WYDZIAŁ
MATEMATYKI
I FIZYKI STOSOWANEJ
POLITECHNIKI RZESZOWSKIEJ

Projekt 2
Algorytmy i Struktury danych

Weronika Zagaja
Inżynieria i analiza danych
Nr indeksu 166718

08.01.2021r

Spis treści

1. Wstęp

Celem projektu było dokonanie implementacji kodu, który polegał na napisaniu programu, którego zadaniem była realizacja czynności, opisanych wcześniej w instrukcji. Dodatkowo projekt zawiera schematy blokowe oraz pseudokody. Program został napisany w środowisku Code::Blocks IDE w języku C++. Środowisko to posiada otwarty kod źródłowy i kluczową zaletą tego oprogramowania jest jego wieloplatformowość.

2. Treść zadania do wykonania

Problematyka projektu polegała na implementacji sortowania liczb przez wstawianie oraz sortowania kubełkowego. Głównymi cechami programu są:

a) program powinien mieć możliwość odczytywania danych wejściowych z pliku tekstowego i zapisu posortowanego ciągu wynikowego również do pliku tekstowego
b) na potrzeby wykonywanych testów należy zaimplementować funkcję generującą ”losowe” ciągi elementów (o zadanej długości) i zapisującą je do pliku tekstowego
c) założyć, że sortowanymi elementami są liczby całkowite z przedziału [0, N], gdzie N powinno być ”odpowiednio dużym” parametrem ustalonym wewnątrz programu
d) kod powinien być opatrzony stosownymi komentarzami.

Na początku realizowania projektu, powinna zostać wykonana definicja problemu np.: w postaci modelu matematycznego, następnie koncepcja rozwiązania i zapis algorytmu w postaci schematu blokowego i pseudokodu w celu ułatwienia implementacji kodu. Na końcu należało przejść do urzeczywistnienia kodu w wybranym środowisku programistycznym.

3. Teoretyczne postawy sortowań

Sortowanie przez wstawianie

Algorytm ***sortowania przez wstawianie*** można porównać do sposobu układania kart pobieranych z talii. Najpierw bierzemy pierwszą kartę. Następnie pobieramy kolejne, aż do wyczerpania talii. Każdą pobraną kartę porównujemy z kartami, które już trzymamy w ręce i szukamy dla niej miejsca przed pierwszą kartą starszą znajdziemy takie miejsce, rozsuwamy karty i nową wstawiamy na przygotowane w ten sposób miejsce sortowanie przez wstawianie, ang. **Insertion Sort/Insert Sort.** Jeśli nasza karta jest najstarsza, to umieszczamy ją na samym końcu. Tak porządkujemy karty.

Algorytm sortowania przez wstawianie będzie składał się z dwóch pętli.

a) Pętla główna (zewnętrzna) symuluje pobieranie kart, czyli w tym wypadku elementów zbioru. Odpowiednikiem kart na ręce jest tzw. lista uporządkowana, którą sukcesywnie będziemy tworzyli na końcu zbioru.
b) Pętla sortująca (wewnętrzna) szuka dla pobranego elementu miejsca na liście uporządkowanej. Jeśli takie miejsce zostanie znalezione, to elementy listy są odpowiednio rozsuwane, aby tworzyć miejsce na nowy element i element wybrany przez pętlę główną trafia tam.

W ten sposób lista uporządkowana rozrasta się. Jeśli na liście uporządkowanej nie ma elementu większego od wybranego, to element ten trafia na koniec listy.

Sortowanie zakończymy, gdy pętla główna wybierze wszystkie elementy zbioru.

Najważniejszą operacją w opisywanym algorytmie sortowania jest wstawianie wybranego elementu na listę uporządkowaną. Zasady są następujące:

1. Na początku sortowania lista uporządkowana zawiera tylko jeden, ostatni element zbioru. Jednoelementowa lista jest zawsze uporządkowana.
2. Ze zbioru zawsze wybieramy element leżący tuż przed listą uporządkowaną. Element ten zapamiętujemy w zewnętrznej zmiennej. Miejsce, które zajmował, możemy potraktować jak puste.
3. Wybrany element porównujemy z kolejnymi elementami listy uporządkowanej.
4. Jeśli natrafimy na koniec listy, element wybrany wstawiamy na puste miejsce - lista rozrasta się o nowy element.
5. Jeśli element listy jest większy od wybranego, to element wybrany wstawiamy na puste miejsce - lista rozrasta się o nowy element.
6. Jeśli element listy nie jest większy od wybranego, to element listy przesuwamy na puste miejsce. Dzięki tej operacji puste miejsce wędruje na liście przed kolejny element. Kontynuujemy porównywanie, aż wystąpi sytuacja z punktu 4 lub 5.

