POLITECHNIKA
RZESZOWSKA
im. IGNACEGO ŁUKASIEWICZA

Karolina Magdoń

Inżynieria i analiza danych
I rok

Sprawozdanie z projektu sortowania metodą bąbelkową oraz metodą przez scalanie (C++)

Praca przygotowana na zajęcia
„Algorytmy i struktury danych”
w roku akademickim 2022/2023

Rzeszów, grudzień 2022

Spis treści

Spis rysunków

Spis tabel

Spis wykresów

1. Wstęp i opis zagadnień projektu

Moim zadaniem projektowym było utworzenie programu, który będzie służył do sortowania danego ciągu liczb metodą sortowania bąbelkową oraz metodą przez scalanie. Dodatkowymi warunkami było: przedstawienie podstaw teoretycznych zagadnienia, schematy blokowe obu metod sortowań (również pseudokod). Zobrazować rezultaty działania programu. Ukazać złożoność obliczeniową, czasową (typowa, optymistyczna, pesymistyczna).

2. Podstawy teoretyczne zagadnień

2.1 Sortowanie bąbelkowe

Sortowanie bąbelkowe jest jedną z podstawowych form sortowania w programowaniu. Algorytmy sortowania bąbelkowego przechodzą przez sekwencję danych (zazwyczaj liczb całkowitych) i przestawiają je w porządku rosnącym lub malejącym, po jednej liczbie na raz. W tym celu algorytm porównuje liczbę X z sąsiednią liczbą Y. Jeśli X jest większe niż Y, te dwie liczby są zamieniane i algorytm zaczyna się od nowa.

Proces ten powtarza się, aż cały zakres liczb zostanie posortowany w żądanej kolejności. Na przykład, jeśli próbujesz ułożyć [1, 2, 3, 4] w porządku rosnącym, algorytm sortowania bąbelkowego uruchomiłby się raz, zamieniając 3 na 2.

Jednak w innej macierzy Twoje liczby mogą wyglądać tak: [3, 1, 4, 2]. W tym przypadku algorytm działałby trzy razy, zamieniając 3 i 1 za pierwszym razem, następnie 4 i 2 za drugim razem, a na koniec 3 i 2.

Nazwa sortowania bąbelkowego pochodzi od faktu, że mniejsze lub większe elementy „przesuwają się” na górę zbioru danych. W poprzednim przykładzie [3, 1, 4, 2], 3 i 4 „bąbelkują” zestaw danych, aby znaleźć swoje właściwe pozycje.

Mocną stroną sortowania bąbelkowego jest jego prostota. Zajmuje tylko kilka linii kodu, jest łatwy do odczytania i można go podłączyć w dowolnym miejscu w programie. Jest to jednak wyjątkowo nieefektywne w przypadku większych zestawów liczb i powinno być odpowiednio używane.

2.2 Sortowanie przez scalanie

Sortowanie przez scalanie wykorzystuje metodę „dziel i zwyciężaj”. Jest to jeden z najpopularniejszych i najskuteczniejszych algorytmów sortowania. Dzieli podaną listę na dwie równe połowy, wywołuje siebie dla dwóch połówek, a następnie łączy dwie posortowane połowy. By móc korzystać z algorytmu musimy zdefiniować funkcję pomocniczą, która przeprowadzi scalanie.

Listy podrzędne są wielokrotnie dzielone na połowy, aż nie można dalej podzielić listy. Następnie łączymy parę list jednoelementowych w listy dwuelementowe, sortując je w procesie. Posortowane dwuelementowe pary są łączone w czteroelementowe listy itd., aż otrzymamy posortowaną listę.

3. Cechy programu

1. Możliwość odczytywania danych wejściowych z pliku tekstowego i zapisu posortowanego ciągu do pliku tekstowego z danymi wyjściowymi.
2. Na potrzeby testów zaimplementowanie funkcji generującej „losowe” ciągi elementów (o zadanej długości)
3. Założenie, że sortowanymi liczbami są liczby całkowite z przedziału [0,N].
4. Dwa algorytmy sortujące i scalająca funkcja pomocnicza.
5. 3 funkcje do testów: dla liczb całkowicie losowych, dla już posortowanych i dla posortowanych, ale w odwrotnym kierunku.
6. Mierzenie czasu działania obu algorytmów i zapis do pliku czasów działania dla serii prób z zwiększającą się ilością elementów tablic.
7. Kod opatrzony stosownymi komentarzami.

Do wykonania tego projektu stworzyłam kod w języku C++ zawierający:

- funkcję wczytującą z pliku serię liczb,
- funkcję generującą losowy zbiór liczb,
- algorytm sortowania bąbelkowego,
- pomocniczą funkcję scalającą,
- algorytm sortowania przez scalanie,
- 3 funkcje testów z mierzeniem i zapisywaniem czasów,
- segmenty zapisujące do plików otrzymane dane.

4. Złożoności obliczeniowe

4.1 Dla sortowania bąbelkowego

1) Optymistyczna

W najlepszym wypadku ma złożoność obliczeniową O(n). Staje się to wtedy, gdy tablica jest już posortowana, ponieważ algorytm musi wykonać tylko n-1 porównań żeby stwierdzić że tablica jest posortowana.

2) Pesymistyczna

W najgorszym wypadku złożoność obliczeniowa wynosi O(n^2). Gdy tablica jest posortowana, ale ułożona odwrotnie, zaczyna się na największym elemencie, a kończy na najmniejszym, wtedy w algorytmie każda liczby na swoim „bąbelku" będzie musiała przebyć pełną podróż przez tablice. Dla każdego elementu przechodzimy przez całą tabelę czyli n*n=n^2.

