając flagi informującej, czy w danej iteracji pętli doszło do jakiejkolwiek zamiany elementów, dzięki czemu dla danych już posortowanych złożoność czasowa tego algorytmu wynosi $O(n)$. Algorytm sortowania bąbelkowego jest stabilnym algorytmem, ponieważ elementy o tej samej wartości zachowują w tablicy końcowej tę samą kolejność co w tablicy wejściowej.

Implementacja algorytmu w języku C/C++:

```
void SortingAlgorithms::bubbleSort(int* array, int size) {
// przekazujemy do algorytmu tablicę do posortowania oraz jej długość
        int temp;
        // deklarujemy zmienną pomocniczą, który posłuży do zamiany elementów miejscami
        bool flag;
        // deklarujemy flagę, która przyjmować będzie wartość false, jeżeli nastąpi jakakolwiek zamiana
        for (int i = 0; i < size; i++) {
        // pętla pozwalająca algorytmowi poruszanie się po tablicy
                flag = true;
                // zmiana wartości flagi na domyślną - true (stan w którym nie nastąpiła żadna zamiana)
                for (int j = 0; j < size - i - 1; j++) {
                // pętla pozwalająca algorytmowi porównywanie sąsiednich wartości ze sobą
                        this->dominantOperations++;
                        // zliczanie operacji dominującej
                        if (array[j] > array[j + 1]) {
                        // warunek porównujący dwie wartości tablicy
                                temp = array[j];
                                array[j] = array[j + 1];
                                array[j + 1] = temp;
                                // instrukcje zamieniające oba elementy miejscami
                                flag = false;
                                // zmiana wartości flagi na false, ponieważ nastąpiła zamiana elementów
                        }
                }
                if (flag)
                        break;
                // warunek sprawdzający, czy podczas aktualnej iteracji pętli nastąpiła zamiana elementów,
                // jeśli nie to zostanie wykonana instrukcja break, która zakończy działanie algorytmu
        }
}
```

2.4.2 Algorytm sortowania przez wstawianie [3]

Algorytm sortowania przez wstawianie jest prostym algorytmem sortującym działającym na podobnej zasadzie jak to w jaki sposób układamy karty w dłoni podczas gry. Cała tablica zostaje podzielona na część nieposortowaną i na część posortowaną. Sukcesywnie wartość z części nieposortowanej zostają porównywane z wartościami z części posortowanej i układane w odpowiednie miejsca w tablicy aż całość nie będzie ułożona w kolejności rosnącej. Algorytm ten podobnie jak algorytm sortowania bąbelkowego jest algorytmem stabilnym posiadającym złożoność czasową $O(n^2)$ dla średniego i najgorszego przypadku, oraz $O(n)$ dla przypadku optymistycznego.

Implementacja algorytmu w języku C/C++:

```
void SortingAlgorithms::insertionSort(int* array, int size) {
// przekazujemy do algorytmu tablicę do posortowania oraz jej długość
        int dominantOperations[2] = { 0,0 };
        // tworzymy tablicę zawierającą wartości operacji dominujących
        bool flag;
        // deklarujemy flagę, która przyjmować będzie wartość true, jeżeli zostanie wykonana
        for (int i = 1; i < size; i++) {
        // pętla pozwalająca algorytmowi poruszanie się po tablicy
                flag = false;
                // zmiana wartości flagi na domyślną - false (stan w którym nie nastąpiła wejście wewnątrz pętli)
                int key = array[i];
                // deklaracja i przypisanie wartości kluczowej (tej do której porównujemy elementy tablicy)
                int j = i - 1;
                // deklaracja i przypisanie wartości j w celu późniejszego porównaniu i zamiany obu elementów
                dominantOperations[0]++;
                // zliczanie operacji dominującej
                while (j >= 0 && array[j] > key) {
                    // pętlą wykonująca się dopóki indeks wartości porównywanej będzie nieujemny
                    // oraz wartość elementu pod tym indeksem będzie większa od wartości kluczowej
                        flag = true;
                        // zmiana wartości flagi true, ponieważ nastąpiło wejście wewnątrz pętli
                        dominantOperations[1]++;
                        // zliczanie operacji dominującej
                        array[j + 1] = array[j];
                        // przypisanie wartości elementu array[j] elementowi array[j+1]
                        j = j - 1;
                        // pomniejszenie wartości j o 1
                }
                if (flag)
                        dominantOperations[1]--;
                // warunek sprawdzający, czy podczas aktualnej iteracji pętli nastąpiło wykonanie pętli while
                // jeśli tak to wartość operacji dominujących zostanie pomniejszona o 1,
                // ponieważ została ona uwzględniona już wcześniej
                array[j + 1] = key;
                // wstawienie elementu kluczowego w poprawne miejsce
        }
        this->dominantOperations = dominantOperations[0] + dominantOperations[1];
        // zliczenie całkowiej ilości operacji dominujących do zmiennej klasowej
}
```

2.4.3 Algorytm sortowania przez scalanie [5]

Algorytm sortowania przez scalanie jest algorytmem sortowania opartym o technikę **dziel i zwyciężaj**. Dzieli on wejściową tablicę na dwie połowy, poprzez rekurencję wywołuję samego siebie dla obu z tych połów i tak aż do uzyskania tablic jednoelementowych. Następnie następuje scalanie obu tablic poprzez porównanie występujących w nich elementów. Do tego celu wykorzystuje zaimplementowaną funkcję merge(). Algorytm ten jest algorytmem stabilnym posiadającym złożoność czasową $O(n * log_2 n)$ dla każdego przypadku.