Sortowanie kubełkowe

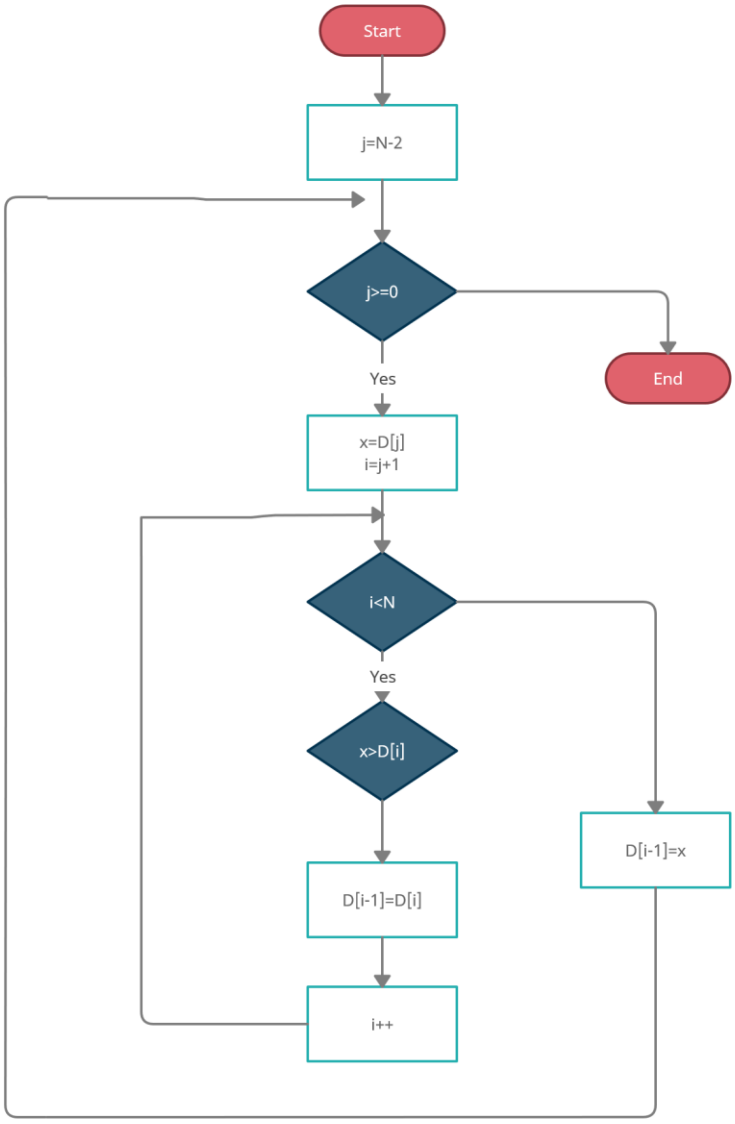
Algorytm sortowania kubełkowego pozwala sortować zbiory liczb całkowitych - najlepiej o dużej ilości elementów, lecz o małym zakresie wartości. Zasada działania jest następująca:

1. Określamy zakres wartości, jakie mogą przyjmować elementy sortowanego zbioru. Niech w_{min} oznacza najmniejszą wartość, a w_{max} niech oznacza wartość największą.
2. Dla każdej możliwej wartości przygotowujemy kubełek-licznik, który będzie zliczał ilość wystąpień tej wartości w sortowanym zbiorze. Liczba liczników jest równa $(w_{max} - w_{min} + 1)$. Każdy licznik jest początkowo ustawiony na wartość zero.
3. Przeglądamy kolejno elementy zbioru od pierwszego do ostatniego. Dla każdego elementu zbioru zwiększamy o jeden zawartość licznika o numerze równym wartości elementu. Na przykład, jeśli kolejny element zbioru ma wartość 3, to zwiększamy licznik o numerze 3. W efekcie po przeglądnięciu wszystkich elementów zbioru liczniki będą zawierały ilość wystąpień każdej z możliwych wartości. Jeśli dany licznik zawiera 0, to wartość równa numerowi licznika w zbiorze nie występuje. Inaczej wartość ta występuje tyle razy, ile wynosi zawartość jej licznika.
4. Przeglądamy kolejne liczniki zapisując do zbioru wynikowego ich numery tyle razy, ile wynosi ich zawartość. Zbiór wyjściowy będzie posortowany.