3) Typowa

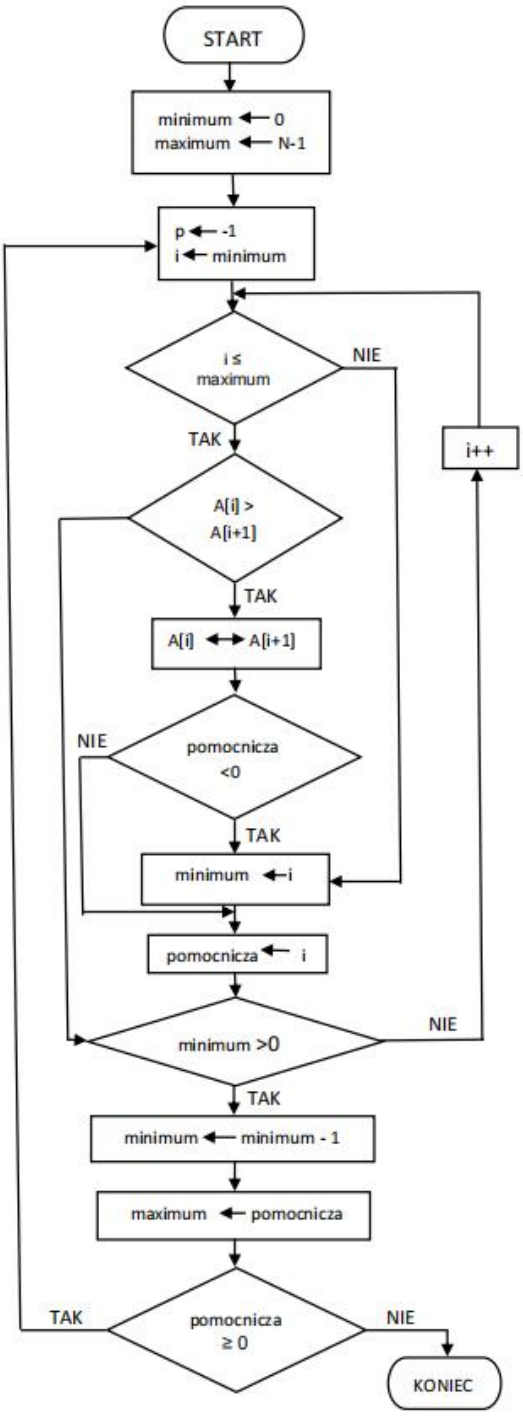
Biorąc pod uwagę, jak często dużo liczb będzie ułożone źle (losowy rozkład elementów), ogólna złożoność dalej pozostaje jako O(n^2).

4.2 Dla sortowania przez scalanie

Złożoność zawsze wynosi O(n*logn). Istnieje jednak przypadek pesymistyczny dla tego algorytmu, gdy lewa połowa tablicy tak zazębi się z prawą, że będziemy musieli dokonać porównania dla każdego scalania. Najlepszym przypadkiem jest znowu posortowana tablica.

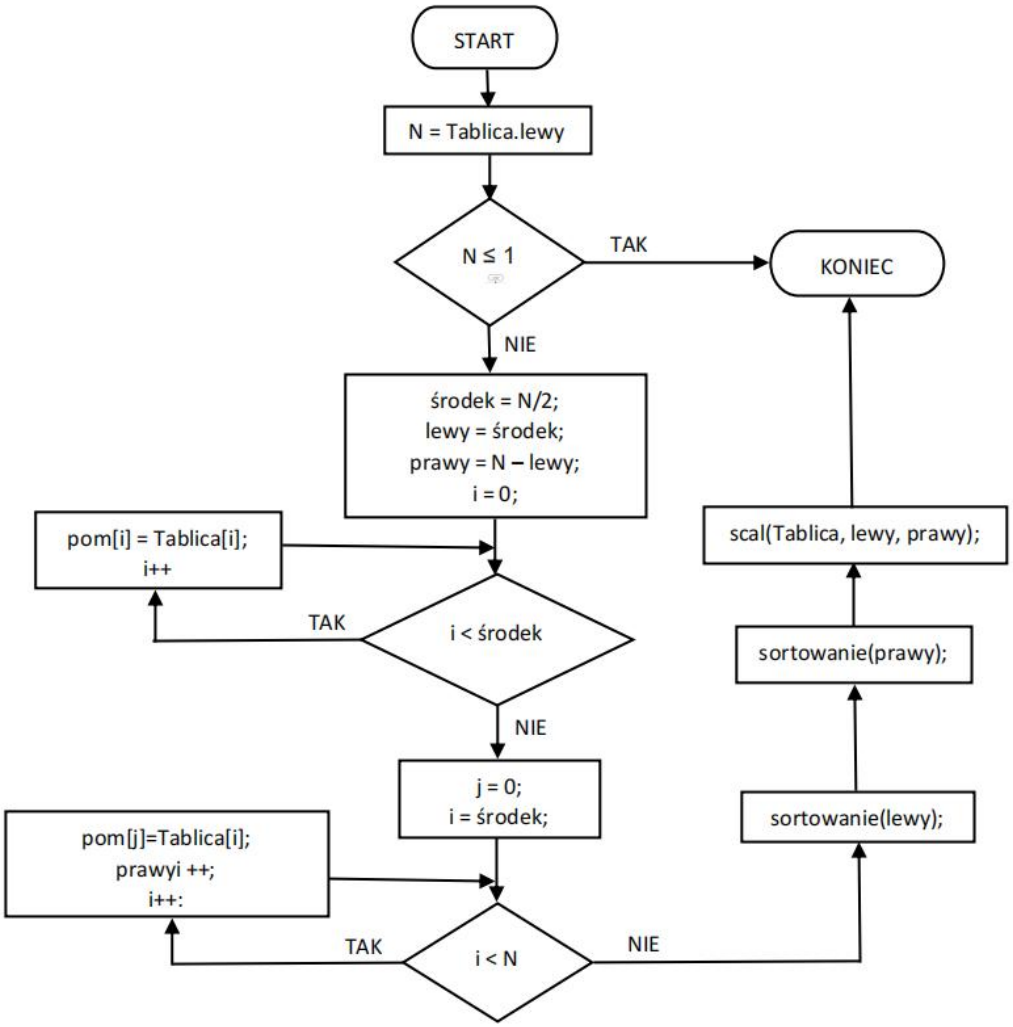
5. Schematy blokowe

5.1 Sortowanie bąbelkowe



Rys. 1 Schemat blokowy sortowania bąbelkowego

5.2 Sortowanie przez scalanie



Rys. 2 Schemat blokowy sortowania przez scalanie

6. Pseudokody

6.1 Sortowanie bąbelkowe

K01: Wczytaj Tabelę

K02: Patrzymy na każdy element po kolei i na element na prawo od niego, jeśli pierwszy z pary jest większy od drugiego, zamieniamy je miejscami

K03: Wracamy na początek tabeli i powtarzamy K02, aż nie przejdziemy całej tabeli nie zamieniając żadnych elementów

K03: Wypisz Tabelę.

6.2 Sortowanie przez scalanie

K01: Wczytaj Tabelę

K02: Tworzymy zmienne lewa i prawa które wyznaczają pierwszy i ostatni element tablicy

K03: Znajdujemy środek licząc (lewa+prawa)/2

K04: Jeśli lewa jest większa od prawa to przejdź do K06

K05: Rekursywnie wywołaj ten algorytm 2 razy, raz na tablicy od lewo do środek, i raz od środek+1 do prawo

K06: Scal połowiczne tabele

K07: Wypisz Tabelę.