Implementacja algorytmu w języku C/C++:

```
void SortingAlgorithms::mergeSort(int* array, int left, int right) {
// przekazujemy do algorytmu tablicę do posortowania oraz indeks początkowy i końcowy tablicy
        if (left < right)
        // warunek sprawdzający, czy indeks początku tablicy jest mniejszy od indeksu końcowego
        {
                int middleValue = left + (right - left) / 2;
                // deklaracja i przypisanie wartości środkowej tablicy
                // (to ona wskazuje nam podział na dwie tablice)
                mergeSort(array, left, middleValue);
                // wywołanie rekurencyjne funkcji dla pierwszej połowy tablicy
                mergeSort(array, middleValue + 1, right);
                // wywołanie rekurencyjne funkcji dla drugiej połowy tablicy
                merge(array, left, middleValue, right);
                // wywołanie funkcji scalającej obie tablice
        }
}
void SortingAlgorithms::merge(int* array, int left, int middleValue, int right) {
// przekazujemy do funkcji tablicę do której przepisane zostaną wartości,
// indeks początkowy, środkowy oraz końcowy scalanej tablicy
        int i, j, k;
        // deklaracja zmiennych do poruszania się po tablicach
        // to one będą definiować w jaki sposób wartości będą porównywane a tablice scalane
        int n1 = middleValue - left + 1;
        // deklaracja i przypisanie zmiennej z ilością elementów pierwszej tablicy
        int n2 = right - middleValue;
        // deklaracja i przypisanie zmiennej z ilością elementów drugiej tablicy
        int* Left = new int[n1];
        // deklaracja tablicy pomocniczej dla pierwszej połowy tablicy
        int* Right = new int[n2];
        // deklaracja tablicy pomocniczej dla drugiej połowy tablicy
        for (i = 0; i < n1; i++)
                Left[i] = array[left + i];
        // pętla przepisująca wartości do pierwszej tablicy pomocniczej
        for (j = 0; j < n2; j++)
                Right[j] = array[middleValue + 1 + j];
        // pętla przepisująca wartości do drugiej tablicy pomocniczej
        i = 0;
        // przypisanie zmiennej "i" indeksu pierwszego elementu dla pierwszej tablicy pomocniczej
        j = 0;
        // przypisanie zmiennej "j" indeksu pierwszego elementu dla pierwszej tablicy pomocniczej
        k = left;
        // przypisanie zmiennej "k" indeksu pierwszego elementu wyjściowej tablicy do przepisania wartości
```

```
bool flag = false;
// deklaracja i przypisanie wartości flagi na domyślną - false
// (stan w którym nie nastąpiło wykonanie kodu wewnątrz pętli)
while (i < n1 && j < n2)
// pętla wykonująca się dopóki indeksy obu pomocniczych tablic są mniejsze od ich długości
{
    flag = true
    // zmiana wartości flagi na true (stan w którym nastąpiło wykonanie pętli wewnętrznej)
        this->dominantOperations++;
        // zliczanie operacji dominującej
        if (Left[i] <= Right[j])
        // warunek sprawdzający czy element pierwszej tablicy pomocniczej jest mniejszy
        // lub równy elementowi drugiej tablicy pomocniczej
        {
                array[k] = Left[i];
                // przepisanie elementu pierwszej tablicy pomocniczej do tablicy wyjściowej
                i++;
                // inkrementacja indeksu pierwszej tablicy pomocniczej
        }
        else
        {
                array[k] = Right[j];
                // przepisanie elementu drugiej tablicy pomocniczej do tablicy wyjściowej
                j++;
                // inkrementacja indeksu drugiej tablicy pomocniczej
        }
        k++;
        // inkrementacja indeksu tablicy wyjściowej
}
if(flag)
    this->dominantOperations--;
// warunek sprawdzający, czy nastąpiło wykonanie pętli while, jeśli tak to wartość operacji
// dominujących zostanie pomniejszona o 1, ponieważ została ona uwzględniona już wcześniej
while (i < n1)
// pętla wykonująca się dopóki indeks pierwszej tablicy pomocniczej jest mniejszy od jej długości
// ma ona na celu przepisanie pozostałych wartości z pierwszej tablicy pomocniczej do tablicy wyjściowej
{
        array[k] = Left[i];
        // przepisanie elementu pierwszej tablicy pomocniczej do tablicy wyjściowej
        i++;
        // inkrementacja indeksu pierwszej tablicy pomocniczej
        k++;
        // inkrementacja indeksu tablicy wyjściowej
}
```

```
        while (j < n2)
        // pętla wykonująca się dopóki indeks drugiej tablicy pomocniczej jest mniejszy od jej długości
        // ma ona na celu przepisanie pozostałych wartości z drugiej tablicy pomocniczej do tablicy wyjściowej
        {
                array[k] = Right[j];
                // przepisanie elementu drugiej tablicy pomocniczej do tablicy wyjściowej
                j++;
                // inkrementacja indeksu drugiej tablicy pomocniczej
                k++;
                // inkrementacja indeksu tablicy wyjściowej
        }
}
```

2.4.4 Algorytm wyszukiwania liniowego [4]

Algorytm wyszukiwania liniowego jest najprostszym algorytmem wyszukiwania polegającym na sekwencyjnym przeszukiwaniu całej tablicy pod względem szukanego elementu. Algorytm ten jako wynik działania zwraca indeks tablicy na którym znalazł szukaną wartość lub -1 w przypadku nieznalezienia elementu. Złożoność czasowa tego algorytmu wynosi $O(n)$ w przypadku najgorszym i $O(1)$ w przypadku, gdy szukana wartość jest na początku tablicy.