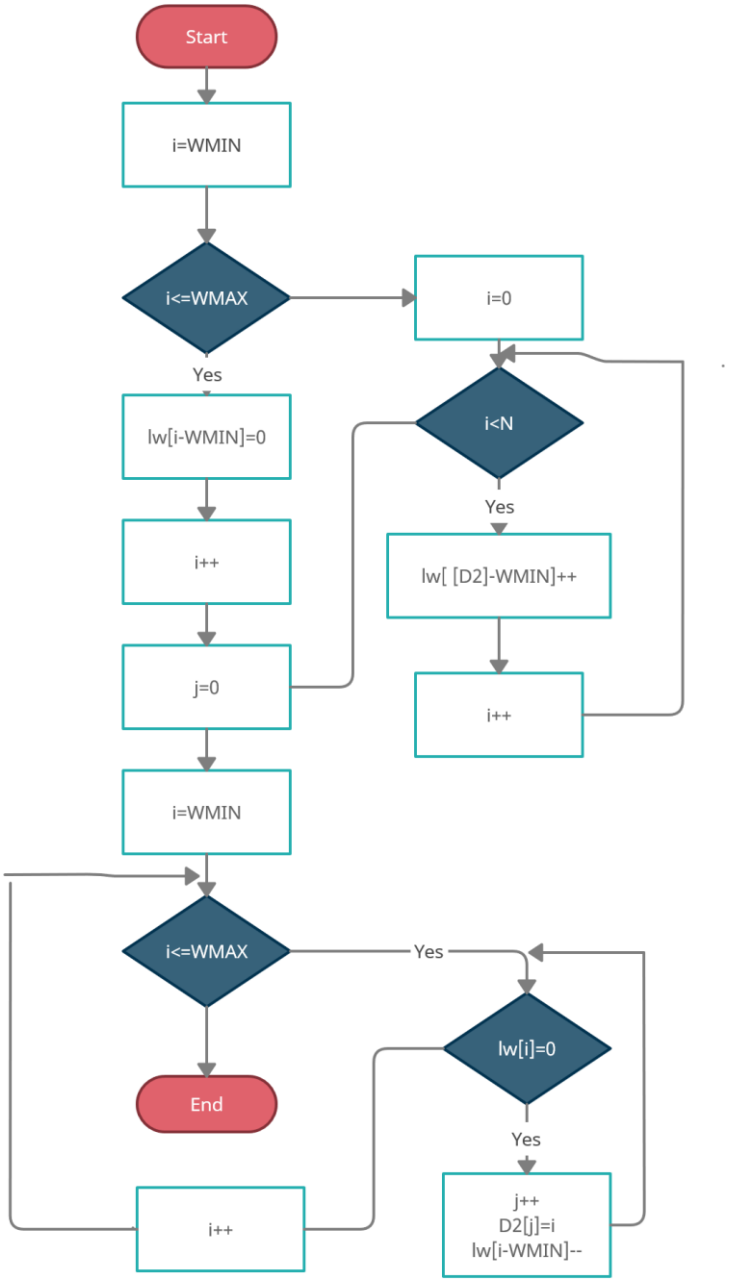
4. Schematy blokowe algorytmów

Schemat blokowy jest podglądową formą graficznego przedstawienia algorytmu. Tworzy się go korzystając ze ściśle określonego zbioru figur geometrycznych oraz stosując ustalone reguły ich łączenia. Każdy schemat blokowy musi być spójny tzn., że do bloku typu *początek* do bloku typu *koniec* musi prowadzić przynajmniej jedna droga.

Poniżej zostały przedstawione schematy blokowe dla obu algorytmów:



Rysunek 1: Schemat blokowy "sortowanie przez wstawianie"



Rysunek 2 Schemat blokowy "sortowanie kubełkowe"

Odzwierciedlone graficznie schematy blokowe, przedstawionego powyżej problemu zawierają wewnątrz bloków w pozorowany sposób zapis występujących w algorytmie operacji, takich jak:

a) arytmetyczne,
b) logiczne,
c) operacje wejścia i wyjścia,
d) warunki, od których zależą decyzje co do kolejności wykonywania działań.

5. Pseudokody

Pseudokodem nazywamy taki sposób zapisu algorytmu, który zachowując strukturę charakterystyczną dla kodu zapisanego w języku programowania, rezygnuje ze ścisłych reguł składniowych na rzecz prostoty i czytelności. W pseudokodzie nie istnieją standardy zapisu, opiera się jedynie na składni istniejących języków programowania.

Poniższe pseudokody miały na celu przedstawić skrót myślowy i pomóc w realizacji danego programu. Zostały w nim opisane krok po kroku potrzebne funkcje, aby prawidłowo wykonać implementację sortowań.

```
1. Dopóki j=N-2 do j>=0
                zmniejsz j o 1
2.Jeżeli( j>=0 ) wykonaj
                x=D[j]
                i=j+1
        w przeciwnym wypadku
                 zakończ
3.Jeżeli (i<N) i (x>D[j]) wykonaj:
                D[i-1]=D[i]
                i=i+1
        w przeciwnym wypadku
                D[i-1]=x
```

Rysunek 3 Pseudokod dla sortowania przez wstawianie

```
1. Dopóki i=WMIN do i <= WMAX
                zwiększ i o 1
2. Jeżeli ( i<= WMAX) wykonuj:
                lw[i-WMIN} = 0
         Dopóki i=WMIN do i <= WMAX
                zwiększ i o 1
        w przeciwnym wypadku
        dla i=0
        jeżeli i<N wykonaj:
                lw[[D2]- WMIN] zwiększ o 1
                i zwiększ o 1

2. Dla j=0
3. Jeżeli (i <=WMAX) wykonuj:
                zakończ
         w przeciwnym wypadku
                lw[i]=0
4. Jeżeli (lw[i]=0) wykonuj:
                zwiększ j o 1
                D2[j]=i
                zmniejsz lw[i-WMIN] o 1
        w przeciwnym wypadku
                zwiększ i o 1
```