7. Dane we/wy i zmienne pomocnicze

- **Dane wejściowe**: Tablica nieposortowana;
- **Dane wyjściowe**: Tablica posortowana;
- **Zmienne pomocnicze:**
    a. dla sortowania bąbelkowego:
        - minimum i maximum – wskazują zakres elementów, po których „idziemy”;
        - iterator- do przechodzenia po tablicy;
        - pomocnicza - numer pozycji zamiany elementów;

    b. dla sortowania przez scalanie:

        - lewa i prawa – granice elementów po których przechodzimy.

8. Rezultaty testów

Wykonałam poniżej testy dla różnych grup liczb (wygenerowanych losowo, posortowanych, posortowanych odwrotnie, przemiennych) i dla odpowiedniej ilości elementów tablicy. Tabele i wykresy zostały wykonane na bazie testów. Wyniki wypisane w milisekundach.

Test 1.

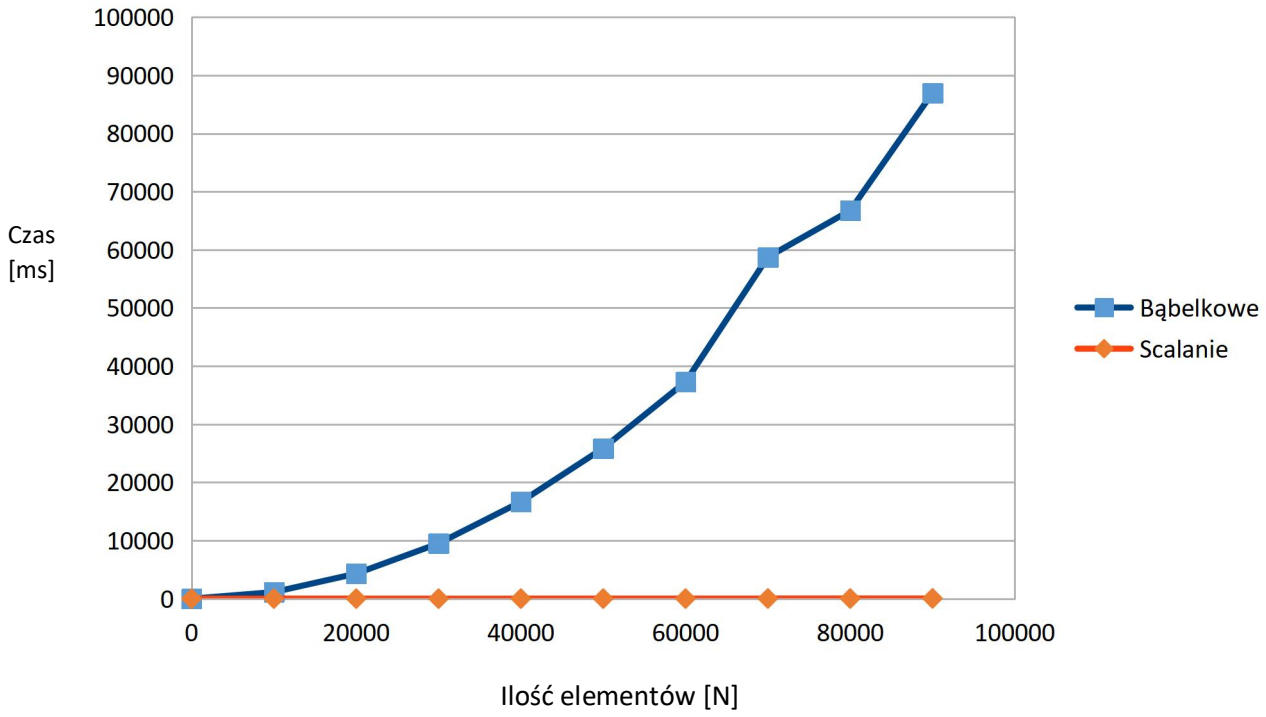
Na początek test dla liczb losowych. Ilość elementów zmienia się od 10 do 90010 o 10000:

Czasy dla liczb wygenerowanych losowo

N	Bąbelkowe	Scalanie
10	0	0
10010	1133	15
20010	4310	6
30010	9498	0
40010	16650	15
50010	25875	15
60010	37280	15
70010	58745	31
80010	66749	31
90010	86971	46

Tab. 1. Czasy dla liczb wygenerowanych losowo

Jak widać, scalanie deklasuje sortowanie bąbelkowe.



Wykres 1. Zależność czasu działania od liczby elementów dla liczb losowych

Oba czasy wykonania rosną, ale widać, że sortowanie metodą bąbelkową jest tak bardzo mniej wydajne, że scalanie wygląda jak płaska linia.

Test 2.

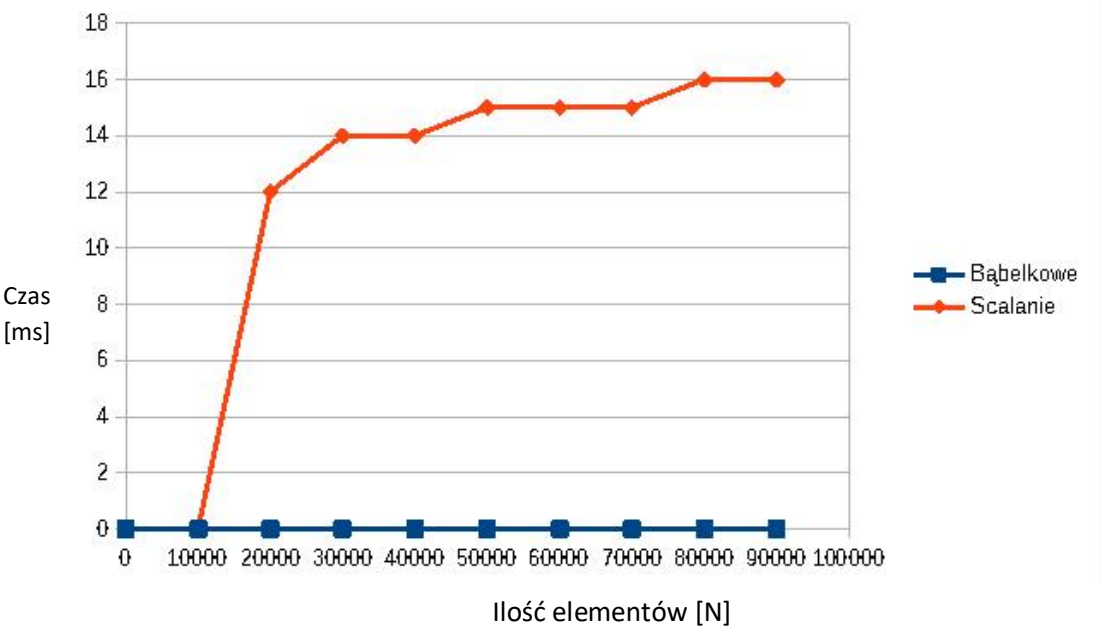
Dalej, chcąc trochę odwrócić to, który algorytm jest lepszy, weźmiemy najlepszy scenariusz dla bąbelkowego, czyli liczby już posortowane.

