

**WYŻSZA SZKOŁA HANDLOWA
W RADOMIU**



**RADOM
ACADEMY OF ECONOMICS**

Wydział Studiów Strategicznych i Technicznych

Kierunek: Informatyka, rok II, semestr III (2021/2022)

LABORATORIUM ALGORYTMY I ZŁOŻONOŚĆ OBLICZENIOWA

Prowadzący: doktor habilitowany Filip Rudziński

Zespół laboratoryjny:

Magdalena Szafrńska, nr albumu: 18345

Spis treści

Wstęp	2
Użyte technologie	2
Zastosowana konwencja w pisaniu kodu projektu	2
Pliki źródłowe	2
Cel laboratoriów	3
Laboratorium nr 1	4
Wstęp	4
Cel laboratorium nr 1	5
Wykonanie ćwiczenia cz. I: implementacja algorytmów	5
Create, RandomFill, Print, Delete	5
Adding, Subtraction	6
Copy	7
Trans	8
Multiplication	9
Wykonanie ćwiczenia cz. II: pomiar czasów wykonywania	10
Funkcja GetTickCount()	10
Funkcja clock()	12
Laboratoria nr 2	14
Wstęp	14
Cel laboratorium nr 2	14
Wykonanie ćwiczenia cz. I: czasy wykonywania	14
Uruchomienie bazowej aplikacji	14
Pomiar czasów wykonywania	20
Wykonanie ćwiczenia cz. II: charakterystyki złożoności	21
Generowanie wyników aplikacji	21
Tworzenie wykresu dla sortowania bąbelkowego, przez wybór i przez wstawianie	22
Tworzenie wykresu dla sortowania przez scalanie, szybkiego i przez zliczanie	26
Laboratoria nr 3	30
Wstęp	30
Cel laboratorium nr 3	31
Wykonanie ćwiczenia cz. I: lista jednokierunkowa	31
Implementacja listy jednokierunkowej	31
Wykonanie ćwiczenia cz. II: lista dwukierunkowa	39
Algorytm sumujący elementy na liście - wersja podstawowa	39
Algorytm sumujący elementy na liście - wersja ulepszona	43
Wnioski	54
Dot. laboratorium nr 1	54
Dot. laboratorium nr 2	54
Dot. laboratorium nr 3	54

Wstęp

Użyte technologie

- Na wszystkich laboratoriach korzystałam z następujących narzędzi:
 - język programowania C++
 - Git - system kontroli wersji
 - Github - zdalne repozytorium
 - Git Bash - konsola Gita
 - Google Documents (Dokumenty i Arkusze Google)
- Dodatkowo specyficznie dla wybranych laboratoriów:
 - Laboratorium nr 1 i 3:
 - Microsoft Visual Studio Community 2019 (wersja 16.11.1)
 - Laboratorium 2:
 - środowisko IDE Code::Blocks (wersja Release 20.03 rev 11983)
 - kompilator MinGW z kompilatorem gcc 8.1.0 Windows/unicode w wersji 64-bitowej)

Zastosowana konwencja w pisaniu kodu projektu

Według poznanych w ubiegłym semestrze dobrych praktych programowania podzieliłam w kodzie funkcje na ich prototypy podane w deklaracji (przed funkcją main()) oraz definicje (za funkcją main()).

Podane w deklaracji nagłówki funkcji zawierające informacje o nazwie, typie zwracanym oraz typie i liczbie parametrów mają na celu ogólny pogląd co dana funkcja będzie wykonywać. Jest to wszystko, czego potrzebuje kompilator, aby sprawdzić wstępną, formalną prawidłowość ich użycia.

Pliki źródłowe

Każde z trzech laboratoriów znajduje się w osobnym repozytorium na GitHub. Linki poniżej:

- [Laboratorium nr 1](#)
- [Laboratorium nr 2](#)
- Laboratorium nr 3
 - [część I](#) - lista jednokierunkowa
 - [część II](#) - lista dwukierunkowa

Link do niniejszego sprawozdania dostępny jest [tutaj](#).

Cel laboratoriów

Celem laboratoriów było zapoznanie się z podstawowymi algorytmami struktur danych oraz analiza złożoności obliczeniowej i czasu ich wykonywania. Wykresy złożoności dogłębnie pokazały zależność pomiędzy rodzajami wybranych do implementacji algorytmów wskazując na ich słabe i mocne strony.

Laboratorium nr 1

Wstęp

Macierz (ang. matrix) jest tablicą dwuwymiarową, która składa się z n wierszy oraz m kolumn. Wiersze numerowane są kolejno od 0 do $n-1$, a kolumny od 0 do $m-1$. Fakt posiadania przez macierz n wierszy oraz m kolumn zapisujemy w sposób następujący: $A_{n \times m}$

Elementy macierzy posiadają dwa indeksy - pierwszy indeks określa numer wiersza, w którym dany element występuje, a drugi indeks określa numer kolumny. Na przykład macierz $A_{3 \times 4}$ posiada trzy wiersze i 4 kolumny. Elementy tej macierzy są następujące:

$$A_{3 \times 4} = \begin{bmatrix} a_{0,0} & ,a_{0,1} & ,a_{0,2} & ,a_{0,3} \\ a_{1,0} & ,a_{1,1} & ,a_{1,2} & ,a_{1,3} \\ a_{2,0} & ,a_{2,1} & ,a_{2,2} & ,a_{2,3} \end{bmatrix}$$

Element $a_{2,1}$ leży w wierszu o numerze 2 (wiersz trzeci z uwagi na stratę numeracji od 0!) oraz w kolumnie o numerze 1 (kolumna druga!).

W języku C++ macierz możemy utworzyć następująco:

```
double A[n][m]; // macierz n wierszy na m kolumn utworzona statycznie
double * B = new double[n * m]; // macierz n wierszy na m kolumn utworzona dynamicznie
```

Pierwszy sposób jest przydatny, gdy znamy liczbę wierszy i kolumn w trakcie tworzenia tekstu programu. Do elementów tak utworzonej macierzy odwołujemy się bezpośrednio za pomocą indeksów:

$A[i][j]$ - element w i -tym wierszu i j -tej kolumnie

Drugi sposób pozwala tworzyć dynamicznie dowolne macierze. Dostęp do elementu wykonywany jest następująco:

$B[m * i + j]$ - element w i -tym wierszu i j -tej kolumnie.

Pozornie wydaje się to bardziej czasochłonne niż sposób pierwszy. Jednakże te same działania wykonuje mikroprocesor również w pierwszym przypadku, tyle tylko, iż kompilator ukrywa je w zapisie $A[i][j]$. Dlatego kompilator musi znać wielkość wymiaru m - w przeciwnym razie nie policzyłby właściwych adresów elementów w obszarze macierzy. Jeśli m jest potęgą liczby 2, to zamiast mnożenia można wykorzystać szybkie przesunięcia bitowe w lewo. Np. dla $m = 4$ będzie:

$B[(i << 2) + j]$ - element w i -tym wierszu i j -tej kolumnie macierzy $B[n][4]$

Algorytmy do zaimplementowania na laboratorium 1 obejmowały utworzenie funkcji o następujących właściwościach:

- utworzenie macierzy
- wypełnienie macierzy randomowymi wartościami
- wyświetlenie macierzy w oknie konsoli
- usunięcie macierzy
- dodawanie do siebie dwóch macierzy
- odejmowanie od siebie dwóch macierzy
- transpozycja macierzy
- kopiowanie macierzy
- mnożenie macierzy

Pliki projektu znajdują się w publicznym repozytorium zdalnym Gita w Githubie. Link do niego: https://github.com/Yaviena/Matrix_operations_AHNS_Algorithms_Lab_1.

Cel laboratorium nr 1

Pierwsze laboratorium miało na celu zaimplementowanie algorytmów podstawowych działań na macierzach. Posłużyły one w dalszej części do mierzenia czasu wykonywania oraz ich złożoności obliczeniowej.

Wykonanie ćwiczenia cz. I: implementacja algorytmów

1. Create, RandomFill, Print, Delete

- Rozpoczęłam od 4 bazowych metod.

W głównej funkcji main() stworzyłam pomocnicze zmienne lokalne określające ilość wierszy i kolumn oraz utworzyłam na ich podstawie pierwszą tablicę.

```
int rowCount = 5;
int colCount = 4;
double** tab1 = CreateTab(rowCount, colCount);
```

Na stworzonej tablicy wywołałam w funkcji main() metody do wypełnienia jej wartościami, wypisania w oknie konsoli oraz usunięcia tablicy.

```
RandomTab(rowCount, colCount, tab1);
PrintTab(rowCount, colCount, tab1);
DeleteTab(rowCount, colCount, tab1);
```

Zgodnie z wcześniejszym wyjaśnieniem i przyjętą konwencją we wstępie teoretycznym, wszystkie funkcje podzieliłam na ich prototypy podane w deklaracji (przed funkcją main()) oraz definicje (za funkcją main()).

Poniżej wklejam jedynie dla ogólnego poglądu podane w deklaracji nagłówki funkcji (prototypy).

Pełny kod wraz z ich definicjami znajduje się na końcu tej części laboratorium.

```
double** CreateTab(int rowCount, int colCount);  
void RandomTab(int rowCount, int colCount, double** tab);  
void PrintTab(int rowCount, int colCount, double** tab);  
void DeleteTab(int rowCount, int colCount, double** tab);
```

Na koniec w funkcji main() usunęłam stworzoną macierz aby zwolnić pamięć.