Rysunek 4 Pseudokod dla sortowania kubełkowego

6. Kod programu

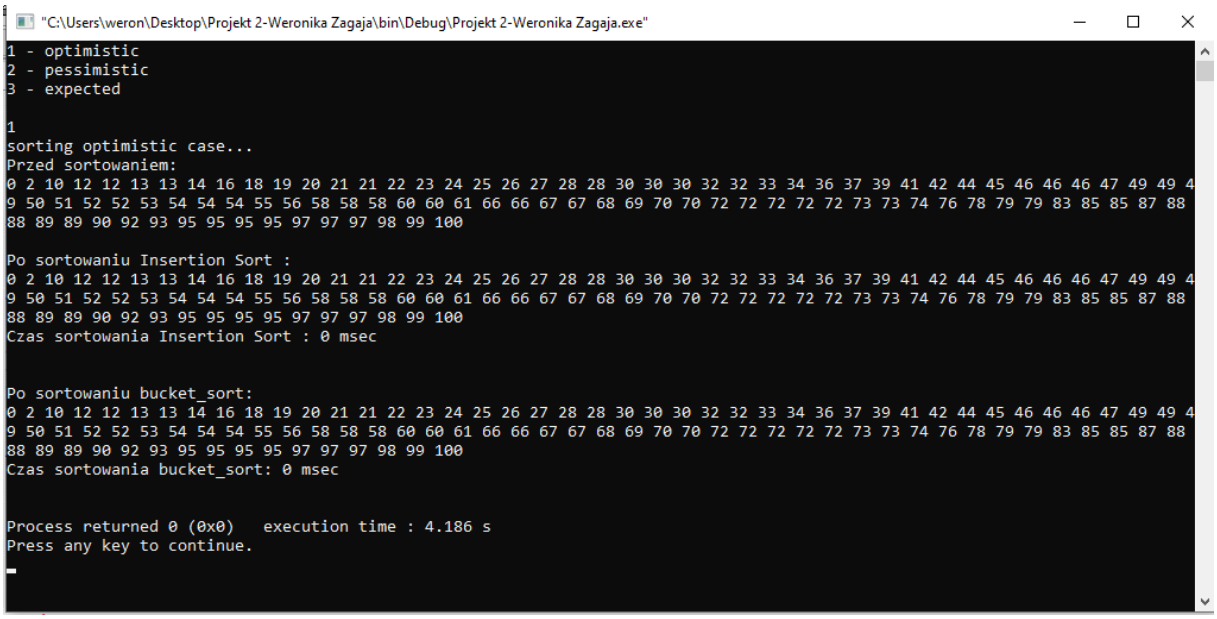
```cpp
// biblioteki
#include <iostream>
#include <fstream>
#include <time.h>
#include<cstdlib>
#include <iomanip>
#include <windows.h>
using namespace std;

//deklaracja czasu
clock_t start, stop;
double czas;

// deklaracja zmiennych
const int N=100;
const int WMIN = 0;
const int WMAX = N;
int d[N];
int d2[N];

void losowe() //implementacja funcji do losowania liczb
{
    fstream output("wylosowane liczby.txt", ios::out);

    // Najpierw wypelniamy tablice d[] liczbami pseudolosowymi
    srand(time(NULL));
    cout<<"Przed sortowaniem: "<<endl;
    output<<"Przed sortowaniem: "<<endl;
    for(int i=0; i<N; i++)
    {
        d[i] = WMIN + rand() % (WMAX - WMIN + 1);
        cout  << d[i]<<" ";
        output << d[i]<<" ";
    }
    cout << endl;

    //przepisanie tablicy d[]
    for(int i=0; i<N; i++)
    {
        d2[i] = WMIN + rand() % (WMAX - WMIN + 1);
    }
    for(int i=0; i<N; i++)
    {
        d2[i]=d[i];
    }

    output.close();
}
```

```
49
50  void  Insertion_Sort( int *D, int N) //implementacja funcji do sortowania przez wstawianie
51  {
52      //sortujemy przez wstawianie
53      int j, i, x;
54      {
55          for(j = N - 2; j >= 0; j--)
56          {
57              x = D[j];
58              i = j + 1;
59              while((i < N) && (x > D[i]))
60              {
61                  D[i - 1] = D[i];
62                  i++;
63              }
64              D[i - 1] = x;
65          }
66
67      }
68
69  }
70  void bucket_sort(int *D2, int WMIN, int WMAX) //implementacja funkcji do sortownia kubelkowego
71  {
72      // stworzenie wskaznika
73      int *lw;
74      lw = new int [WMAX - WMIN + 1];
75      int i,j;
76      {
77          // sortujemy kubelkowo
78
79          for(i = WMIN; i <= WMAX; i++) lw[i - WMIN] = 0;
80          for(i = 0; i < N; i++) lw[D2[i] - WMIN]++;
81          j = 0;
82          for(i = WMIN; i <= WMAX; i++) while(lw[i - WMIN]--) D2[j++] = i;
83      }
84      delete [] lw;
85      cout<<endl;
86  }
87
88  void Sort(string inputFile) //funkcja umozliwiajaca sortowanie dla ciagow
89                              //optymistycznych pesymistycznych i oczekiwanych
90  {
91      fstream output("wyniki sortowania.txt",ios::out);
92      if(inputFile != "expected")
93      {
94          int x;
95          fstream input(inputFile, ios::in);
96
97          cout << "Przed sortowaniem:" << endl;
98          output << "Przed sortowaniem:" << endl;
99          int i=0;
100         while (input >> x)      // zczytuje nosortowane liczby z pliku "optimistyczne.txt"
101         {
102             d[i] = x;
103             cout  << d[i]<<" ";
104             output << d[i]<<" ";
105             i++;
106         }
107         cout << endl;
108         output << endl;
109
110         input.close();
111     }
112     else
113     {
114         losowe();   //wywoluje funkcje "losowe"
115     }
116
```