Czasy Posortowane 1

N	Bąbelkowe	Scalanie
10	0	0
10010	0	0
20010	0	12
30010	0	14
40010	0	14
50010	0	15
60010	0	15
70010	0	15
80010	0	16
90010	0	16

Tab. 2. Czasy dla liczb posortowanych 1

Jak widać, sortowanie bąbelkowe nawet nie rejestruje czasów sprawdzenia, czy tablica jest posortowana, natomiast scalanie wykonuje się jak zawsze, niezależnie od tego, jakie liczby będą mu dane.



Wykres 2. Zależność czasu działania od liczby elementów dla liczb posortowanych 1

Widzimy teraz odwrotny przypadek do poprzedniego wykresu, bo dominuje szybkością metoda „bąbelkowa”.

Test 3.

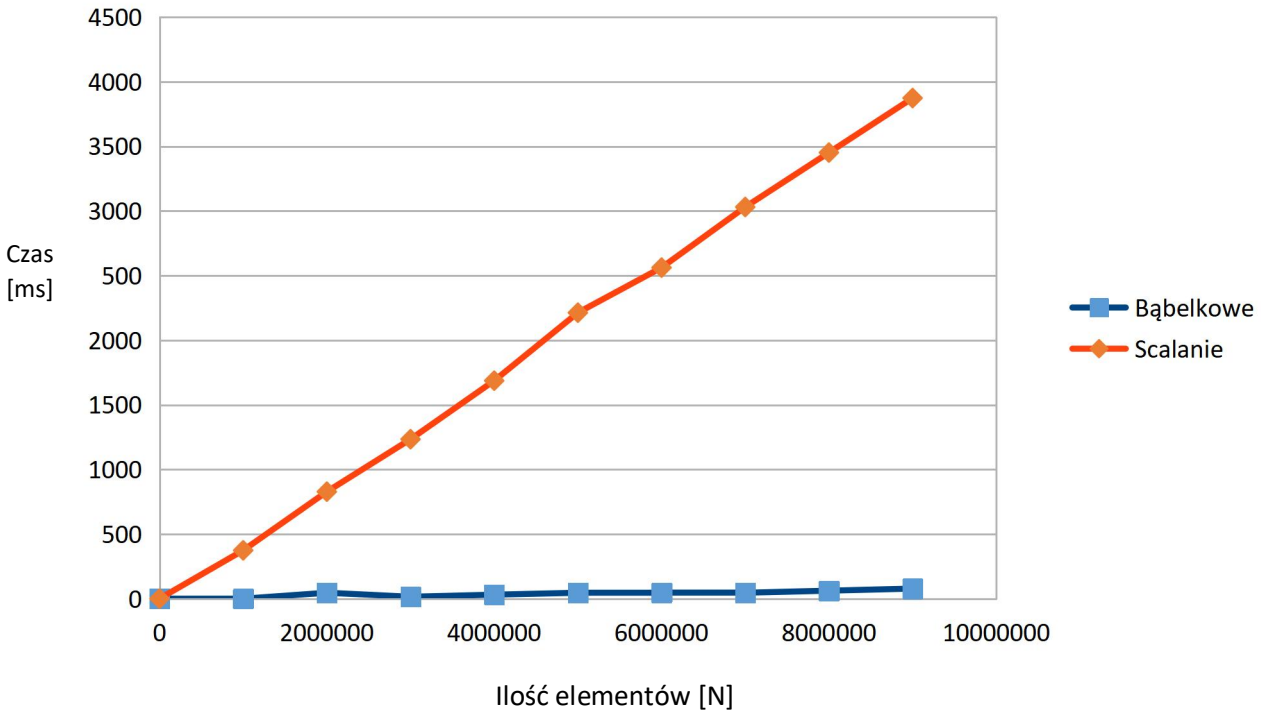
Natomiast nie chciałam zostawiać przy zerowych czasach więc na czas tej próby zwiększyłam trochę ilości elementów.

Czasy dla liczb posortowanych 2

N	Bąbelkowe	Scalanie
10	0	0
1000010	0	374
2000010	46	828
3000010	15	1234
4000010	31	1687
5000010	46	2213
6000010	47	2562
7000010	46	3031
8000010	62	3452
9000010	78	3874

Tab. 3. Czasy dla liczb posortowanych 2

Zależność jest dalej taka sama, ale widzimy że w pewnym momencie i sortowaniu bąbelkowemu zaczyna zajmować czas sprawdzić, czy tablica jest posortowana.



Wykres 3. Zależność czasu działania od liczby elementów dla liczb posortowanych 2

Dostajemy też ładniejszy wykres, ponieważ czasy nie są już tak małe i mają większą dokładność. Widać też, że przyrost wartości dla scalania jest wykresem nlogn, jak przewidywała teoria.

Test 4.

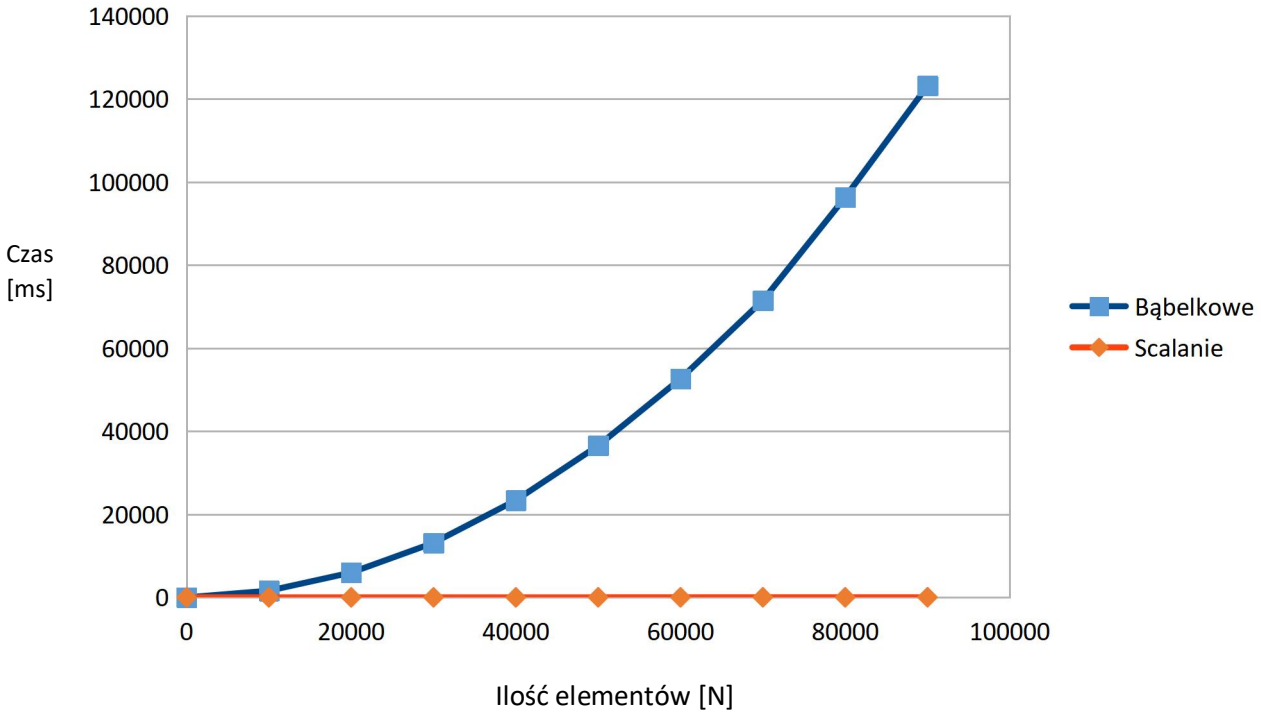
Kolejne próby wykonałam na danych z najgorszego dla sortowania bąbelkowego scenariusza, liczb posortowanych, ale ustawionych od największej do najmniejszej (malejąco).