Zbudowałam i skompilowałam projekt. Wszystko przebiegło pomyślnie w wyniku czego została utworzona macierz o wymiarze 5x4, wypełniona przykładowymi wartościami liczb zmiennoprzecinkowych, wypisana w oknie konsoli oraz usunięta z programu.

Microsoft Visual Studio Debug Console

```
0.41, 0.85, 0.72, 0.38  
0.80, 0.69, 0.65, 0.68  
0.96, 0.22, 0.49, 0.67  
0.51, 0.61, 0.63, 0.87  
0.66, 0.24, 0.80, 0.83
```

2. Adding, Subtraction

Po upewnieniu się, że wszystko kompiluje się poprawnie i wszelkie zmiany są umieszczane w repozytorium zdalnym na Github, analogicznie zajęłam się operacjami dodawania i odejmowania dwóch macierzy.

W funkcji main() stworzyłam dwie kolejne tablice. Jedną z nich wypełniłam przykładowymi wartościami (tab2), a ostatnia posłuży do przechowywania wyniku działań na macierzach.

```
double** tab2 = CreateTab(rowCount, colCount);  
double** tab3 = CreateTab(rowCount, colCount); // to keep the result
```

Utworzyłam prototypy funkcji dodawania i odejmowania podane w deklaracji.

```
void AddTab(int rowCount, int colCount, double** tab1, double** tab2, double** tab3);  
void SubtractTab(int rowCount, int colCount, double** tab1, double** tab2, double** tab3);
```

Wywołałam funkcje dodawania i odejmowania na tablicach tab1 i tab2 przechowując za każdym razem wynik operacji w tab3.

```
cout << endl << "Addition matrix 1 and 2" << endl;  
AddTab(rowCount, colCount, tab1, tab2, tab3);  
PrintTab(rowCount, colCount, tab3);  
  
cout << endl << "Subtraction matrix 1 and 2" << endl;  
SubtractTab(rowCount, colCount, tab1, tab2, tab3);  
PrintTab(rowCount, colCount, tab3);
```

Na koniec w funkcji main() usunęłam stworzone kolejne macierze dla zwolnienia pamięci. Kompilacja projektu przebiegła pomyślnie.

```
Microsoft Visual Studio Debug Console

Matrix 1
0.41, 0.85, 0.72, 0.38
0.80, 0.69, 0.65, 0.68
0.96, 0.22, 0.49, 0.67
0.51, 0.61, 0.63, 0.87
0.66, 0.24, 0.80, 0.83

Matrix 2
0.71, 0.60, 0.64, 0.52
0.90, 0.60, 0.49, 0.31
0.23, 0.99, 0.94, 0.11
0.25, 0.24, 0.51, 0.15
0.13, 0.39, 0.67, 0.97

Addition matrix 1 and 2
1.12, 1.45, 1.36, 0.90
1.70, 1.29, 1.14, 0.99
1.19, 1.21, 1.43, 0.78
0.76, 0.85, 1.14, 1.02
0.79, 0.63, 1.47, 1.80

Subtraction matrix 1 and 2
-0.30, 0.25, 0.08, -0.14
-0.10, 0.09, 0.16, 0.37
0.73, -0.77, -0.45, 0.56
0.26, 0.37, 0.12, 0.72
0.53, -0.15, 0.13, -0.14
```

3. Copy

Kolejnym krokiem było kopiowanie macierzy. W funkcji main() stworzyłam kolejną tablicę, która będzie przechowywać skopiowane wartości z innej tablicy.

```
double** tab4 = CreateTab(colCount, colCount);
```

Utworzyłam prototyp funkcji kopiującej podany w deklaracji.

```
void CopyTab(int rowCount, int colCount, double** tab1, double** tab2);
```

Ciało metody jak zwykle umieściłam w definicji za funkcją main().

Wywołałam funkcję przechowując wynik operacji kopiowania macierzy tab1 w tab4.

```
cout << endl << "Copied matrix 1 to empty matrix" << endl;
CopyTab(rowCount, colCount, tab1, tab4);
PrintTab(rowCount, colCount, tab4);
```

Na koniec w funkcji main() zwolniłam pamięć i pomyślnie skompilowałam projekt.

4. Trans

Transpozycja macierzy jest nieco bardziej wymagająca skupienia. Należy tu pamiętać o zamianie ilości wierszy i kolumn transponowanej macierzy oraz macierzy wynikowej.

W funkcji main() stworzyłam tablicę, która będzie przechowywać przetransponowaną macierz.

```
double** transTab = CreateTab(colCount, rowCount);
```

Utworzyłam prototyp funkcji kopiującej podany w deklaracji.


```
void TransTab(int rowCount, int colCount, double** tab, double** temp_tab);
```

Ciało metody jak zwykle umieściłam w definicji za funkcją main().

Wywołałam funkcję przechowując wynik operacji transponowania macierzy tab1 w transTab i wyświetliłam wynikową przetransponowaną macierz transTab w oknie konsoli.

```
cout << endl << "Transposition of matrix 1" << endl;  
TransTab(rowCount, colCount, tab1, transTab);  
PrintTab(colCount, rowCount, transTab);
```

Na koniec w funkcji main() zwolniłam pamięć i z powodzeniem skompilowałam projekt.

 Microsoft Visual Studio Debug Console

```
Matrix 1  
0.41, 0.85, 0.72, 0.38  
0.80, 0.69, 0.65, 0.68  
0.96, 0.22, 0.49, 0.67  
0.51, 0.61, 0.63, 0.87  
0.66, 0.24, 0.80, 0.83  
  
Transposition of matrix 1  
0.41, 0.80, 0.96, 0.51, 0.66  
0.85, 0.69, 0.22, 0.61, 0.24  
0.72, 0.65, 0.49, 0.63, 0.80  
0.38, 0.68, 0.67, 0.87, 0.83
```

5. Multiplication

Aby pomnożyć dwie macierze muszą one spełniać warunek, iż ilość kolumn pierwszej macierzy jest równa ilości wierszy drugiej. Macierzą wynikową będzie macierz o ilości rzędów takiej jak pierwsza z mnożonych macierzy oraz z ilością kolumn taką samą jak druga mnożona macierz.

$$\begin{bmatrix} 2 & 5 \\ 6 & 8 \\ 1 & 2 \end{bmatrix} \times \begin{bmatrix} 7 & 2 & 1 \\ 8 & 5 & 2 \end{bmatrix} = \begin{bmatrix} 54 & 29 & 12 \\ 106 & 52 & 22 \\ 23 & 12 & 5 \end{bmatrix}$$

3×2 These should be equal 2×3
 3×3
 These are the dimensions of the resulting matrix

W projekcie wykorzystam tablicę tab1 oraz stworzę kolejną tablicę (tab5) o odpowiednich wymiarach.

W funkcji main() stworzyłam pomocniczą zmienną lokalną przechowującą ilość kolumn drugiej z mnożonych macierzy.

```
int colCount2 = 3;
```

Stworzyłam także dwie tablice: jedną o konkretnym wymiarze do pomnożenia (tab5) oraz drugą (tabMultiplication), która będzie przechowywać rezultat mnożenia macierzy tab1 oraz tab5.

```
double** tab5 = CreateTab(colCount, colCount2);
double** tabMultiplication = CreateTab(rowCount, colCount2);
```

Utworzyłam prototyp funkcji mnożącej macierze podany w deklaracji.

```
void MulTab(int rowCount, int colCount, int colCount2, double** tab1, double** tab2,
double** tab3);
```

Ciało metody jak zwykle umieściłam w definicji za funkcją main().

Wypełniłam przykładowymi wartościami i wyświetliłam tab5 aby móc sprawdzić w następnym kroku poprawność działania mnożenia na obu tablicach.

```
cout << endl << "Matrix 5 to multiplication" << endl;
RandomTab(colCount, colCount2, tab5);
PrintTab(colCount, colCount2, tab5);
```

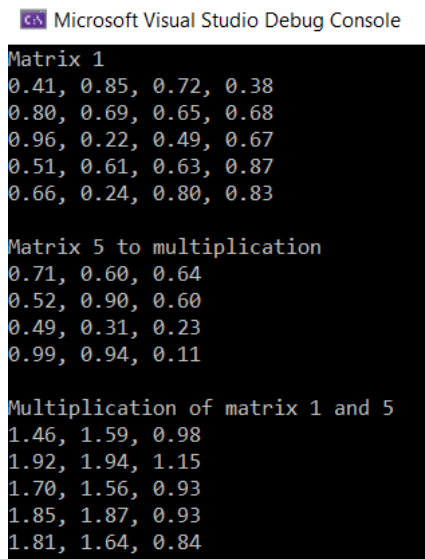
Wywołałam funkcję podając za argumenty kolejno zgodnie z definicją:

- ilość wierszy pierwszej macierzy
- ilość kolumn pierwszej macierzy
- ilość kolumn drugiej macierzy
- pierwsza macierz (tab1) do mnożenia
- drugą macierz (tab5) do mnożenia
- macierz wynikową (tabMultiplication) przechowującą wynik operacji mnożenia

Następnie wyświetliłam wynikową macierz w oknie konsoli.

```
cout << endl << "Multiplication of matrix 1 and 5" << endl;
MulTab(rowCount, colCount, colCount2, tab1, tab5, tabMultiplication);
PrintTab(rowCount, colCount2, tabMultiplication);
```

Na koniec w funkcji main() zwolniłam pamięć i z powodzeniem skompilowałam projekt.