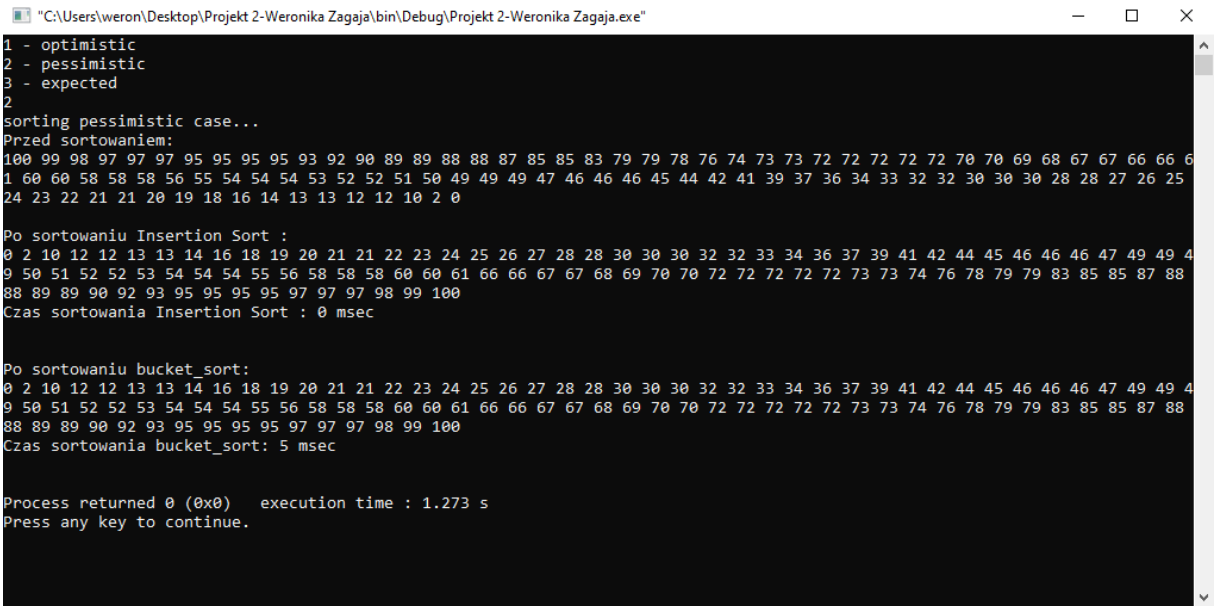
```
    //liczymy czas sortowania czasu wstawiania
    start = clock();
    Insertion_Sort(d, N);

    output<<endl;
    stop = clock();
    czas = (double)(stop-start)/(CLOCKS_PER_SEC/1000); // zliczamy czas w milisekundach

//wypisywanie posortowanej tablicy
    cout << "Po sortowaniu Insertion Sort :" << endl;
       for(int i = 0; i < N; i++){ cout << d[i]<<" ";}
    output << "Po sortowaniu Insertion Sort :" << endl;
       for(int i = 0; i < N; i++) output << setw(4) << d[i];
    cout<<endl;

    cout << "Czas sortowania Insertion Sort : " << czas << " msec" << endl;
    cout<<endl;

    //liczymy czas sortowania kubelkowego
    start=clock();
    bucket_sort(d2, WMIN, WMAX);
    stop = clock();
    czas = (double)(stop-start)/(CLOCKS_PER_SEC/1000);

    cout << "Po sortowaniu bucket_sort: "<<endl;
          for(int i = 0; i < N; i++) {cout <<  d[i]<<" ";}
    output<< "Po sortowaniu bucket_sort: ";
          for(int i = 0; i < N; i++) output << setw(4) << d[i];

    cout<<endl;
    cout << "Czas sortowania bucket_sort: " << czas << " msec" << endl << endl;
    output.close();
}

int main()
{
    int wybor = 0;
    cout << "1 - optimistic" << endl;
    cout << "2 - pessimistic" << endl;
    cout << "3 - expected" << endl;
    cin >> wybor;

    switch(wybor) //pozwala wybrac plik do sortowania
    {
    case 1:
        cout << "sorting optimistic case..." << endl;
        Sort("optymistyczne.txt");
        break;
    case 2:
        cout << "sorting pessimistic case..." << endl;
        Sort("pesymistyczne.txt");
        break;
    case 3:
        cout << "sorting random set of numbers..." << endl;
        Sort("expected");
        break;
    }
    return 0;
}
```

7. Działanie programu

Program miał dokonać sortowania liczb dwoma metodami i zliczenie czasu sortowań w celu porównania szybkości obu algorytmów. W związku z tym, kod został napisany tak, aby użytkownik miał możliwość wybrania danego ciągu liczb - optymistycznego, oczekiwanego oraz pesymistycznego. Zestaw danych - optymistyczny zawiera liczby w dużej mierze posortowane, (zob. Rysunek 5) pesymistyczny zaś obejmuje liczby w najbardziej niekorzystnym układzie, czyli w takim gdzie algorytmy muszą sortować najwięcej liczb (zob. Rysunek 6). Oba te ciągi odczytywane są z plików tekstowych. Ostatni zestaw danych- oczekiwany, posiada liczby wylosowane z funkcji