Implementacja algorytmu w języku C/C++:

```
int SearchingAlgorithms::linearSearch(int* array, int searchingValue, int size) {
// przekazujemy do algorytmu tablicę, którą będziemy przeszukiwać, szukany element oraz długość tej tablicy
        int position = -1;
        // deklaracja i przypisanie zmiennej przechowującej pozycje znalezionego elementu
        for (int i = 0; i < size; i++) {
        // pętla pozwalająca algorytmowi poruszanie się po tablicy
                this->dominantOperations++;
                // zliczanie operacji dominującej
                if (array[i] == searchingValue) {
                // warunek sprawdzający czy element tablicy jest równy szukanej wartości
                        position = i;
                        // przypisanie pozycji indeksu do zmiennej
                        break;
                        // przerwanie działania pętli
                }
        }
        return position;
        // zwrócenie pozycji szukanego elementu
}
```

2.4.5 Algorytm wyszukiwania binarnego [1][6]

Algorytm wyszukiwania binarnego jest algorytmem opartym o technikę **dziel i zwyciężaj** oraz podobnie jak poprzednik, jako wynik działania zwraca indeks tablicy na którym znalazł szukaną wartość lub -1 w przypadku nieznalezienia elementu. Algorytm ten dzieli tablicę na coraz mniejsze przedziały aż do uzyskania tablic jednoelementowych. Technika dziel i zwyciężaj ma tu na celu stwierdzać w którym z przedzielonych tablic powinien znajdować się szukany element. Należy pamiętać, że przekazana tablica musi być już posortowana w kolejności rosnącej, ponieważ to dzięki temu uporządkowaniu elementów w tablicy dany przedział jest zawężany. Złożoność czasowa tego algorytmu wynosi $O(log_2 n)$ w przypadku najgorszym oraz $O(1)$ w przypadku, gdy szukana wartość znajduje się w połowie tablicy, co plasuje go w czołówce algorytmów wyszukujących.

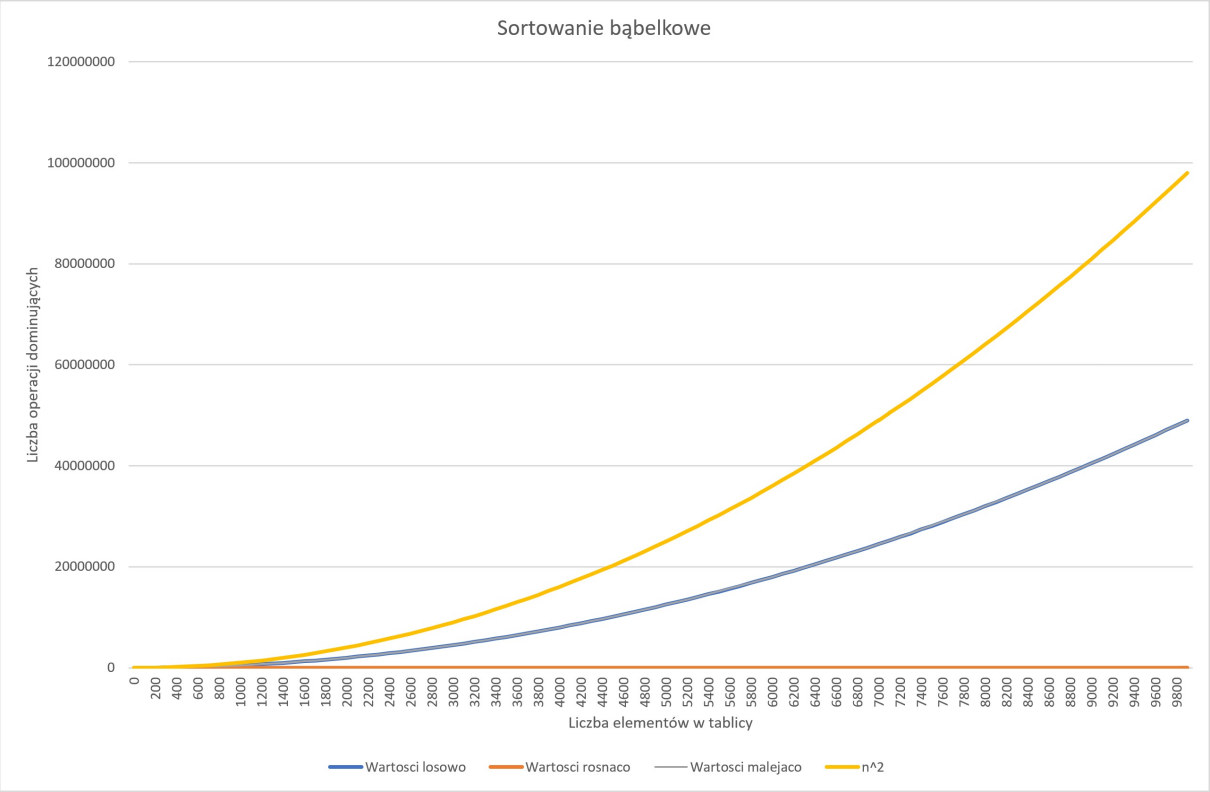
Implementacja algorytmu w języku C/C++:

```
int SearchingAlgorithms::binarySearch(int* array, int searchingValue, int left, int right) {
// przekazujemy do algorytmu tablicę, którą będziemy przeszukiwać, szukany element oraz jej skrajne indeksy
        while (left <= right) {
        // pętla wykonująca się dopóki początkowy indeks przeszukiwanej tablicy jest mniejszy lub równy końcowemu
                int middle = left + (right - left) / 2;
                // deklaracja i przypisanie indeksu środka tablicy
                if (array[middle] == searchingValue)
                        return middle;
                // warunek sprawdzający, czy środkowa wartość tablicy jest równa szukanemu elementowi,
                // jeżeli tak, to algorytm zwróci pozycję znalezionego elementu
                this->dominantOperations++;
                // zliczanie operacji dominującej
                if (array[middle] < searchingValue)
                // warunek sprawdzający czy środkowa wartość jest mniejsza od szukanej wartości
                        left = middle + 1;
                        // przypisanie początkowemu indeksowi tablicy, indeks środkowy tablicy powiększony o 1
                else
                        right = middle - 1;
                        // przypisanie końcowemu indeksowi tablicy, indeks środkowy tablicy pomniejszony o 1
        }
        return -1;
        // w przypadku nie znalezienia wartości zwrócenie wartości -1
}
```