Microsoft Visual Studio Debug Console

```
Matrix 1
0.41, 0.85, 0.72, 0.38
0.80, 0.69, 0.65, 0.68
0.96, 0.22, 0.49, 0.67
0.51, 0.61, 0.63, 0.87
0.66, 0.24, 0.80, 0.83

Matrix 5 to multiplication
0.71, 0.60, 0.64
0.52, 0.90, 0.60
0.49, 0.31, 0.23
0.99, 0.94, 0.11

Multiplication of matrix 1 and 5
1.46, 1.59, 0.98
1.92, 1.94, 1.15
1.70, 1.56, 0.93
1.85, 1.87, 0.93
1.81, 1.64, 0.84
```

Wykonanie ćwiczenia cz. II: pomiar czasów wykonywania

Drugą część laboratoriów stanowił pomiar czasów wykonywania zaimplementowanych algorytmów. Dzisiejsze komputery oferują ogromną moc obliczeniową w krótkim czasie. Wykonanie operacji dodawania czy odejmowania na macierzy o wymiarach chociażby 5x4 jest tak szybkie, że procesor postrzega ten czas jako bliski zeru.

Czas wykonywania algorytmów wyznaczyłam na II sposoby, gdyż pierwszy nie dawał rezultatów przy mniejszych macierzach a zastosowanie innej metody pokazało dużą rozbieżność.

1. Funkcja GetTickCount()

W pierwszej kolejności użyłam GetTickCount(). Zwraca ona typ całkowity (int) jako liczbę milisekund, która upłynęła pomiędzy dwoma wydarzeniami. Wymaga użycia biblioteki windows.h. Biorąc pod uwagę fakt, iż mój system operacyjny jest 64-bitowy, dla prawidłowego funkcjonowania użyję odmiany GetTickCount64() tej funkcji.

Na przykładzie macierzy 1 (tab1) omówię mierzenie czasu wypełniania jej wartościami i wyświetlenia na ekranie.

Tworzę w funkcji main() zmienne lokalne do rozpoczęcia i zakończenia pomiaru.

```
int startT1, stopT1;
```

Utworzyłam prototyp funkcji obliczającej czas wykonywania pierwszą metodą podany w deklaracji.

```
void ExecutionTimeByGetTickCount64(int startTime, int stopTime);
```

Ciało metody jak zwykle umieściłam w definicji za funkcją main(). Dla lepszej widoczności wyniku zmieniłam kolor wyświetlanego wewnątrz komunikatu na żółty.

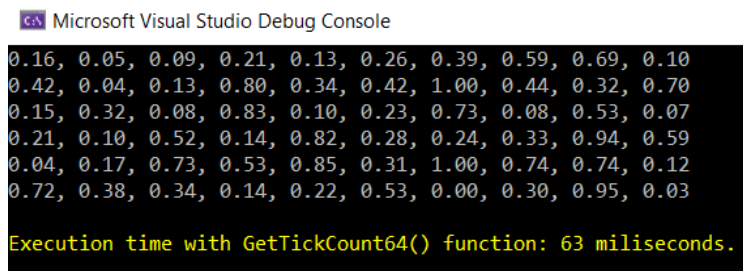
Bezpośrednio przed wywołaniem funkcji wypełniające tablicę oraz bezpośrednio po funkcji wypisującej dane na ekran wywołuję funkcję GetTickCount64() na zmiennych lokalnych i zapisuję w nich wynik (zmierzony czas).

```
startT1 = (int)GetTickCount64();  
stopT1 = (int)GetTickCount64();
```

Aby uzyskać czas wykonywania wywołałam funkcję podając jako argumenty powyższe zmienne przechowujące czasy.

```
ExecutionTimeByGetTickCount64(startT1, stopT1);
```

Kompilacja przebiegła pomyślnie a przykładowy wynik jaki otrzymałam na konsoli dla tab1 o wymiarach 50x10 to 47 milisekund. Poniżej kluczowy wycinek z komunikatem.



Microsoft Visual Studio Debug Console

```
0.16, 0.05, 0.09, 0.21, 0.13, 0.26, 0.39, 0.59, 0.69, 0.10  
0.42, 0.04, 0.13, 0.80, 0.34, 0.42, 1.00, 0.44, 0.32, 0.70  
0.15, 0.32, 0.08, 0.83, 0.10, 0.23, 0.73, 0.08, 0.53, 0.07  
0.21, 0.10, 0.52, 0.14, 0.82, 0.28, 0.24, 0.33, 0.94, 0.59  
0.04, 0.17, 0.73, 0.53, 0.85, 0.31, 1.00, 0.74, 0.74, 0.12  
0.72, 0.38, 0.34, 0.14, 0.22, 0.53, 0.00, 0.30, 0.95, 0.03  
  
Execution time with GetTickCount64() function: 63 milliseconds.
```

Po przykładowych 7 kompilacjach okazało się, że czas wykonywania wynosił kolejno:

- 47 ms
- 62 ms
- 47 ms
- 63 ms
- 47 ms
- 62 ms
- 46 ms

a więc **średnio 53.43 ms**.

Zastosowanie tego samego sposobu analogicznie przy mnożeniu macierzy dało poniższe wyniki przy 7 próbach:

- 62 ms
- 78 ms
- 47 ms
- 62 ms
- 31 ms
- 47 ms
- 47 ms

a więc **średnio 53.43 ms**.

Przy zwiększeniu wymiarów mnożonych macierzy do 500x100 i 100x3, czas wykonywania przy 7 próbach przedstawiał się następująco:

- 328 ms
- 344 ms
- 328 ms
- 329 ms
- 328 ms
- 343 ms
- 359 ms

a więc **średnio 337 ms**.

```
Microsoft Visual Studio Debug Console
21.77, 22.78, 24.29
22.34, 23.81, 23.14
23.20, 22.97, 24.35
21.68, 20.85, 22.04
22.52, 22.87, 21.38
21.18, 20.50, 21.17
Execution time with GetTickCount64() function: 328 milliseconds.
```

Stosując opisany przebieg mierząc wyłącznie czas tworzenia macierzy i jej wypełnienia (bez wypisywania danych na ekran), czas wynosił 0 ms nawet dla bardzo dużych macierzy. Oznacza to, że to procesor oblicza i wykonuje polecenia bardzo szybko a to jedynie wypisanie danych na ekran zajmuje jakikolwiek czas dostrzegalny "gołym okiem".

Przez wzgląd na prostotę i złożoność pozostałych funkcji, jedyną interesującą funkcją pod względem czasu wykonywania algorytmu jest jeszcze transpozycja. Implementacja pomiaru czasu wykonywania na macierzy poddanej transpozycji znajduje się w finalnym kompletnym kodzie na końcu tej części sprawozdania.

2. Funkcja clock()

Jako drugiej użyłam funkcji clock(). Zwraca ona typ zmiennoprzecinkowy (double) jako liczbę sekund tym razem, jaka upłynęła pomiędzy dwoma wydarzeniami. analogicznie do pierwszej metody wykonałam pomiar za pomocą clock() na przykładzie macierzy 1 (tab1).

Deklaruję zmienne lokalne do rozpoczęcia i zakończenia pomiaru. Obie zmienne reprezentują liczbę cykli procesora w momencie startu i zakończenia odliczania

```
clock_t startT2, stopT2;
```

Utworzyłam prototyp funkcji obliczającej czas wykonywania pierwszą metodą podany w deklaracji.

```
void ExecutionTimeByClock(int startTime, int stopTime);
```

Ciało metody jak zwykle umieściłam w definicji za funkcją main(). Dla lepszej widoczności wyniku zmieniłam kolor wyświetlanego wewnątrz komunikatu na niebieski.

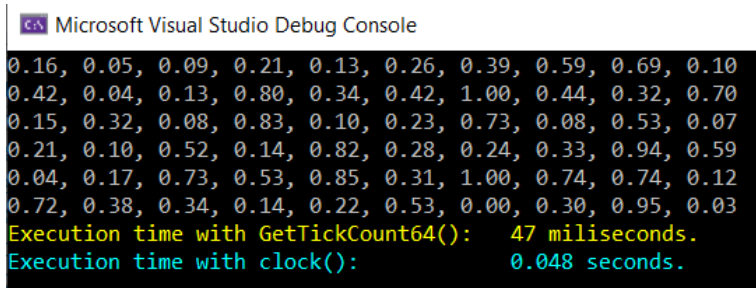
Bezpośrednio przed wywołaniem funkcji wypełniającej tablicę oraz bezpośrednio po wypisaniu danych na ekran wywołuję funkcję GetTickCount64() na zmiennych lokalnych i zapisuję w nich wynik (zmierzony czas).

```
startT2 = clock();  
stopT2 = clock();
```

Aby uzyskać czas wykonywania wywołałam funkcję podając jako argumenty powyższe zmienne przechowujące czasy.

```
ExecutionTimeByClock(startT2, stopT2);
```

Kompilacja przebiegła pomyślnie a przykładowy wynik jaki otrzymałam w oknie konsoli dla tab1 o wymiarach 50x10 to 0.048 sekund. Poniżej kluczowy wycinek z wynikami obu metod.