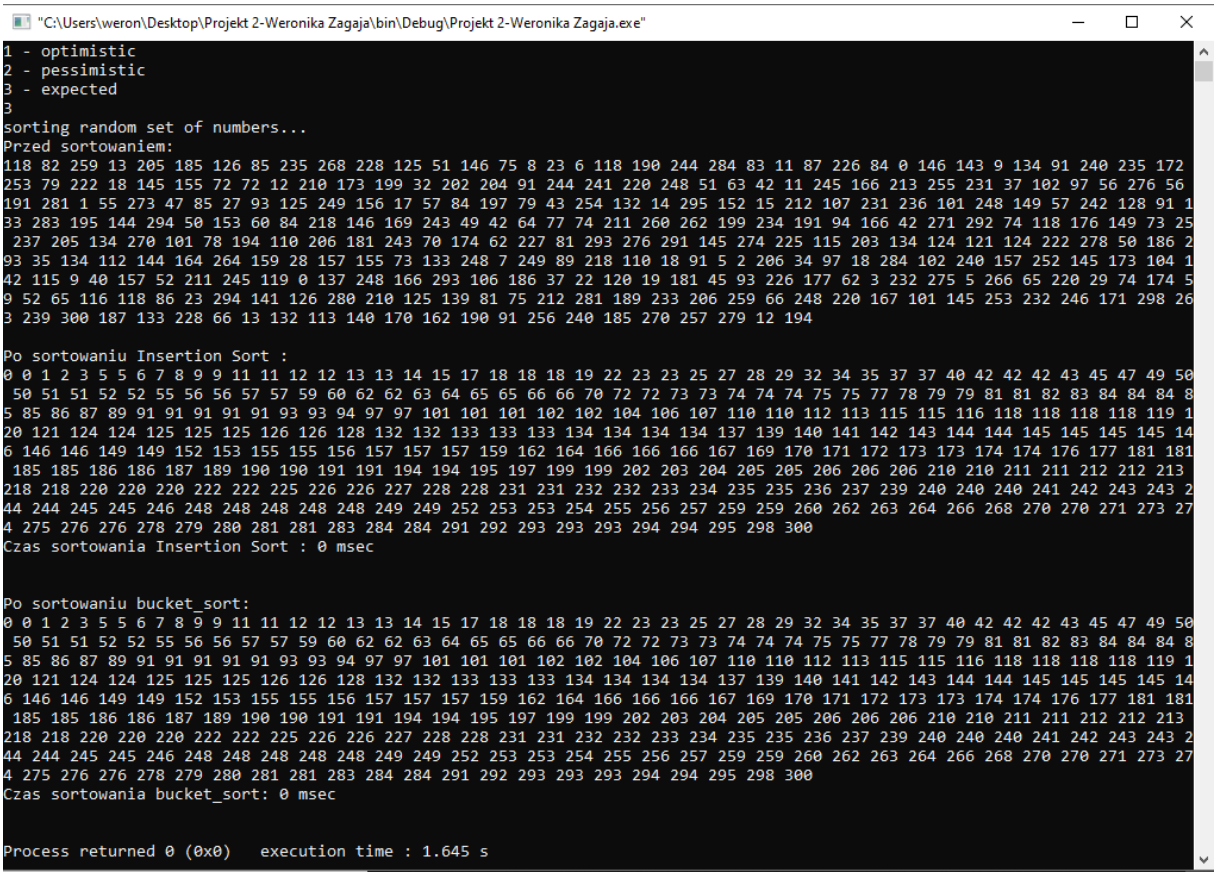


Rysunek 5 Sortowanie dla ciągu liczb optymistycznych



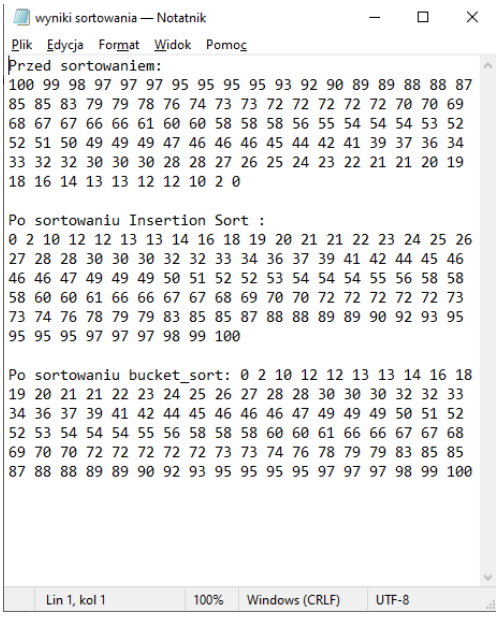
Rysunek 6 Sortowanie dla ciągu liczb pesymistycznych

W programie dodana jest funkcja generująca ”losowe” ciągi elementów o zadanej długości **N** (zob. Rysunek 7) i zapisującą je do pliku tekstowego „wylosowane liczby.txt”.



Rysunek 7 Sortowanie liczb losowych dla N=300

Dodatkowo program zawiera możliwość zapisu wyników do pliku testowego „wyniki sortowania.txt” (zob. Rysunek 8).



Rysunek 8 Zapis wyniku sortowań dla ciągu pesymistycznego w pliku tekstowym

8. Złożoność obliczeniowa

Sortowanie przez wstawianie

Złożoność algorytmu wynosi O(n^2). Istnieje jednak modyfikacja algorytmu, pozwalająca zmniejszyć liczbę porównań. Zamiast za każdym razem iterować po już posortowanym fragmencie (etap wstawiania elementu), można posłużyć się wyszukiwaniem binarnym. Pozwala to zmniejszyć liczbę porównań do O(*n*logn), nie zmienia się jednak złożoność algorytmu, ponieważ liczba przesunięć elementów to nadal O(n^2).