Czasy dla liczb odwrotnie posortowanych

N	Bąbelkowe	Scalanie
10	0	0
10010	1602	0
20010	5968	0
30010	13126	0
40010	23378	0
50010	36536	15
60010	52635	15
70010	71493	20
80010	96326	15
90010	123168	31

Tab. 4. Czasy dla liczb odwrotnie posortowanych

Jak widać, czasy sortowania bąbelkowego stają się astronomicznie duże, natomiast scalanie stoi tak jak stało, zawsze z czasem działania tych samych rzędów.



Wykres 4. Zależność czasu działania od liczby elementów dla liczb odwrotnie posortowanych

Widać, że sortowanie metodą bąbelkową „pcha” się do góry szybciej. Tutaj też warto zaznaczyć, że przyrost „bąbelkowego” przypomina wykresy n^2, więc wszystko zgadza się z teorią.

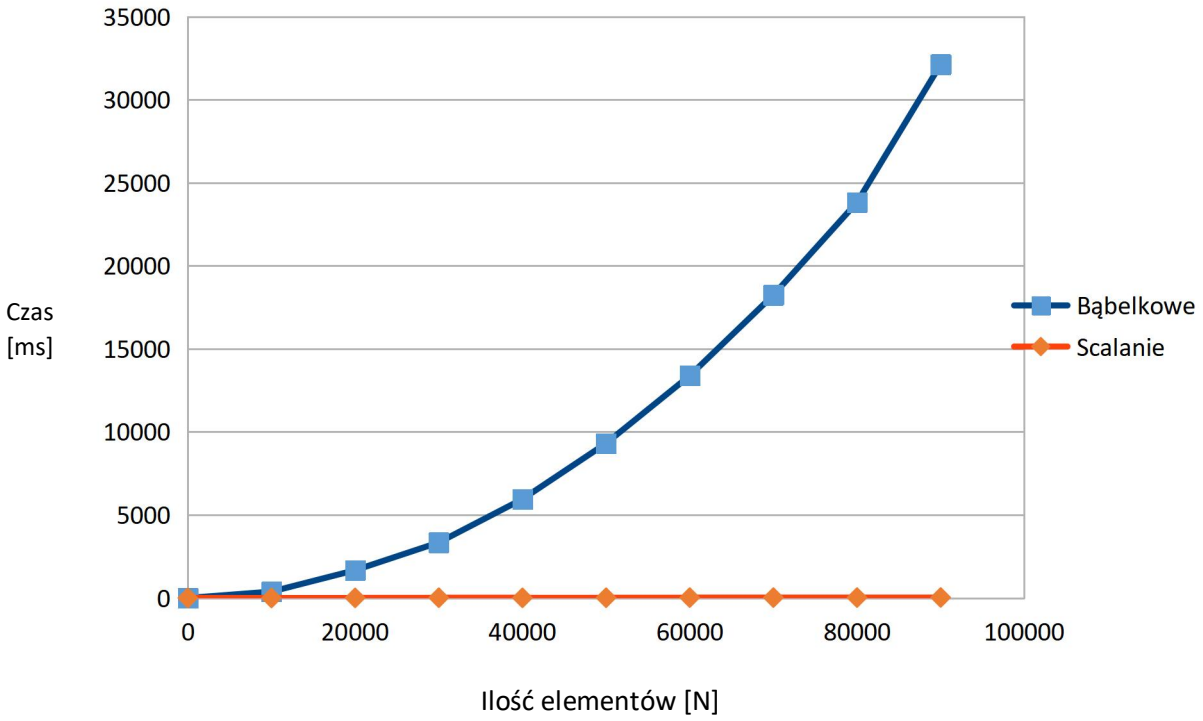
Test 5.

Ostatni test miał „pokonać” scalanie jego najgorszym przypadkiem, niestety okazał się bardziej zabójczy dla sortowania bąbelkowego, a kolejny wykres pokazałby nam to samo, co już dobrze znamy.

Czasy dla liczb przemiennych (ustawionych nieparzyste-parzyste)

N	Bąbelkowe	Scalanie
10	0	0
10010	383	0
20010	1672	7
30010	3343	8
40010	5951	12
50010	9311	15
60010	13383	20
70010	18239	23
80010	23822	24
90010	32125	31

Tab. 5. Czasy dla liczb przemiennych



Wykres 5. Zależność czasu działania od liczby elementów dla liczb ustawionych przemiennie (nieparzyste-parzyste)

9. Wnioskując, sortowanie bąbelkowe jest dużo wolniejsze.

9. Podsumowanie i wnioski

Założenia projektu udało się zrealizować. W ogólnym rozrachunku widać, że szybsze jest sortowanie przez scalanie. Ma ono zawsze czas działania tych samych rzędów, a sortowanie bąbelkowe szybko rośnie do ogromnych wartości. Sortowanie bąbelkowe można stosować, kiedy tablica jest posortowana lub prawie posortowana, bo wtedy będzie bardzo mało zamian, w innych przypadkach scalanie je deklasuje. Ciekawym faktem jest to, że przy sortowaniu przez scalanie trudno jest go spowolnić, nawet to, co miało być jego najgorszym przypadkiem obliczeniowym nie ma na niego znaczącego wpływu.