The screenshot shows the Microsoft Visual Studio Debug Console with a list of 50 floating-point numbers representing matrix elements. Below the list, two lines of output are shown: 'Execution time with GetTickCount64(): 47 milliseconds.' in yellow and 'Execution time with clock(): 0.048 seconds.' in blue.

```
0.16, 0.05, 0.09, 0.21, 0.13, 0.26, 0.39, 0.59, 0.69, 0.10  
0.42, 0.04, 0.13, 0.80, 0.34, 0.42, 1.00, 0.44, 0.32, 0.70  
0.15, 0.32, 0.08, 0.83, 0.10, 0.23, 0.73, 0.08, 0.53, 0.07  
0.21, 0.10, 0.52, 0.14, 0.82, 0.28, 0.24, 0.33, 0.94, 0.59  
0.04, 0.17, 0.73, 0.53, 0.85, 0.31, 1.00, 0.74, 0.74, 0.12  
0.72, 0.38, 0.34, 0.14, 0.22, 0.53, 0.00, 0.30, 0.95, 0.03  
Execution time with GetTickCount64(): 47 milliseconds.  
Execution time with clock(): 0.048 seconds.
```

Powyższą metodę zaimplementowałam analogicznie do dwóch pozostałych kluczowych algorytmów: mnożenia oraz transpozycji macierzy.

Kompletne wnioski znajdują się na końcu sprawozdania w sekcji [Wnioski](#).

Laboratoria nr 2

Wstęp

Informacja, ile czasu dany algorytm był wykonywany jest niewystarczająca. Dodatkowo trzeba wiedzieć, jak wydłuża się czas wykonywania wraz ze zwiększaniem liczby elementów. Notacja dużego O to specjalny sposób opisu szybkości działania algorytmów. Ta notacja nie wyraża szybkości w sekundach. Notacja dużego O pozwala porównać liczbę operacji do wykonania. Informuje, jak szybko rośnie czas wykonywania algorytmu. Zawsze opisuje najgorszy przypadek to rodzaj zapewnienia, że wyszukiwanie proste na pewno nie będzie trwać dłużej niż $O(n)$.

Szybkość wykonywania algorytmów nie wyraża się w sekundach, tylko w tempie wzrostu liczby operacji. Omawiając szybkość działania algorytmu, podaje się, jak szybko rośnie czas wykonywania wraz ze zwiększaniem rozmiaru zbioru wejściowego. Dla przykładu, algorytm o czasie wykonywania $O(\log n)$ jest szybszy niż $O(n)$, a im większy zbiór danych do przeszukania, tym większa robi się różnica.

Przykładowe algorytmy sortowania:

- sortowanie bąbelkowe (ang. bubblesort)	$O(n^2)$
- sortowanie przez wstawianie (ang. insertion sort)	$O(n^2)$
- sortowanie przez scalanie (ang. merge sort)	$O(n \log n)$
- wymaga dodatkowej pamięci	$O(n)$
- sortowanie przez zliczanie (ang. counting sort)	$O(n+k)$
- wymaga dodatkowej pamięci	
- sortowanie kubelkowe (ang. bucket sort)	$O(n)$
- wymaga dodatkowej pamięci	
- sortowanie biblioteczne (ang. library sort)	$O(n \log n)$
- sortowanie przez wybieranie (ang. selection sort)	$O(n^2)$
- sortowanie Shella – (ang. shellsort)	$O(n \log n)$
- sortowanie szybkie – (ang. quicksort)	$O(n \log n)$
- sortowanie przez kopcowanie – (ang. heapsort)	$O(n \log n)$

Cel laboratorium nr 2

Na podstawie wyników czasów wykonywania sześciu algorytmów należy zbudować charakterystyki złożoności obliczeniowej (na jednym wspólnym rysunku na całej stronie A4) dla wszystkich algorytmów podanych w programie będącym materiałem do laboratorium. Dopuszczalne jest rozłożenie charakterystyk na 2 wykresy dla lepszej czytelności wzrostu tempa czasów ich wykonywania.

Wszystkie pliki projektu znajdują się w publicznym repozytorium zdalnym Gita w Githubie. Link do niego:

https://github.com/Yaviena/Algorithms_Lab_2_Sorting_diagrams_Magda_Szafranska

Wykonanie ćwiczenia cz. I: czasy wykonywania

Uruchomienie bazowej aplikacji

Dla swobodnej pracy utworzyłam nowy projekt w środowisku Code::Blocks. Umieściłam w nim skopiowany kod z otrzymanego od Pana doktora pliku "main.cpp".

- **Prześledzenie działania programu.**

Cały kod programu wkleiłam na końcu tej części ćwiczenia.

Program składał się z zaimplementowanych 6 podstawowych rodzajów sortowania:

- Sortowanie bąbelkowe (ang. bubble sort)
- Sortowanie przez wybór (ang. selection sort)
- Sortowanie przez wstawianie (ang. insertion sort)
- Sortowanie przez scalanie (ang. merge sort)
- Sortowanie szybkie (ang. quick sort)
- Sortowanie przez zliczanie (ang. counting sort)

W programie znajdowało się również kilka funkcji pomocniczych powiązanych z powyższymi algorytmami.

Dla poprawności działania bibliotek, z których program korzystał, zmieniłam funkcję

```
T[i] = random(N);
```

na

```
T[i] = rand()%N;
```

- **Analiza funkcji głównej main() programu.**

Wewnątrz funkcji main() znalazły się wywołania poszczególnych funkcji algorytmów sortowania pokazujące czas ich wykonywania dla określonej wielkości tablicy. Tablica tworzona jest na początku na podstawie wymiaru podanego przez użytkownika przy uruchamianiu aplikacji. Podawany parametr N określa długość naszej tablicy. Jej zawartość jest losowana i wyświetlana (maksymalnie do 30 znaków).

- **Funkcje obliczające czasy sortowania.**

W funkcji main() mamy następnie dwie bardzo istotne, skojarzone ze sobą kluczowe funkcje:

- `QueryPerformanceFrequency((LARGE_INTEGER*)&F);` - zwraca częstotliwość pracy procesora
- `QueryPerformanceCounter((LARGE_INTEGER*)&T1);` - zwraca liczbę cykli wykonanych przez procesor od momentu uruchomienia komputera.

Obie te funkcje służą do wyznaczania czasu pracy procesora. Mogą być użyte do aproksymacji czasu pracy danego algorytmu - co posłużyło mi w drugiej części laboratorium.

Przeanalizuję ich zależność na przykładzie sortowania bąbelkowego jak poniżej:

```
__int64 T1, T2, F;
QueryPerformanceFrequency((LARGE_INTEGER*)&F);

printf("Sortowanie bąbelkowe (ang. bubble sort)...", N);
RandomTab(N, T);
QueryPerformanceCounter((LARGE_INTEGER*)&T1);
BubbleSort(N, T);
QueryPerformanceCounter((LARGE_INTEGER*)&T2);
printf("\t\tCzas [s] = %G\n", (T2 - T1) / (float)F);
```

Liczba cykli wykonywanych przez procesor zwracana jest do zmiennej całkowitej 64-bitowej T1. Jeżeli odczytamy taki parametr (T1) tuż przed uruchomieniem danego algorytmu oraz tuż po jego wykonaniu (T2), to następnie różnica tych dwóch wielkości (T2-T1) da nam liczbę taktów procesora, które były potrzebne do wykonania danego algorytmu. Jeśli podzielimy tę liczbę taktów przez częstotliwość, którą wcześniej odczytaliśmy, otrzymamy czas przetwarzania danego algorytmu w sekundach.

Proces ten powtarza się w programie dla wszystkich sześciu algorytmów sortujących. Każdy z tych czasów jest wyświetlany.

- **Implementacja zapisu pomiarów do pliku**

Aby zebrać dane potrzebne do sporządzania wykresów, dodałam funkcje zapisu do plików z biblioteki *fstream*. Dla wygody późniejszego kopiowania danych do arkusza Google Sheets, pomiary czasów dla każdego z algorytmów zapisywałam w osobnym pliku (z opcją nadpisywania). Poniżej adekwatny wycinek kodu.

```
163 // writing to the text file - a separate file to each of an algorithm
164 fstream fileN1, fileN2, fileBubble, fileSelection, fileInsertion, fileMerge, fileQuick, fileCounting;
165 fileN1.open("N1.txt", ios::out | ios::app);
166 fileN2.open("N2.txt", ios::out | ios::app);
167 fileBubble.open("BubbleSort.txt", ios::out | ios::app);
168 fileSelection.open("SelectionSort.txt", ios::out | ios::app);
169 fileInsertion.open("InsertionSort.txt", ios::out | ios::app);
170 fileMerge.open("MergeSort.txt", ios::out | ios::app);
171 fileQuick.open("QuickSort.txt", ios::out | ios::app);
172 fileCounting.open("CountingSort.txt", ios::out | ios::app);
173
174 int* T = new int[N];
175
176 RandomTab(N, T);
177 printf("T = "); WriteTab(MIN(N, 30), T);
178 fileN1 << N << endl;
179 fileN2 << N << endl;
```

- **Zbudowanie aplikacji**

Po analizie kodu zbudowałam i skompilowałam program generując tym samym plik wykonywalny .exe. Do kompilacji użyłam kompilatora C++ MinGW zawierającego się w moim środowisku programistycznym Code::Blocks.