2.5 Prezentacja wyników

2.5.1 Sortowanie bąbelkowe

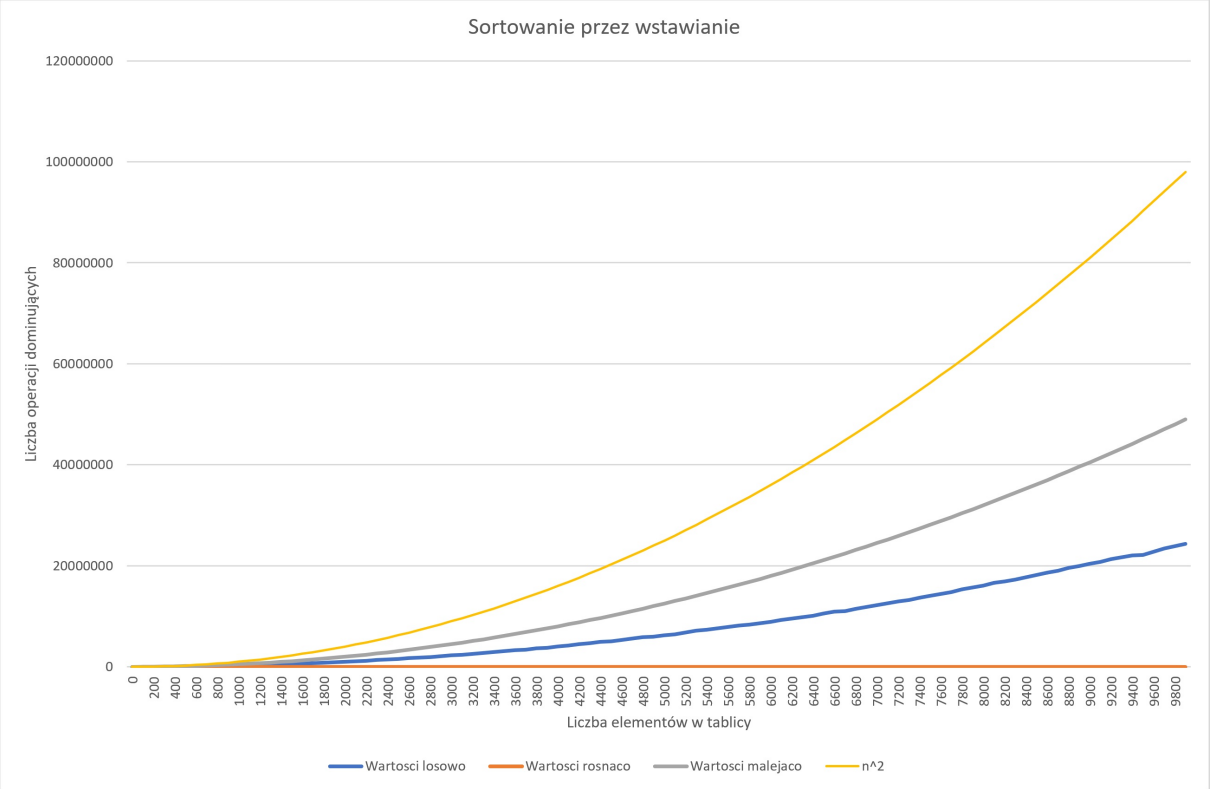
Jak możemy zauważyć na poniższym wykresie algorytm sortowania bąbelkowego w wersji zoptymalizowanej wykonuje najwięcej operacji dominujących w przypadku przekazania mu tablicy z danymi losowymi oraz posortowanymi malejąco. Dla danych posortowanych rosnąco liczba tych operacji wynosi $n - 1$.



Rysunek 2.1: Wykres przedstawiający porównanie działania algorytmu sortowania bąbelkowego dla różnego typu danych wejściowych

2.5.2 Sortowanie przez wstawianie

Jak możemy zauważyć na poniższym wykresie algorytm sortowania przez wstawianie, podobnie jak algorytm sortowania bąbelkowego wykonuje najwięcej operacji dominujących w przypadku przekazania mu tablicy z danymi posortowanymi malejąco, z tą różnicą, że w przypadku danych losowych liczba tych operacji je średnio o połowę mniejsza. Dla danych posortowanych rosnąco liczba tych operacji wynosi $n - 1$.



Rysunek 2.2: Wykres przedstawiający porównanie działania algorytmu sortowania przez wstawianie dla różnego typu danych wejściowych