Sortowanie kubełkowe

Algorytm ma klasę czasowej złożoności obliczeniowej O(m + n), gdzie m oznacza ilość możliwych wartości, które mogą przyjmować elementy zbioru, a n to ilość sortowanych elementów. Jeśli m jest małe w porównaniu z n (sortujemy dużo elementów o małym zakresie wartości), to na czas sortowania będzie miała wpływ głównie ilość elementów n i klasa złożoności uprości się do postaci O(n). Dzieje się tak dlatego, iż przy równomiernym rozkładzie dużej ilości elementów o małym zakresie wartości liczniki będą równomiernie zapełnione. W sytuacji odwrotnej, gdy sortujemy mało elementów o dużym zakresie wartości klasa złożoności zredukuje się z kolei do O(m). W przypadku ogólnym pesymistyczna złożoność obliczeniowa tego algorytmu wynosi O(n^2).

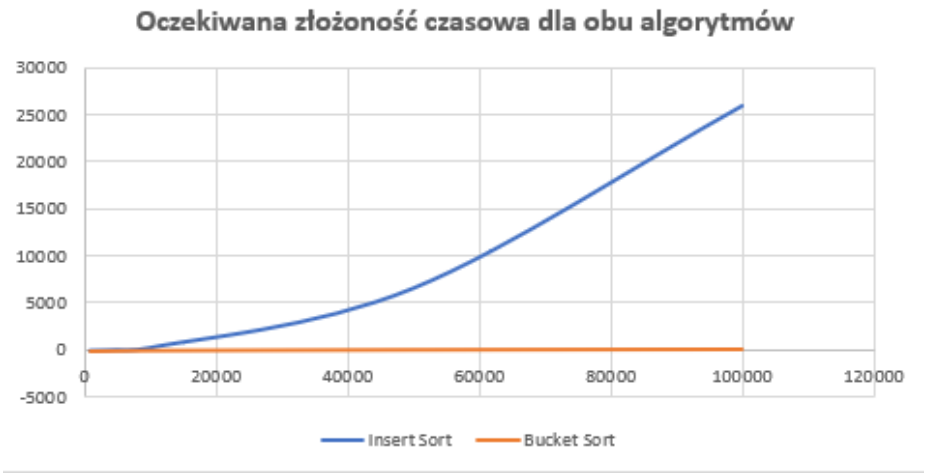
Wykresy porównujące

W celu oszacowania czasu sortowań dla zadanej tablicy N liczb losowych została wyliczona średnia z przeprowadzonych testów(zob. Rysunek 9) . Dodatkowo czas sortowań wyrażony jest w milisekundach, aby lepiej zauważyć różnice sortowań obu algorytmów. Ponadto trzeba zwrócić uwagę, że czas sortowań może się różnić na różnych procesorach komputerów.

Oczekiwana złożoność czasowa

N	1000		5000		10000		50000		100000	
rodzaj sortowania	Insert Sort	Bucket Sort	Insert Sort	Bucket Sort	Insert Sort	Bucket Sort	Insert Sort	Bucket Sort	Insert Sort	Bucket Sort
Proba 1	10	0	65	0	278	0	6920	10	25912	6
Proba 2	0	0	65	0	280	9	6510	0	26054	6
Proba 3	2	1	65	0	320	0	6494	8	26222	8
Proba 4	0	0	66	2	276	0	6549	3	26010	5
Proba 5	0	0	77	0	267	0	6561	0	25925	7
Proba 6	3	0	75	5	275	5	6512	8	25730	8
Średnia prób	2,5	0,16	68,83	1,16	282,6	2,33	6591	4,83	25975,5	6,6

Rysunek 9 Tabela czasów uzyskanych w sortowaniach



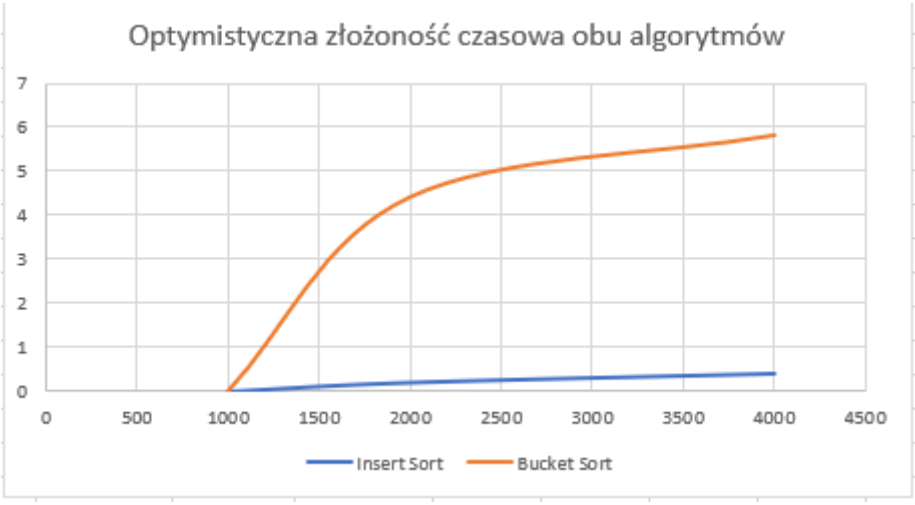
Rysunek 10 Wykres oczekiwanej złożoności czasowej dla obu algorytmów

Na wykresie można zauważyć, że wraz z rosnącymi próbkami danych **N** czas sortowania Insert Sort wzrasta parabolicznie, zaś dla sortowania Bucket Sort różnica czasów dla poszczególnych próbek danych jest bardzo mała, co przyczynia się do niewielkich zmian w wykresie.