- **Kod programu**

Poniżej znajduje się cały kod programu. Jest do pobrania również w zdalnym repozytorium razem z całym projektem. Link do niego na początku sekcji Laboratorium nr 2.

```
// Autor: Magda Szafranska, nr indeksu AHNS: 18345
// Informatyka NST, rok 2, sem. 3
// Algorytmy, laboratoria nr 2

using namespace std;

//-----
#include <tchar.h>
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <random>
#include <fstream>

//-----
int MIN(int A, int B)
{
    return A < B ? A : B;
}
//-----
void SWAP(int& A, int& B)
{
    int T = A;
    A = B;
    B = T;
}
//-----
void RandomTab(int N, int* T)
{
    srand(0);
    for (int i = 0; i < N; ++i) T[i] = rand() % N;
}
//-----
void WriteTab(int N, int* T)
{
    printf("%i", T[0]);
    for (int i = 1; i < N; ++i) printf(", %i", T[i]);
    printf("\n");
}
//-----
int MaxOfTab(int N, int* T)
{
    int M = T[0];
    for (int i = 1; i < N; ++i) if (M < T[i]) M = T[i];
    return M;
}
//-----
//-----
void BubbleSort(int N, int* T)
{
    bool Changed;
    do {
        Changed = false;
        for (int i = N - 2; i >= 0; --i)
            if (T[i] > T[i + 1]) {
                SWAP(T[i], T[i + 1]);
                Changed = true;
            }
    }
}
```

```

    } while (Changed);
}
//-----
void InsertionSort(int N, int* T)
{
    for (int i = 1; i < N; ++i) {
        int j = i;
        int V = T[i];
        while ((T[j - 1] > V) && (j > 0)) T[j--] = T[j - 1];
        T[j] = V;
    }
}
//-----
void SelectionSort(int N, int* T)
{
    for (int i = 0; i < N - 1; ++i) {
        int m = i;
        for (int j = i + 1; j < N; ++j) if (T[j] < T[m]) m = j;
        SWAP(T[m], T[i]);
    }
}
//-----
void QS(int I1, int I2, int* T)
{
    int i = I1;
    int j = I2;
    int V = T[(I1 + I2) / 2];
    do
    {
        while (T[i] < V) i++;
        while (V < T[j]) j--;
        if (i <= j) SWAP(T[i++], T[j--]);
    } while (i <= j);
    if (I1 < j) QS(I1, j, T);
    if (i < I2) QS(i, I2, T);
}

void QuickSort(int N, int* T)
{
    QS(0, N - 1, T);
}
//-----
void CountingSort(int N, int* T)
{
    int M = MaxOfTab(N, T);

    int* P = new int[M];
    for (int i = 0; i < M; ++i) P[i] = 0;
    for (int i = 0; i < N; ++i) ++P[T[i]];
    for (int i = 0, j = 0; (i < N) && (j < M);)
        if (P[j] > 0) {
            T[i++] = j;
            P[j]--;
        }
        else j++;
    delete[] P;
}
//-----
void MM(int I1, int K, int I2, int* T, int* P)
{
    for (int i = I1; i <= I2; ++i) P[i] = T[i];
    int i = I1;
    int j = K + 1;
    int q = I1;
    while ((i <= K) && (j <= I2)) {

```

```

        if (P[i] < P[j]) T[q++] = P[i++];
        else T[q++] = P[j++];
    }
    while (i <= K) T[q++] = P[i++];
}

void MS(int I1, int I2, int* T, int* P)
{
    if (I1 < I2) {
        int k = (I1 + I2) / 2;
        MS(I1, k, T, P);
        MS(k + 1, I2, T, P);
        MM(I1, k, I2, T, P);
    }
}

void MergeSort(int N, int* T)
{
    int* P = new int[N];
    MS(0, N - 1, T, P);
    delete[] P;
}

//-----
int main(int argc, char* argv[])
{
    printf("<<< Test algorytmów sortowania tablicy liczb ca³kowitych >>>\n");
    if (argc != 2) {
        printf("Schemat wywo³ania programu: sort tab_size\n");
        printf("  tab_size - rozmiar tablicy\n");
        return 0;
    }

    int N = atoi(argv[1]);
    printf("N = %i\n", N);
    if (N <= 0) {
        printf("Nieprawid³owy rozmiar tablicy!\n");
        return 0;
    }

    // writing to the text file - a separate file to each of an algorithm
    fstream fileN1, fileN2, fileBubble, fileSelection, fileInsertion, fileMerge, fileQuick,
fileCounting;
    fileN1.open("N1.txt", ios::out | ios::app);
    fileN2.open("N2.txt", ios::out | ios::app);
    fileBubble.open("BubbleSort.txt", ios::out | ios::app);
    fileSelection.open("SelectionSort.txt", ios::out | ios::app);
    fileInsertion.open("InsertionSort.txt", ios::out | ios::app);
    fileMerge.open("MergeSort.txt", ios::out | ios::app);
    fileQuick.open("QuickSort.txt", ios::out | ios::app);
    fileCounting.open("CountingSort.txt", ios::out | ios::app);

    int* T = new int[N];

    RandomTab(N, T);
    printf("T = "); WriteTab(MIN(N, 30), T);
    fileN1 << N << endl;
    fileN2 << N << endl;

    __int64 T1, T2, F;
    QueryPerformanceFrequency((LARGE_INTEGER*)&F);

    printf("Sortowanie b³belkowe (ang. bubble sort)...", N);
    RandomTab(N, T);
    QueryPerformanceCounter((LARGE_INTEGER*)&T1);
    BubbleSort(N, T);

```

```

QueryPerformanceCounter((LARGE_INTEGER*)&T2);
printf("\t\tCzas [s] = %G\n", (T2 - T1) / (float)F);
fileBubble << (T2 - T1) / (float)F << endl;

printf("Sortowanie przez wybór (ang. selection sort)...", N);
RandomTab(N, T);
QueryPerformanceCounter((LARGE_INTEGER*)&T1);
SelectionSort(N, T);
QueryPerformanceCounter((LARGE_INTEGER*)&T2);
printf("\t\tCzas [s] = %G\n", (T2 - T1) / (float)F);
fileSelection << (T2 - T1) / (float)F << endl;

printf("Sortowanie przez wstawianie (ang. insertion sort)...", N);
RandomTab(N, T);
QueryPerformanceCounter((LARGE_INTEGER*)&T1);
InsertionSort(N, T);
QueryPerformanceCounter((LARGE_INTEGER*)&T2);
printf("\t\tCzas [s] = %G\n", (T2 - T1) / (float)F);
fileInsertion << (T2 - T1) / (float)F << endl;

printf("Sortowanie przez scalanie (ang. merge sort)...", N);
RandomTab(N, T);
QueryPerformanceCounter((LARGE_INTEGER*)&T1);
MergeSort(N, T);
QueryPerformanceCounter((LARGE_INTEGER*)&T2);
printf("\t\tCzas [s] = %G\n", (T2 - T1) / (float)F);
fileMerge << (T2 - T1) / (float)F << endl;

printf("Sortowanie szybkie (ang. quick sort)...", N);
RandomTab(N, T);
QueryPerformanceCounter((LARGE_INTEGER*)&T1);
QuickSort(N, T);
QueryPerformanceCounter((LARGE_INTEGER*)&T2);
printf("\t\t\tCzas [s] = %G\n", (T2 - T1) / (float)F);
fileQuick << (T2 - T1) / (float)F << endl;

printf("Sortowanie przez zliczanie (ang. counting sort)...", N);
RandomTab(N, T);
QueryPerformanceCounter((LARGE_INTEGER*)&T1);
CountingSort(N, T);
QueryPerformanceCounter((LARGE_INTEGER*)&T2);
printf("\t\tCzas [s] = %G\n", (T2 - T1) / (float)F);
fileCounting << (T2 - T1) / (float)F << endl;

printf("T = "); WriteTab(MIN(N, 30), T);

    fileN1.close();
    fileN2.close();
    fileBubble.close();
    fileSelection.close();
    fileInsertion.close();
    fileMerge.close();
    fileQuick.close();
    fileCounting.close();

delete[] T;
return 0;
}
//-----

```

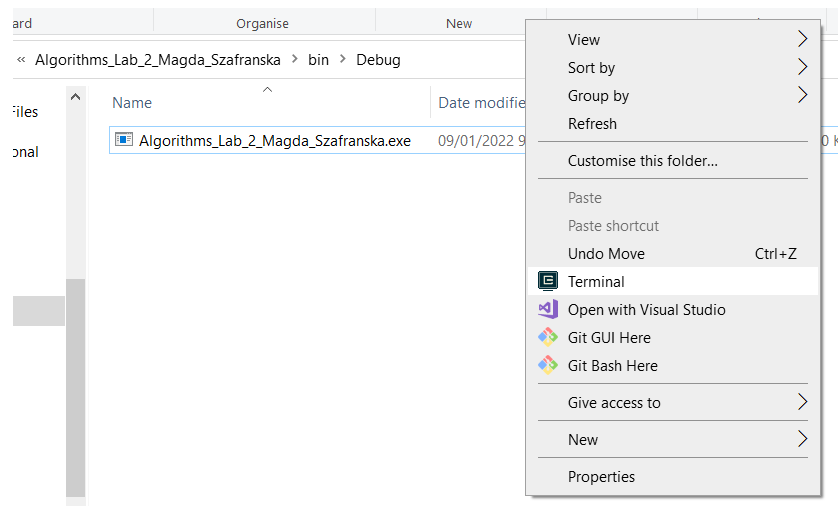
Pomiar czasów wykonywania

- **Uruchomienie aplikacji**

Aby uruchomić plik .exe przeszedłem do folderu "Debug", w którym się on znajdował:

C:\..\Algorithms_Lab_2_Graphs_of_sorting_complex\Algorithms_Lab_2_Magda_Szafranska\bin\Debug

Klikając PPM otworzyłam w tym folderze terminal jak poniżej.