Podsumowując:

1. Program odczytuje dane liczbowe z pliku i zapisuje do tablicy.
2. Program zapisuje wynik do pliku.
3. Wykonano testy sprawdzające działanie algorytmu.
4. Program mierzy czas wykonania się algorytmu.
5. Kod opatrzono komentarzami.
6. Sporządzono schematy blokowe oraz pseudokody.

10. Appendix: kod programu

```
#include <iostream>
#include <ctime> //biblioteka potrzebna to losowania
#include <fstream> //biblioteka do wczytywania i zapisywania do txt
#include <chrono> //biblioteka do mierzenia czasu
using namespace std::chrono; //potrzebne do mierzenia czasu
using namespace std;



//Pierwsza funkcja sluzy do wczytania tabeli z pliku txt
int * WczytanieTablicy (){ //Funkcja zdefiniowana tak zeby nie przyjmowala zadnych argumentow i tak zeby zwracala wczytana tablicy

   fstream wczytywanie; //Tworzenie zmiennej, do ktorej zapisany bedzie plik, typu do tego przeznaczonego

   wczytywanie.open("Liczby.txt", ios::in); //Otwieranie wczesniej przygotowanego pliku tekstowego

   int i=0; //Deklaracja iteratora ktory bedzie sluzyl do zapisywania danych i rownoczesnie do zmierzenia dlugosci tablicy
   int * WczytanaTabela = new int[i]; //Tworzenie tablicy dynamicznej do ktorej zapisane beda liczby
   while(!wczytywanie.eof()){ //While wykonuje sie az skonczy sie plik
      wczytywanie >> WczytanaTabela[i]; //Po kolei wczytywane sa liczby na kolejne miejsca w tablicy
     i++; //Iterator jest zwiekszany przy kazdym wczytaniu liczby, co na raz liczy ile liczb jest wczytane i sluzy do wskazania kolejnego miejsca na ktore wpisywana jest liczba
   }
   wczytywanie.close(); //Zamykanie pliku

   //Wypisywanie do konsoli wczytanej tablicy
   cout<<"Wczytana Tablica = [ ";
   for (int j=0; j<i; j++){ //For idacy od 0 do i (ktore jest jest iloscia liczb)
      cout <<WczytanaTabela[j]<<" ";
   }
   cout<<"]"<<endl;
```

```
return WczytanaTabela; //Funkcja zwraca wczytana tablicy

}

//Druga funkcja generuje tablice liczb naturalnych od 0 do nadanego N

int * GenerowanieTablic (int N, int gran) { //Przyjmuje argument dlugosci tabicy do wygenerowania i granice rozrzutu wartości i zwraca tablice

srand((unsigned) time(0)); //Potrzebne do generowania liczb losowych

int * GenerowanaTablica = new int[N]; //Deklaracja tablicy dynamicznej na liczby

for (int j=0; j<N; j++){ //for dzialajacy N (przeslana do funkcji porzadana dlugosæ tablicy) razy

   GenerowanaTablica[j]=((rand()%gran)); }//Wykonywanie na rand modulo  np. 100 daje liczby od 0 do 100

fstream zapisywanie; //Tworzenie zmiennej, ktora bedzie przekazywaæ dane do pliku, typu do tego przeznaczonego

   zapisywanie.open("WygenerowanaTablica.txt", ios::out); //Otwieranie nowego pliku tekstowego do ktorego zapisywany bedzie wynik

   zapisywanie<<"WygenerowanaTablica = [ "; //Zamiast "cout" mamy zmienna "zapisywanie" wiec wszytko idzie do pliku txt

   for (int j=0; j<N; j++){

      zapisywanie<<GenerowanaTablica[j]<<" ";

   }zapisywanie<<"]"<<endl;

   zapisywanie.close(); //Zamkniecie pliku

return GenerowanaTablica; //Funkcja zwraca wygenerowana tablice

}

//Funkcja Sortowania Babelkowego

void SortowanieBabelkowe(int Tablica[], int N) //Przyjmuje tablice i jej dlugosc

{

   int minimum,maximum,i,pomocnicza; //Tworzone są  zmienne potrzebne nam do algorytmu, minimum i maximum mowia nam od którego do którego elementu mamy porownywac i pomocnicza ktora zapisuje ostatnia granice

   minimum = 0; maximum = N - 1; //Ustawinie pierwszego minimum na 0 i pierwszego maximum jako przedostatni element

  do //Wykonywana jest petla
```

```
  {
    pomocnicza = -1; //Przypisanie -1 do zmiennej pomocniczej
    for(i = minimum; i < maximum; i++) //Petla wykonuje sie od indeksu minimalnego elementu do maxymalnego
      if(Tablica[i] > Tablica[i+1]) //Sprawdzanie czy kolejny element jest mniejszy od poprzedniego
      {
        swap(Tablica[i], Tablica[i+1]); //Zamiana elementow miejscami
        if(pomocnicza < 0) minimum = i; //Gdy pomocnicza jest mniejsza od 0 minimum jest ustawiane na wartosc i
        pomocnicza = i; //Przypisanie wartosci i do pomocniczej
      }
    if(minimum) minimum--; //Jesli minimum nie jest rowne 0, to je zmniejszamy
    maximum = pomocnicza; //Maximum przyjmuje wartosc pomocniczej
  } while(pomocnicza >= 0); //Petla konczy sie jesli i jest rowne zero i po tym pomocnicza przejmie ta wartosc


}


int *pom; //Tablica pomocnicza, potrzebna przy scalaniu


//Scalenie posortowanych podtablic
void Scal(int Tablica[], int lewy, int srodek, int prawy) {//Przyjmuje Tablice, indeksy brzegowych elementow i srodek scalanej tablicy
            int i = lewy, j = srodek + 1; //Ustawianie i i j tak by przechodzic po odpowiednich elementach



  for(int i = lewy;i<=prawy; i++) //Kopiowanie calej tablicy do tablicy pomocniczej
    pom[i] = Tablica[i];


  for(int k=lewy;k<=prawy;k++) //Przechodzenie po tablicy od lewej granicy do prawej
  if(i<=srodek) //Jesli indeks idacy od lewej nie dotar jeszcze do srodka
    if(j <= prawy) //Jesli indeks idacy od srodka nie dotar jeszcze do prawej
        if(pom[j]<pom[i]) //Jesli element o indeksie j jest mniejszy od elementu o indeksie i (bierzemy z tabeli pomocniczej zeby nie byly zamienione)