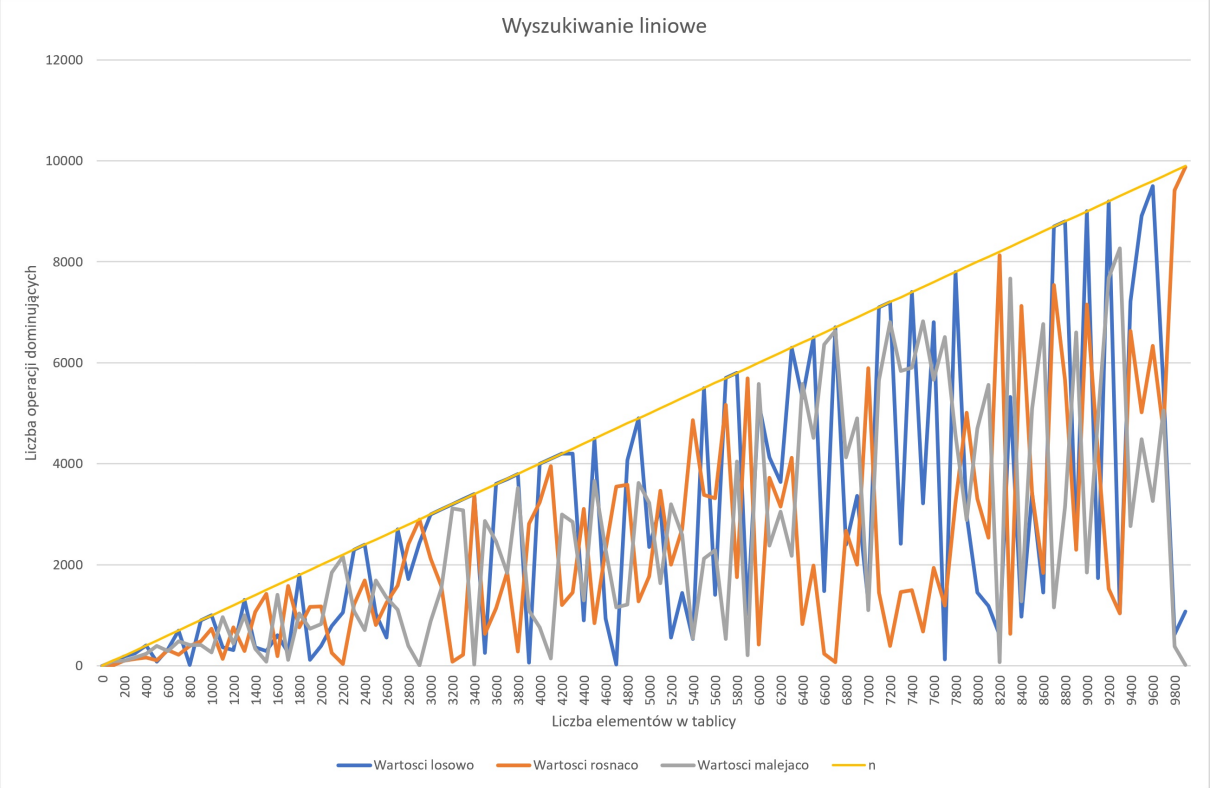
2.5.3 Sortowanie przez scalanie

Jak możemy zauważyć na poniższym wykresie algorytm sortowania przez scalanie wykonuje najwięcej operacji dominujących w przypadku przekazania mu tablicy z danymi losowymi, w przypadku danych posortowanych rosnąco i malejąco liczba tych operacji jest mniejsza i się przeplata. Wszystkie serie danych mieszczą się w pożądanym zakresie $n * log_2 n$ dla tego algorytmu.

Rysunek 2.3: Wykres przedstawiający porównanie działania algorytmu sortowania przez scalanie dla różnego typu danych wejściowych

2.5.4 Wyszukiwanie liniowe

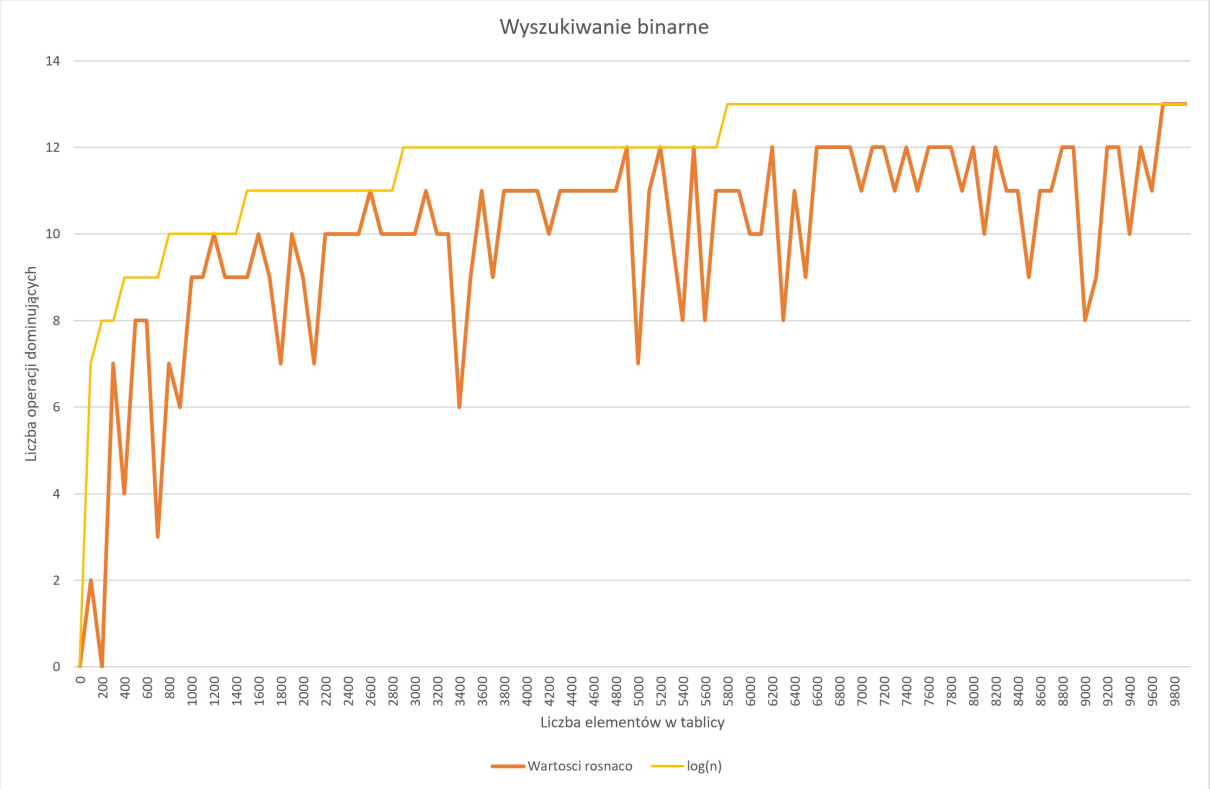
Jak możemy zauważyć na poniższym wykresie algorytm wyszukiwania liniowego wyszukuje losową wartość wykonując różną liczbę operacji dominujących, lecz nigdy nie przekraczającą n. Wykresy dla danych posortowanych rosnąco i malejąco są względem siebie odwrotnie proporcjonalne.