W terminalu uruchomiłam plik wykonywalny "Algorithms_Lab_2_Magda_Szafranska.exe" podając po odstępach po nazwie pliku przykładowy parametr 1000 określający wielkość tablicy do posortowania.

```
Debug - "C:\Users\Modify\Downloads\All\Algorytmy\Algorithms_Lab_2_Graphs_of_sorting_complex\Algorithms_Lab_2_Magda_Szafranska\bin\Debug"
C:\Users\Modify\Downloads\All\Algorytmy\Algorithms_Lab_2_Graphs_of_sorting_complex\Algorithms_Lab_2_Magda_Szafranska\bin\Debug (main)
> Algorithms_Lab_2_Magda_Szafranska.exe 1000
<<< Test algorytmów sortowania tablicy liczb całkowitych >>>
N = 1000
T = 38, 719, 238, 437, 855, 797, 365, 285, 450, 612, 853, 100, 142, 281, 537, 921, 945, 285, 997, 680, 976, 891, 655, 906, 457, 323, 881, 240, 725, 278
Sortowanie bąbelkowe (ang. bubble sort)...      Czas [s] = 0.0032798
Sortowanie przez wybór (ang. selection sort)...  Czas [s] = 0.0012405
Sortowanie przez wstawianie (ang. insertion sort)...  Czas [s] = 0.0006189
Sortowanie przez scalanie (ang. merge sort)...    Czas [s] = 0.0001194
Sortowanie szybkie (ang. quick sort)...          Czas [s] = 8.9E-005
Sortowanie przez zliczanie (ang. counting sort)...  Czas [s] = 2.15E-005
T = 1, 1, 2, 2, 4, 4, 5, 5, 5, 6, 7, 7, 9, 10, 10, 12, 14, 15, 16, 18, 18, 19, 19, 20, 20, 20, 20, 21, 21, 23
```

Kolejne pomiary znajdują się poniżej w drugiej części zadania.

Wykonanie ćwiczenia cz. II: charakterystyki złożoności

Generowanie wyników aplikacji

Jedno uruchomienie algorytmu to jeden punkt na charakterystyce złożoności. Zadanie polegało na wyznaczenie kilkudziesięciu takich punktów dla różnych N. Zaczęłam gromadzenie wyników dla zbudowania charakterystyk od dużych wartości N takich, aby czas dla sortowania bąbelkowego (dla niego będzie najdłuższy) był stosunkowo długi, do około 1 minuty.

Pierwszy pomiar dla N = 1000 elementów okazał się za mały dlatego ponownie wywołałam program używając za każdym razem innej wielkości N uzależnionej od otrzymywanego czasu wykonywania dla algorytmu sortowania bąbelkowego.

Poniżej zrzuty ekranu kilku przykładowych wyników:

- dla N = 120 000, czas (sortowania bąbelkowego) ≈ 65 s

```
Debug - "C:\Users\Modify\Downloads\All\Algorytmy\Algorithms_Lab_2_Graphs_of_sorting_complex\Algorithms_Lab_2_Magda_Szafranska\bin\Debug"
C:\Users\Modify\Downloads\All\Algorytmy\Algorithms_Lab_2_Graphs_of_sorting_complex\Algorithms_Lab_2_Magda_Szafranska\bin\Debug (main)
A Algorithms_Lab_2_Magda_Szafranska.exe 120000
<<< Test algorytm w sortowania tablicy liczb całkowitych >>>
N = 120000
T = 38, 7719, 21238, 2437, 8855, 11797, 8365, 32285, 10450, 30612, 5853, 28100, 1142, 281, 20537, 15921, 8945, 26285, 2997, 14680, 20976, 31891, 21655, 25906, 18457, 1323, 28881, 2240, 9725, 32278
Sortowanie bąbelkowe (ang. bubble sort)... Czas [s] = 65.0123
Sortowanie przez wybór (ang. selection sort)... Czas [s] = 16.8886
Sortowanie przez wstawianie (ang. insertion sort)... Czas [s] = 8.41247
Sortowanie przez scalanie (ang. merge sort)... Czas [s] = 0.0203716
Sortowanie szybkie (ang. quick sort)... Czas [s] = 0.0156342
Sortowanie przez zliczanie (ang. counting sort)... Czas [s] = 0.0014635
T = 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 7, 7, 7
```

- dla N = 115 000, czas ≈ 60 s

```
Debug - "C:\Users\Modify\Downloads\All\Algorytmy\Algorithms_Lab_2_Graphs_of_sorting_complex\Algorithms_Lab_2_Magda_Szafranska\bin\Debug"
C:\Users\Modify\Downloads\All\Algorytmy\Algorithms_Lab_2_Graphs_of_sorting_complex\Algorithms_Lab_2_Magda_Szafranska\bin\Debug (main)
A Algorithms_Lab_2_Magda_Szafranska.exe 115000
<<< Test algorytm w sortowania tablicy liczb całkowitych >>>
N = 115000
T = 38, 7719, 21238, 2437, 8855, 11797, 8365, 32285, 10450, 30612, 5853, 28100, 1142, 281, 20537, 15921, 8945, 26285, 2997, 14680, 20976, 31891, 21655, 25906, 18457, 1323, 28881, 2240, 9725, 32278
Sortowanie bąbelkowe (ang. bubble sort)... Czas [s] = 60.1761
Sortowanie przez wybór (ang. selection sort)... Czas [s] = 15.5655
Sortowanie przez wstawianie (ang. insertion sort)... Czas [s] = 7.69804
Sortowanie przez scalanie (ang. merge sort)... Czas [s] = 0.01899
Sortowanie szybkie (ang. quick sort)... Czas [s] = 0.0145236
Sortowanie przez zliczanie (ang. counting sort)... Czas [s] = 0.0012101
T = 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 7, 7, 7
```

- dla N = 110 000, czas ≈ 54 s

```
Debug - "C:\Users\Modify\Downloads\All\Algorytmy\Algorithms_Lab_2_Graphs_of_sorting_complex\Algorithms_Lab_2_Magda_Szafranska\bin\Debug"
C:\Users\Modify\Downloads\All\Algorytmy\Algorithms_Lab_2_Graphs_of_sorting_complex\Algorithms_Lab_2_Magda_Szafranska\bin\Debug (main)
A Algorithms_Lab_2_Magda_Szafranska.exe 110000
<<< Test algorytm w sortowania tablicy liczb całkowitych >>>
N = 110000
T = 38, 7719, 21238, 2437, 8855, 11797, 8365, 32285, 10450, 30612, 5853, 28100, 1142, 281, 20537, 15921, 8945, 26285, 2997, 14680, 20976, 31891, 21655, 25906, 18457, 1323, 28881, 2240, 9725, 32278
Sortowanie bąbelkowe (ang. bubble sort)... Czas [s] = 54.3976
Sortowanie przez wybór (ang. selection sort)... Czas [s] = 14.1472
Sortowanie przez wstawianie (ang. insertion sort)... Czas [s] = 7.04702
Sortowanie przez scalanie (ang. merge sort)... Czas [s] = 0.0189447
Sortowanie szybkie (ang. quick sort)... Czas [s] = 0.0144428
Sortowanie przez zliczanie (ang. counting sort)... Czas [s] = 0.0012274
T = 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 7, 7, 7
```

Tworzenie wykresu dla sortowania bąbelkowego, przez wybór i przez wstawianie

- **Założenia wstępne.**

Według zaleceń Pana Profesora, rysunek powinien być zbudowany na jednej kartce A4 (jednym wspólnym wykresie) zorientowanej pionowo, a więc oś N będzie na krótszym boku a oś czasu na dłuższym.

Dążyłam do uzyskania jak najlepszej widoczności wykresu, niemniej już przy pierwszych uruchomieniach programu widać było wyraźną rozbieżność wyników poszczególnych algorytmów. Szczególnie dla trzech ostatnich ciężko byłoby wyznaczyć charakterystyki na jednym wspólnym wykresie z trzema pierwszymi tak, aby zachować czytelność. Idąc za sugestią Pana Profesora pokusiłam się o dwa osobne rysunki:

- jeden rysunek dla trzech pierwszych rodzajów sortowania:
 - ◆ sortowanie bąbelkowe (ang. bubble sort)
 - ◆ sortowanie przez wybór (ang. selection sort)
 - ◆ sortowanie przez wstawianie (ang. insertion sort)
- drugi dla trzech kolejnych (przez scalanie, szybkie, przez zliczanie)
 - ◆ sortowanie przez scalanie (ang. merge sort)
 - ◆ sortowanie szybkie (ang. quick sort)
 - ◆ sortowanie przez zliczanie (ang. counting sort)

- **Dopasowanie skali osi.**

Maksymalna wartość na osi czasu to ok. 60 sekund. N, czyli ilość elementów sortowanej tablicy, dla którego czas wykonywania algorytmu bąbelkowego będzie najbardziej zbliżony do czasu 60 sekund, będzie tym samym moją maksymalną wartością na osi poziomej.

Spośród wszystkich poprzednich eksperymentalnych uruchomień programu wynika, że dla $N=115000$ czas wykonywania to ok. 60 sekund a więc je właśnie przyjąłam za moje skrajne wartości na obu osiach. Następnie oś czasu podzieliłam proporcjonalnie według skali liniowej na odcinki czasowe (co 1 cm) po czym uruchamiałam program dobierając za każdym razem taki parametr N, jaki wynika z podziału tej osi poziomej. Wykonałam kilkanaście takich punktów pomiarowych (uruchomień programu) dla każdego algorytmu.