            Tablica[k] = pom[j++];
        else                    //Do tablicy wstawiamy liczbe po prawej od mniejszej liczby
            Tablica[k] = pom[i++];
```

```
   else
      Tablica[k] = pom[i++]; //Dla wiekszego od prawy wstwaiamy element o jeden w prawo
 else
    Tablica[k] = pom[j++]; //Dla wiekszego od srodek wstwaiamy element o jeden w prawo
}

//Funkcja Sortowania Przez Scalanie
void SortowaniePrzezScalanie(int Tablica[],int lewy, int prawy) //Przyjmuje Tablice, lewy i prawy koniec
{


        if(prawy<=lewy) return; //Gdy jest jeden element, to jest on już posortowany


        int srodek = (prawy+lewy)/2; //Wyznaczanie srodka tablicy

        SortowaniePrzezScalanie(Tablica, lewy, srodek); //Rekursywnie wywołujemy funkcje na dwoch polowkach tablicy
        SortowaniePrzezScalanie(Tablica, srodek+1, prawy);

        Scal(Tablica, lewy, srodek, prawy); //Scalamy tablice spowrotem
}

//Funkcja do testowania
void TestyLosowe (int Tablica[]) //Przyjmuje pusta zmienna dla tablic
{

   fstream zapisywanie; //Tworzenie zmiennej, ktora bedzie przekazywaæ dane do pliku, typu do tego przeznaczonego
   zapisywanie.open("CzasyLosowe.txt", ios::out); //Otwieranie nowego pliku tekstowego do ktorego zapisywane beda wyniki

   for (int ile=10; ile<100000; ile+=10000){ //For ktory wykonuje sie 10 razy, wartosc iteratora jest na raz tez dlugoscia tablicy ktora kazemy wygenerowaæ programowi. Jest ona tak duza bo co dopiero taka dlugosæ daje jakikolwiek czas dzialania
```

```
    Tablica = GenerowanieTablic (ile,100); //Wywolanie generowania tablic o dlugosci zgodnej z iteratorem for'a

    auto startb = high_resolution_clock::now(); //Start mierzenia czasu

    SortowanieBabelkowe(Tablica,ile); //Wywolanie glownej funkcji z wygnenrowana tablica

    auto koniecb = high_resolution_clock::now(); //Koniec mierzenia czasu

    auto BabelkoweCzas = duration_cast<milliseconds>(koniecb - startb); //Zapisujemy czas w typie double w formacie milisekund


    Tablica = GenerowanieTablic (ile,100); //Wywolanie generowania tablic o dlugosci zgodnej z iteratorem for'a

    pom = new int[ile]; //Dostosowanie tablicy pomocniczej do dlugosci wygenerowanej

    auto starts = high_resolution_clock::now(); //Start mierzenia czasu

    SortowaniePrzezScalanie(Tablica,0,ile-1); //Wywolanie glownej funkcji z wygnenrowana tablica

    auto koniecs = high_resolution_clock::now(); //Koniec mierzenia czasu

    auto ScalanieCzas = duration_cast<milliseconds>(koniecs - starts); //Zapisujemy czas w typie double w formacie milisekund


    cout<<"Proba dla "<<ile<<" elementow dla sortowania babelkowego. Czas wykonania: "<<BabelkoweCzas.count()<<"ms"<<endl; //Wyswietlanie wynikow

    cout<<"Proba dla "<<ile<<" elementow dla sortowania przez scalanie. Czas wykonania: "<<ScalanieCzas.count()<<"ms"<<endl;


    zapisywanie<<ile<<","<<BabelkoweCzas.count()<<","<<ScalanieCzas.count()<<endl; //zapisywanie czasow do pliku

  }

  zapisywanie.close(); //zamkniecie pliku


}


//Funkcja do testowania

void TestyPosortowane (int Tablica[]) //Przyjmuje pusta zmienna dla tablic

{


  fstream zapisywanie; //Tworzenie zmiennej, ktora bedzie przekazywaæ dane do pliku, typu do tego przeznaczonego

  zapisywanie.open("CzasyPosortowane.txt", ios::out); //Otwieranie nowego pliku tekstowego do ktorego zapisywane beda wyniki
```

```
   for (int ile=10; ile<10000000; ile+=1000000){ //For ktory wykonuje sie 10 razy, wartosc iteratora jest na raz tez dlugoscia tablicy ktora kazemy wygenerowaæ programowi. Jest ona tak duza bo co dopiero taka dlugosæ daje jakikolwiek czas dzialania


      Tablica = new int[ile]; //Tworzenie tablicy zapisując liczby od 0 do ile

      for (int i=0; i<ile; i++){

         Tablica[i]=i;

      }


      auto startb = high_resolution_clock::now(); //Start mierzenia czasu

      SortowanieBabelkowe(Tablica,ile); //Wywolanie glownej funkcji z wygnenrowana tablica

      auto koniecb = high_resolution_clock::now(); //Koniec mierzenia czasu

      auto BabelkoweCzas = duration_cast<milliseconds>(koniecb - startb); //Zapisujemy czas w typie double w formacie milisekund


      pom = new int[ile]; //Dostosowanie tablicy pomocniczej do dlugosci wygenerowanej

      auto starts = high_resolution_clock::now(); //Start mierzenia czasu

      SortowaniePrzezScalanie(Tablica,0,ile-1); //Wywolanie glownej funkcji z wygnenrowana tablica

      auto koniecs = high_resolution_clock::now(); //Koniec mierzenia czasu

      auto ScalanieCzas = duration_cast<milliseconds>(koniecs - starts); //Zapisujemy czas w typie double w formacie milisekund


      cout<<"Proba dla "<<ile<<" elementow dla sortowania babelkowego. Czas wykonania: "<<BabelkoweCzas.count()<<"ms"<<endl; //Wyswietlanie wynikow

      cout<<"Proba dla "<<ile<<" elementow dla sortowania przez scalanie. Czas wykonania: "<<ScalanieCzas.count()<<"ms"<<endl;


      zapisywanie<<ile<<","<<BabelkoweCzas.count()<<","<<ScalanieCzas.count()<<endl; //zapisywanie czasow do pliku

   }

   zapisywanie.close(); //zamkniecie pliku


}


//Funkcja do testowania

void TestyOdwrotniePosortowane (int Tablica[]) //Przyjmuje pusta zmienna dla tablic

{
```