Rysunek 2.4: Wykres przedstawiający porównanie działania algorytmu wyszukiwania liniowego dla różnego typu danych wejściowych

2.5.5 Wyszukiwanie binarne

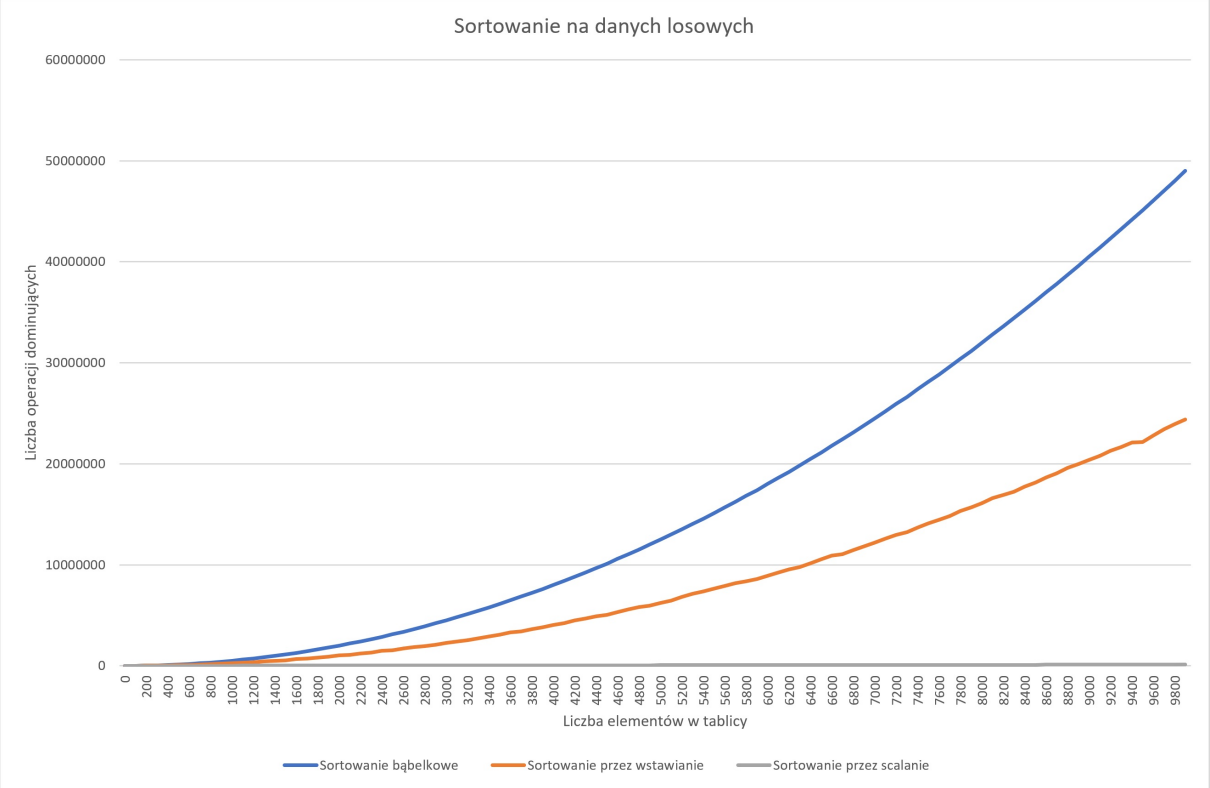
Jak możemy zauważyć na poniższym wykresie algorytm wyszukiwania binarnego wyszukuje losową wartość wykonując różną liczbę operacji dominujących, lecz nigdy nie przekraczającą $log_2 n$.



Rysunek 2.5: Wykres przedstawiający działanie algorytmu wyszukiwania binarnego dla danych posortowanych rosnąco

2.5.6 Porównanie algorytmów sortujących dla danych losowych

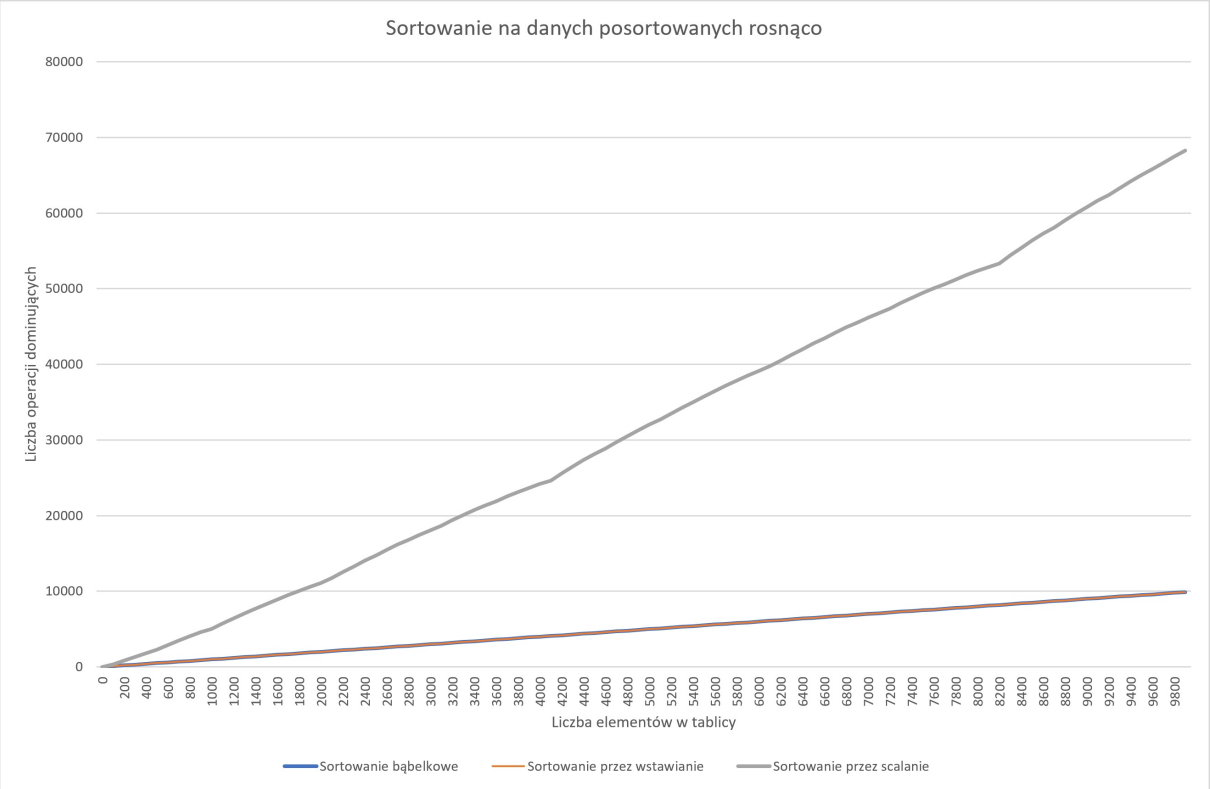
Jak możemy zauważyć na poniższym wykresie algorytm sortowania przez scalanie deklasuje pozostałe algorytmy (algorytm sortowania bąbelkowego i sortowania przez wstawianie) ze względu na liczbę operacji dominujących potrzebnych do posortowania tablic różnej długości.