- **Charakterystyka złożoności dla pierwszych trzech algorytmów.**

Poniżej prezentuję wyniki zapisywane do plików przy 30 kolejnych uruchomieniach programu. Każdy plik to czas wykonywania innego algorytmu (adekwatnie do nazwy pliku). Przy pierwszym uruchomieniu programu podałam $N=1000$ i za każdym razem zwiększałam ten parametr o 4000.

N.txt	BubbleSort.txt	SelectionSort.txt	InsertionSort.txt
1 1000	1 0.0033422	1 0.001241	1 0.0006133
2 3000	2 0.0318872	2 0.0118991	2 0.0053863
3 7000	3 0.196895	3 0.0632352	3 0.0312699
4 11000	4 0.514392	4 0.15523	4 0.0766182
5 15000	5 1.00027	5 0.288516	5 0.146478
6 19000	6 1.63231	6 0.461835	6 0.230424
7 23000	7 2.42324	7 0.676806	7 0.33338
8 27000	8 3.37868	8 0.925919	8 0.460673
9 31000	9 4.45542	9 1.22757	9 0.607199
10 35000	10 5.85334	10 1.58965	10 0.808468
11 39000	11 7.30936	11 1.97376	11 1.01063
12 43000	12 9.0528	12 2.46588	12 1.22358
13 47000	13 10.4758	13 2.79878	13 1.39813
14 51000	14 12.4281	14 3.30533	14 1.64149
15 55000	15 14.912	15 4.00378	15 1.92739
16 59000	16 16.7792	16 4.43471	16 2.19537
17 63000	17 19.1344	17 5.03811	17 2.51128
18 67000	18 21.7837	18 5.7733	18 2.83781
19 71000	19 24.5324	19 6.54011	19 3.45615
20 75000	20 28.0436	20 7.29135	20 3.52177
21 79000	21 30.0776	21 7.85043	21 3.90423
22 83000	22 33.1665	22 8.74002	22 4.31181
23 87000	23 36.6519	23 9.63167	23 4.95073
24 91000	24 40.9031	24 10.381	24 5.16169
25 95000	25 44.3881	25 11.9507	25 5.82984
26 99000	26 48.0285	26 12.3579	26 6.13056
27 103000	27 52.0202	27 13.3107	27 6.62503
28 107000	28 56.6259	28 14.7825	28 7.39033
29 111000	29 60.7628	29 15.3604	29 7.73905
30 115000	30 63.4528	30 16.5225	30 8.24211
31	31	31	31
Win UTF-8 INS	Win UTF-8 INS	Win UTF-8 INS	Win UTF-8 INS

Uzyskane w wyniku zapisu do plików pomiary skopiowałam do odpowiednich kolumn w arkuszu Google Sheets.

Wielkość tablicy N	Sortowanie bąbelkowe	Sortowanie przez wybór	Sortowanie przez wstawianie
	czas [s]	czas [s]	czas [s]
1000	0,0033422	0,001241	0,0006133
3000	0,0318872	0,0118991	0,0053863
7000	0,196895	0,0632352	0,0312699
11000	0,514392	0,15523	0,0766182
15000	1,00027	0,288516	0,146478
19000	1,63231	0,461835	0,230424
23000	2,42324	0,676806	0,33338
27000	3,37868	0,925919	0,460673
31000	4,45542	1,22757	0,607199
35000	5,85334	1,58965	0,808468
39000	7,30936	1,97376	1,01063
43000	9,0528	2,46588	1,22358
47000	10,4758	2,79878	1,39813
51000	12,4281	3,30533	1,64149
55000	14,912	4,00378	1,92739
59000	16,7792	4,43471	2,19537
63000	19,1344	5,03811	2,51128
67000	21,7837	5,7733	2,83781
71000	24,5324	6,54011	3,45615
75000	28,0436	7,29135	3,52177
79000	30,0776	7,85043	3,90423

83000	33,1665	8,74002	4,31181
87000	36,6519	9,63167	4,95073
91000	40,9031	10,381	5,16169
95000	44,3881	11,9507	5,82984
99000	48,0285	12,3579	6,13056
103000	52,0202	13,3107	6,62503
107000	56,6259	14,7825	7,39033
111000	60,7628	15,3604	7,73905
115000	63,4528	16,5225	8,24211

Na podstawie uzyskanych pomiarów stworzyłam wykres charakterystyki złożoności wybranych algorytmów (poniżej).

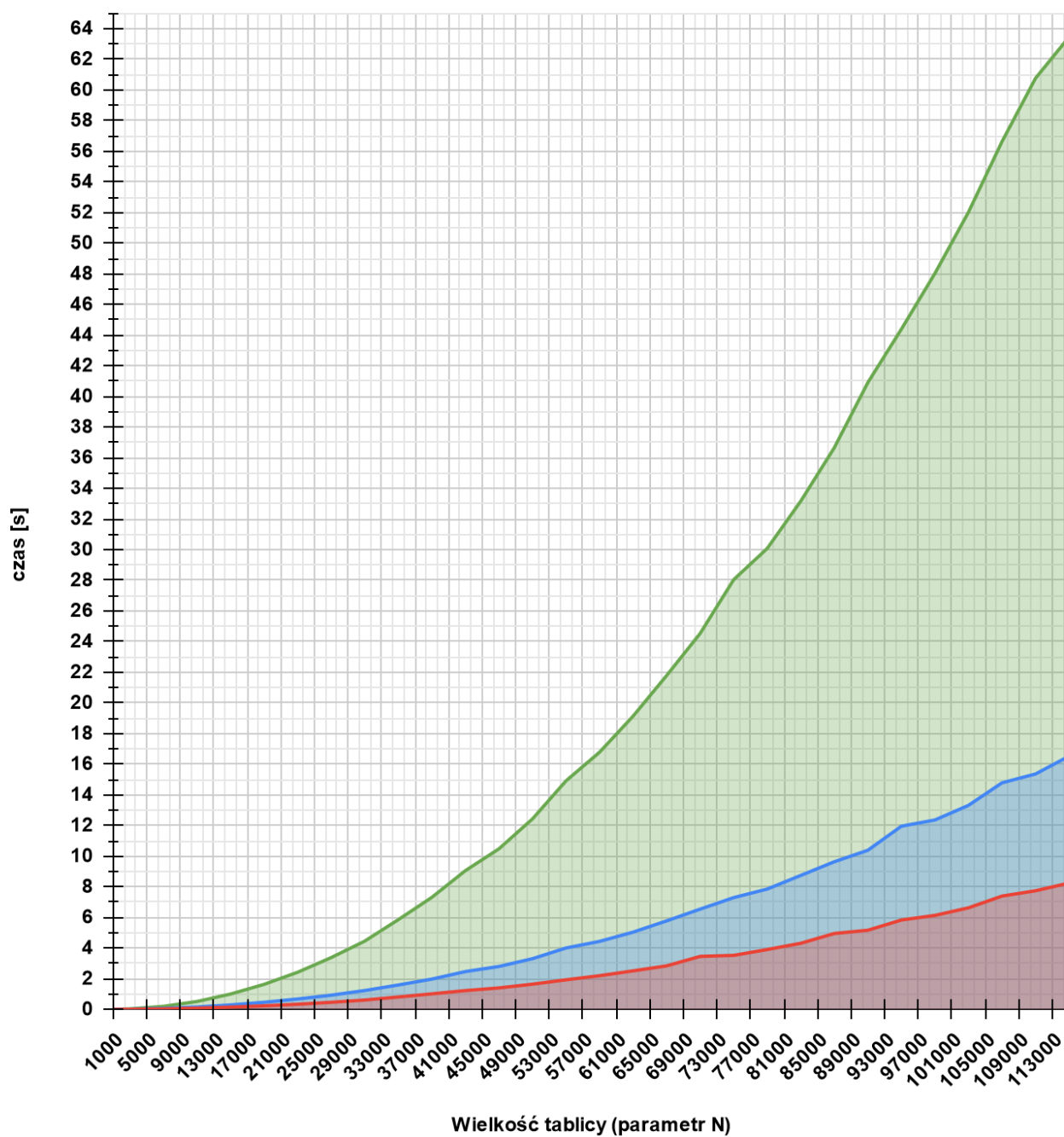
Na osi poziomej oznaczyłam N, czyli długość tablicy - parametr podawany podczas wywoływania programu.

Oś pionowa to czas, w jakim sortowana jest tablica, czyli wygenerowane wyniki w sekundach wyświetlane w oknie konsoli po zadaniu wielkości tablicy.

Charakterystyki złożoności obliczeniowej

dla wybranych rodzajów algorytmów sortowań

Sortowanie bąbelkowe Sortowanie przez wybór Sortowanie przez wstawianie



Tworzenie wykresu dla sortowania przez scalanie, szybkiego i przez zliczanie

- Charakterystyka złożoności dla kolejnych trzech algorytmów

Aby uzyskać większą czytelność drugiego rysunku i zmienność czasu od ilości elementów tablicy, dobrałam inne zakresy osi (dużo większe N). W tym celu opatrzyłam komentarzem poszczególne linijki w funkcji `main()` odnoszące się do obliczeń trzech pierwszych algorytmów aby nie musieć czekać długo na wyniki.