Optymistyczna złożoność czasowa

N	1000		2000		4000	
Rodzaj sortowa	Insert Sort	Bucket Sort	Insert Sort	Bucket Sort	Insert Sort	Bucket Sort
próba 1	0	0	0	4	0	0
próba 2	0	0	0	2	1	11
próba 3	0	0	0	2	0	0
próba 4	0	0	1	4	0	8
próba 5	0	0	0	10	1	10
Średnia obliczeń	0	0	0,2	4,4	0,4	5,8

Rysunek 11 Tabela czasów uzyskanych w sortowaniach



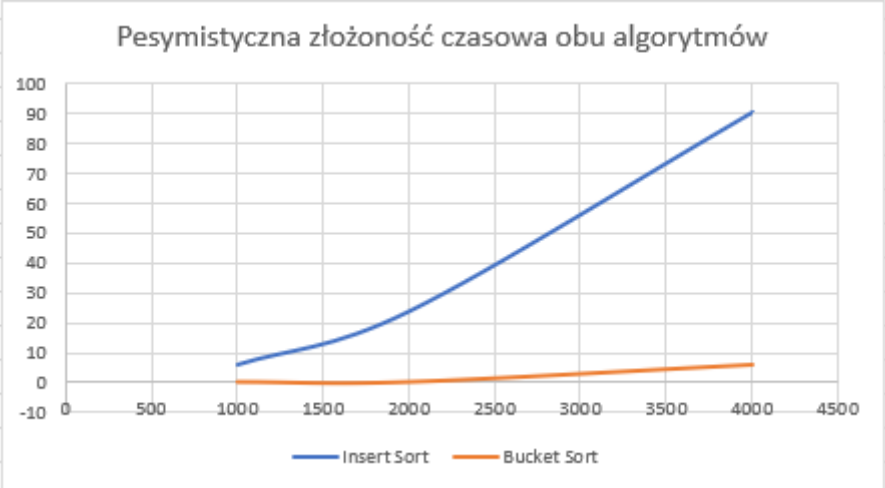
Rysunek 12 Wykres optymistycznej złożoności czasowej obu algorytmów

Na wykresie zauważyć można, że dla optymistycznych próbek danych czas sortowań obu algorytmów wzrasta logarytmicznie. Dodatkowo dostrzec można, że czas sortowania Insert Sort wzrasta szybciej niż czas sortowania Bucket Sort.

Pesymistyczna złożoność czasowa

N	1000		2000		4000	
Rodzaj sortowa	Insert Sort	Bucket Sort	Insert Sort	Bucket Sort	Insert Sort	Bucket Sort
próba 1	10	0	30	0	90	0
próba 2	0	0	20	0	90	10
próba 3	10	0	25	0	90	10
próba 4	11	0	25	0	90	10
próba 5	10	0	20	0	92	0
Średnia obliczeń	6,2	0	24	0	90,4	6

Rysunek 13 Tabela czasów uzyskanych w sortowaniach



Rysunek 14 Wykres pesymistycznej złożoności czasowej obu algorytmów

Na ostatnim wykresie widać, że dla pesymistycznych próbek danych czas wzrasta w obu algorytmach parabolicznie. Podobnie jak dla złożoności optymistycznej czas sortowania Insert Sort wzrasta szybciej niż czas Bucket Sort.

Porównując wszystkie wykresy sortowania przez wstawianie, zauważamy, że dla oczekiwanych i pesymistycznych próbek danych wzrastają one parabolicznie. Jednak zauważyć można, że czas złożoności pesymistycznej jest dłuższy niż dla złożoności oczekiwanej. Podobnie do sortowania przez wstawianie zachowuje się sortowanie kubełkowe. Jednak różnice czasowe pomiędzy złożonością oczekiwaną, a pesymistyczną są niewielkie.

9. Wnioski

Projekt udało się zrealizować i pokazać jego prawidłowe działanie. Obie metody sortowania zostały zaimplementowane w osobnych funkcjach. Dodatkowo zaimplementowana została funkcja generująca losowe liczby. Program posiada możliwość zapisu i odczytu danych z plików tekstowych. Ponadto, charakteryzuje się możliwością wybierania przez użytkownika próbek danych do sortowania, dzięki temu może zauważyć różnice złożoności obliczeniowej dla różnych zestawów zarówno dla sortowania przez wstawianie, tak i dla sortowania kubełkowego. W projekcie zostało wykonanych kilka prób w celu oszacowania średniego czasu sortowań. Ponadto czas został podany w milisekundach, aby zauważyć różnice czasowe w poszczególnych próbkach danych. Warto zwrócić uwagę, że czasy te mogą się różnić na odmiennych procesorach. Dodatkowo w kodzie zostały umieszczone stosowne komentarze, które pozwalają odczytać kod. W projekcie znajdują się również pseudokody oraz schematy blokowe, pomagające zrozumienie algorytmów. W projekcie zostały zawarte wszystkie zagadnienia, dotyczące prawidłowego zrealizowania zadania.