fstream zapisywanie; //Tworzenie zmiennej, ktora bedzie przekazywaæ dane do pliku, typu do tego przeznaczonego

zapisywanie.open("CzasyOdwrotniePosortowane.txt", ios::out); //Otwieranie nowego pliku tekstowego do ktorego zapisywane beda wyniki

for (int ile=10; ile<100000; ile+=10000){ //For ktory wykonuje sie 10 razy, wartosc iteratora jest na raz tez dlugoscia tablicy ktora kazemy wygenerowaæ programowi. Jest ona tak duza bo co dopiero taka dlugosæ daje jakikolwiek czas dzialania

Tablica = new int[ile+1]; //Tworzenie tablicy zapisując liczby od ile do 0

for (int i=ile; i>-1; i--){

Tablica[ile-i]=i;

}

auto startb = high_resolution_clock::now(); //Start mierzenia czasu

SortowanieBabelkowe(Tablica,ile); //Wywolanie glownej funkcji z wygnenrowana tablica

auto koniecb = high_resolution_clock::now(); //Koniec mierzenia czasu

auto BabelkoweCzas = duration_cast<milliseconds>(koniecb - startb); //Zapisujemy czas w typie double w formacie milisekund

pom = new int[ile]; //Dostosowanie tablicy pomocniczej do dlugosci wygenerowanej

auto starts = high_resolution_clock::now(); //Start mierzenia czasu

SortowaniePrzezScalanie(Tablica,0,ile-1); //Wywolanie glownej funkcji z wygnenrowana tablica

auto koniecs = high_resolution_clock::now(); //Koniec mierzenia czasu

auto ScalanieCzas = duration_cast<milliseconds>(koniecs - starts); //Zapisujemy czas w typie double w formacie milisekund

cout<<"Proba dla "<<ile<<" elementow dla sortowania babelkowego. Czas wykonania: "<<BabelkoweCzas.count()<<"ms"<<endl; //Wyswietlanie wynikow

cout<<"Proba dla "<<ile<<" elementow dla sortowania przez scalanie. Czas wykonania: "<<ScalanieCzas.count()<<"ms"<<endl;

zapisywanie<<ile<<","<<BabelkoweCzas.count()<<","<<ScalanieCzas.count()<<endl; //zapisywanie czasow do pliku

}

zapisywanie.close(); //zamkniecie pliku

```
}

//Funkcja do testowania
void TestyPrzemienne (int Tablica[]) //Przyjmuje pusta zmienna dla tablic
{

   fstream zapisywanie; //Tworzenie zmiennej, ktora bedzie przekazywaæ dane do pliku, typu do tego przeznaczonego
   zapisywanie.open("CzasyPrzemienne.txt", ios::out); //Otwieranie nowego pliku tekstowego do ktorego zapisywane beda wyniki


   for (int ile=10; ile<100000; ile+=10000){ //For ktory wykonuje sie 10 razy, wartosc iteratora jest na raz tez dlugoscia tablicy ktora kazemy wygenerowaæ programowi. Jest ona tak duza bo co dopiero taka dlugosæ daje jakikolwiek czas dzialania

      Tablica = new int[ile]; //Tworzenie tablicy zapisując najpierw liczby nieparzyste a potem parzyste
      for (int i=0; i<ile; i+=2){
         Tablica[(i/2)+1]=i+1;
      }
      for (int i=0; i<ile; i+=2){
         Tablica[i/2+ile/2]=i;
      }

      auto startb = high_resolution_clock::now(); //Start mierzenia czasu
      SortowanieBabelkowe(Tablica,ile); //Wywolanie glownej funkcji z wygnenrowana tablica
      auto koniecb = high_resolution_clock::now(); //Koniec mierzenia czasu
      auto BabelkoweCzas = duration_cast<milliseconds>(koniecb - startb); //Zapisujemy czas w typie double w formacie milisekund

      Tablica = new int[ile]; //Tworzenie wykonujemy 2 razy by tablica nie była już posortowana

      for (int i=0; i<ile; i+=2){
         Tablica[(i/2)+1]=i+1;
      }
```

```
    for (int i=0; i<ile; i+=2){
        Tablica[i/2+ile/2]=i;
    }

    pom = new int[ile]; //Dostosowanie tablicy pomocniczej do dlugosci wygenerowanej
    auto starts = high_resolution_clock::now(); //Start mierzenia czasu
    SortowaniePrzezScalanie(Tablica,0,ile-1); //Wywolanie glownej funkcji z wygnenrowana tablica
    auto koniecs = high_resolution_clock::now(); //Koniec mierzenia czasu
    auto ScalanieCzas = duration_cast<milliseconds>(koniecs - starts); //Zapisujemy czas w typie double w formacie milisekund

    cout<<"Proba dla "<<ile<<" elementow dla sortowania babelkowego. Czas wykonania: "<<BabelkoweCzas.count()<<"ms"<<endl; //Wyswietlanie wynikow
    cout<<"Proba dla "<<ile<<" elementow dla sortowania przez scalanie. Czas wykonania: "<<ScalanieCzas.count()<<"ms"<<endl;

    zapisywanie<<ile<<","<<BabelkoweCzas.count()<<","<<ScalanieCzas.count()<<endl; //zapisywanie czasow do pliku
  }
  zapisywanie.close(); //zamkniecie pliku
}

//Funkcja main, tu odpalane sa inne fukncje i ich obsluga
int main()
{
  int * Tablica; //Tworzenie tablicy dynamicznej do ktorej beda przypisywane wyniki

  cout<<"1. Przyklad wczytywania z pliku:"<<endl<<endl;

  Tablica=WczytanieTablicy();

  SortowanieBabelkowe(Tablica,sizeof(Tablica));

  cout<<"Posortowana Tablica = [ ";
  for (int j=0; j<sizeof(Tablica); j++){
```

```
    cout <<Tablica[j]<<" ";
  }cout<<"]"<<endl<<endl;

  fstream zapisywanie; //Tworzenie zmiennej, ktora bedzie przekazywaæ dane do pliku, typu do tego przeznaczonego
  zapisywanie.open("Wynik.txt", ios::out); //Otwieranie nowego pliku tekstowego do ktorego zapisywany bedzie wynik
  zapisywanie<<"Tablica po sortowaniu = [ "; //Zamiast "cout" mamy zmienna "zapisywanie" wiec wszytko idzie do pliku txt
  for (int j=0; j<sizeof(Tablica); j++){
    zapisywanie<<Tablica[j]<<" ";
  }
  zapisywanie<<"]"<<endl;
  zapisywanie.close(); //Zamkniecie pliku

  cout<<"2. Pomiary czasow dzialania:"<<endl<<endl;

  cout<<endl<<"2.1 Pomiar dla tablicy losowej:"<<endl<<endl;

  TestyLosowe(Tablica); //Wywolywanie testow

  cout<<endl<<"2.2 Pomiar dla tablicy posortowanej:"<<endl<<endl;

  TestyPosortowane(Tablica);

  cout<<endl<<"2.3 Pomiar dla tablicy posortowanej ale zaczynajacej sie od najwiekzej liczby:"<<endl<<endl;

  TestyOdwrotniePosortowane(Tablica);

 cout<<endl<<"2.4 Pomiar dla tablicy z wpisanymi najpierw liczbami nieparzystymi a potem parzystymi:"<<endl<<endl;

  TestyPrzemienne(Tablica);

  return 0;
}
```