Rysunek 2.6: Wykres przedstawiający porównanie algorytmów sortujących dla danych losowych

2.5.7 Porównanie algorytmów sortujących dla danych posortowanych rosnąco

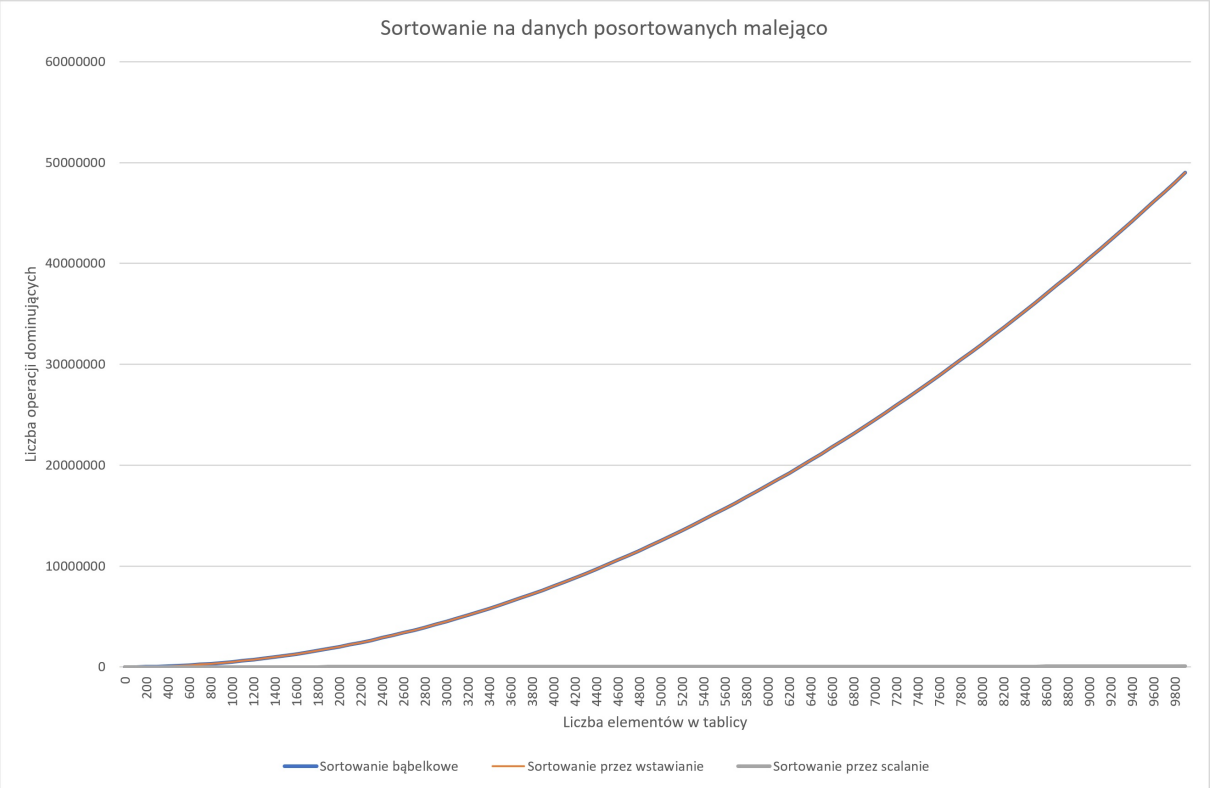
Jak możemy zauważyć na poniższym wykresie algorytm sortowania przez scalanie wykonuje większą liczbę operacji dominujących potrzebnych do posortowania tablic różnej długości od pozostałych algorytmów (algorytmu sortowania bąbelkowego i sortowania przez wstawianie). Wynika to z tego, że jego złożoność obliczeniowa jest stała dla wszystkich typów danych wejściowych i wynosi $n * log_2 n$. Liczba ta jest większa od złożoności obliczeniowej dla przypadku optymistycznego pozostałych algorytmów - n.



Rysunek 2.7: Wykres przedstawiający porównanie algorytmów sortujących dla danych posortowanych rosnąco

2.5.8 Porównanie algorytmów sortujących dla danych posortowanych malejąco

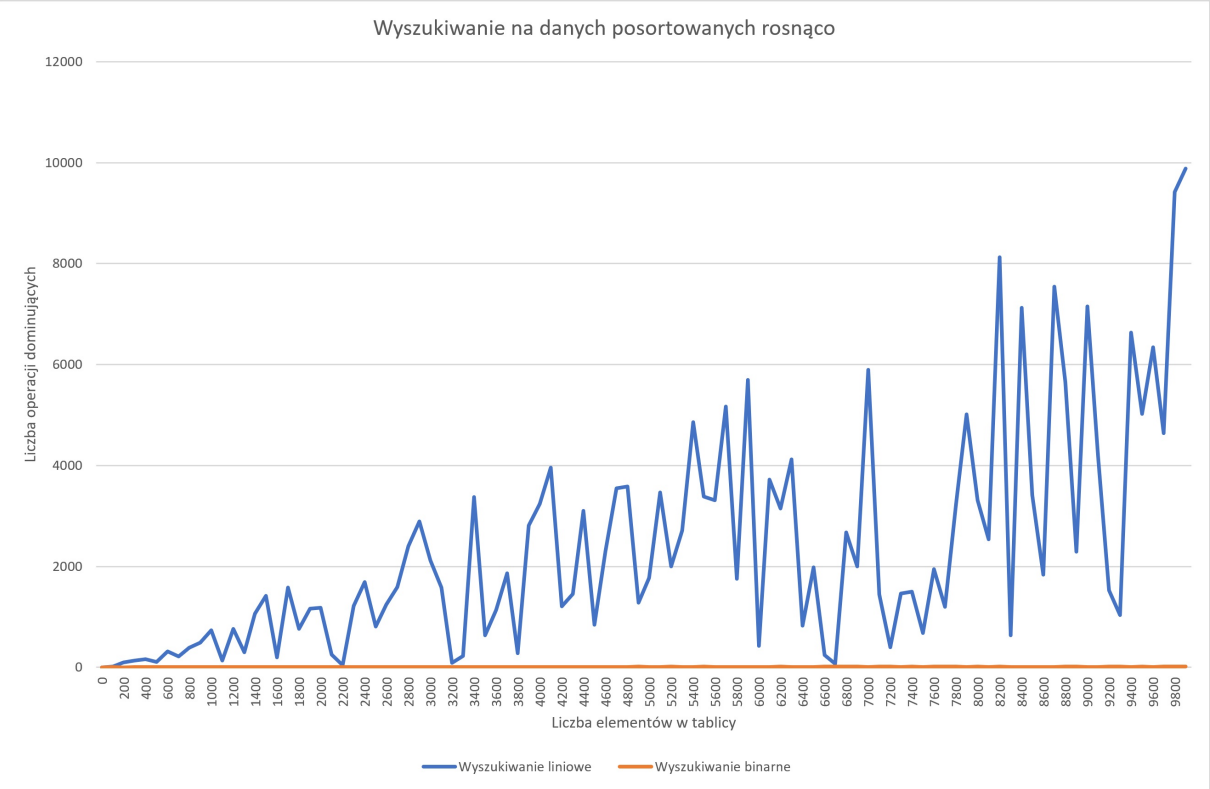
Jak możemy zauważyć na poniższym wykresie algorytm sortowania przez scalanie deklasuje pozostałe algorytmy (algorytm sortowania bąbelkowego i sortowania przez wstawianie) ze względu na liczbę operacji dominujących potrzebnych do posortowania tablic różnej długości.



Rysunek 2.8: Wykres przedstawiający porównanie algorytmów sortujących dla danych posortowanych malejąco

2.5.9 Porównanie algorytmów wyszukiwania dla danych posortowanych rosnąco

Jak możemy zauważyć na poniższym wykresie algorytm wyszukiwania liniowego wypada słabo pod względem liczby operacji dominujących potrzebnych do posortowania tablic różnej długości w stosunku do algorytmu sortowania binarnego. Porównanie to było możliwe tylko dla danych posortowanych rosnąco ze względu na specyfikę algorytmu wyszukiwania binarnego, który przyjmuje tylko taki typ danych.



Rysunek 2.9: Wykres przedstawiający porównanie algorytmów wyszukiwania dla danych posortowanych rosnąco

Rozdział 3

Podsumowanie projektu

3.1 Wnioski

Podsumowując wszystkie powyższe wykresy dochodzimy do wniosku iż różne algorytmy sortowania mają swoje zastosowania dla różnych typów danych, podobnie jak algorytmy wyszukiwania.

Algorytm sortowania bąbelkowego podobnie jak algorytm sortowania przez wstawianie są proste w implementacji i z powodzeniem mogą służyć do sortowania małych serii danych, lecz w przypadku większej ilości danych należałoby zastanowić się nad implementacją algorytmu sortowania przez scalanie, który zdecydowanie góruje nad dwoma powyższymi.

Natomiast w przypadku konieczności wyszukania wartości w tablicy algorytm wyszukiwania binarnego jest zdecydowanie szybszy od algorytmu wyszukiwania liniowego, lecz jego implementacja bywa bardzo kosztowna, ponieważ tablica musi być przedwcześnie posortowana. Atutem algorytmu wyszukiwania liniowego jest to, że działa on niezależnie od kolejności danych w tablicy.

Analizując powyższe wykresy możemy dojść także do wniosku iż wszystkie algorytmy wykorzystujące technikę **dziel i zwyciężaj** są optymalniejsze w każdym przypadku.

Spis rysunków

Jakub Chodziński (65530)
IID-P 2021/2022, SW01
Algorytmy i struktury danych
20 maja 2022 roku

Wykaz źródeł

[1] GeeksforGeeks. *Binary Search*. URL: https://www.geeksforgeeks.org/binary-search/.

[2] GeeksforGeeks. *Bubble Sort*. URL: https://www.geeksforgeeks.org/bubble-sort/.

[3] GeeksforGeeks. *Insertion Sort*. URL: https://www.geeksforgeeks.org/insertion-sort/.

[4] GeeksforGeeks. *Linear Search*. URL: https://www.geeksforgeeks.org/linear-search/.

[5] GeeksforGeeks. *Merge Sort*. URL: https://www.geeksforgeeks.org/merge-sort/.

[6] Wikipedia. *Wyszukiwanie binarne*. URL: https://pl.wikipedia.org/wiki/Wyszukiwanie_binarne.