Poniżej prezentuję wyniki zapisywane do plików przy 30 kolejnych uruchomieniach programu. Każdy plik to czas wykonywania innego algorytmu (adekwatnie do nazwy pliku). Przy pierwszym uruchomieniu programu podałam $N=100000$ i za każdym razem zwiększałam ten parametr o 3330000 aż do 96670000.

N2.txt	MergeSort.txt	QuickSort.txt	CountingSort.txt
1 100000	1 0.0174196	1 0.0131041	1 0.0014246
2 3430000	2 0.670284	2 0.45564	2 0.0247848
3 6760000	3 1.36006	3 0.921544	3 0.0478105
4 10090000	4 2.06107	4 1.35401	4 0.0712634
5 13420000	5 2.75984	5 1.83709	5 0.0992885
6 16750000	6 3.48444	6 2.31826	6 0.115347
7 20080000	7 4.15058	7 2.7566	7 0.137915
8 23410000	8 4.93851	8 3.2373	8 0.164419
9 26740000	9 5.63405	9 3.69691	9 0.183064
10 30070000	10 6.37645	10 4.3957	10 0.230542
11 33400000	11 7.0564	11 4.68749	11 0.236585
12 36730000	12 7.77537	12 5.0812	12 0.259991
13 40060000	13 8.61068	13 5.66435	13 0.276228
14 43390000	14 9.16406	14 5.91405	14 0.296995
15 46720000	15 9.79867	15 6.3654	15 0.31819
16 50050000	16 10.53	16 6.87675	16 0.340542
17 53380000	17 11.7007	17 7.42295	17 0.369639
18 56710000	18 11.968	18 7.85447	18 0.393956
19 60040000	19 12.6763	19 8.35744	19 0.418152
20 63370000	20 13.4264	20 8.7485	20 0.438763
21 66700000	21 14.0643	21 9.26447	21 0.458671
22 70030000	22 14.91	22 9.81813	22 0.4825
23 73360000	23 15.5985	23 10.272	23 0.503506
24 76690000	24 16.3587	24 10.7477	24 0.528905
25 80020000	25 17.1322	25 11.2943	25 0.554722
26 83350000	26 17.8561	26 11.6398	26 0.57559
27 86680000	27 18.6491	27 12.092	27 0.599933
28 90010000	28 19.3789	28 13.0354	28 0.622602
29 93340000	29 19.9851	29 13.1163	29 0.646195
30 96670000	30 20.8429	30 13.5528	30 0.665596
31	31	31	31
Win UTF-8 INS	Win UTF-8 INS	Win UTF-8 INS	Win UTF-8 INS

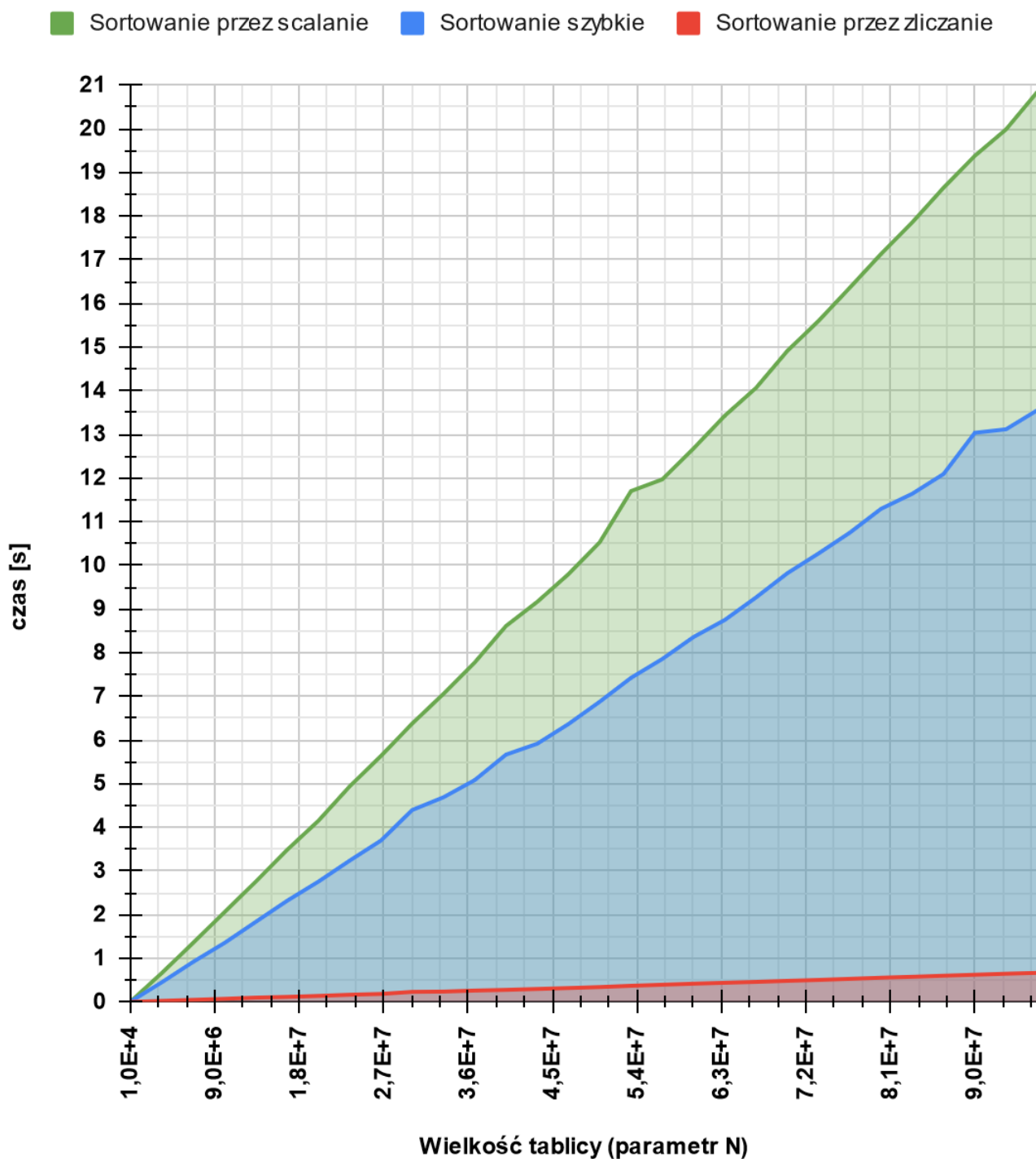
- **Pomiary użyte do wykresu.**

Uzyskane w wyniku zapisu do plików pomiary skopiowałam do odpowiednich kolumn w arkuszu Google Sheets.

Wielkość tablicy N	Sortowanie przez scalanie	Sortowanie szybkie	Sortowanie przez zliczanie
	czas [s]	czas [s]	czas [s]
100000	0,0174196	0,0131041	0,0014246
3430000	0,670284	0,45564	0,0247848
6760000	1,36006	0,921544	0,0478105
10090000	2,06107	1,35401	0,0712634
13420000	2,75984	1,83709	0,0992885
16750000	3,48444	2,31826	0,115347
20080000	4,15058	2,7566	0,137915
23410000	4,93851	3,2373	0,164419
26740000	5,63405	3,69691	0,183064
30070000	6,37645	4,3957	0,230542
33400000	7,0564	4,68749	0,236585
36730000	7,77537	5,0812	0,259991
40060000	8,61068	5,66435	0,276228
43390000	9,16406	5,91405	0,296995
46720000	9,79867	6,3654	0,31819
50050000	10,53	6,87675	0,340542
53380000	11,7007	7,42295	0,369639
56710000	11,968	7,85447	0,393956
60040000	12,6763	8,35744	0,418152
63370000	13,4264	8,7485	0,438763
66700000	14,0643	9,26447	0,458671
70030000	14,91	9,81813	0,4825
73360000	15,5985	10,272	0,503506
76690000	16,3587	10,7477	0,528905
80020000	17,1322	11,2943	0,554722
83350000	17,8561	11,6398	0,57559
86680000	18,6491	12,092	0,599933
90010000	19,3789	13,0354	0,622602
93340000	19,9851	13,1163	0,646195
96670000	20,8429	13,5528	0,665596

Na podstawie uzyskanych pomiarów stworzyłam podobnie jak w poprzednim przypadku wykres charakterystyki złożoności wybranych algorytmów (poniżej).

Charakterystyki złożoności obliczeniowej dla wybranych rodzajów algorytmów sortowań



- **Podsumowanie.**

Najszybszy ze wszystkich okazał się algorytm sortowania przez zliczanie, a najwolniejsze sortowanie bąbelkowe.

Dostępne złożoności czasowe w notacji dużego O dla analizowanych algorytmów potwierdzają moje wyniki:

- Sortowanie bąbelkowe: **$O(n^2)$**
- Sortowanie przez wybór: **$O(n^2)$**
- Sortowanie przez wstawianie: **$O(n^2)$**
- Sortowanie przez scalanie: **$O(n \cdot \log n)$**
- Sortowanie szybkie: **$O(n \cdot \log n)$**
- Sortowanie przez zliczanie (ang. counting sort): **$O(n)$**

Kompletne wnioski po przeanalizowaniu charakterystyk znajdują się na końcu sprawozdania w sekcji [Wnioski](#).

Laboratoria nr 3

Wstęp

Struktury danych to zaawansowane pojemniki na dane, które gromadzą je i układają w odpowiedni sposób, inaczej mówiąc: zarządzają danymi. Na strukturach danych operują algorytmy. Przykładami struktur są:

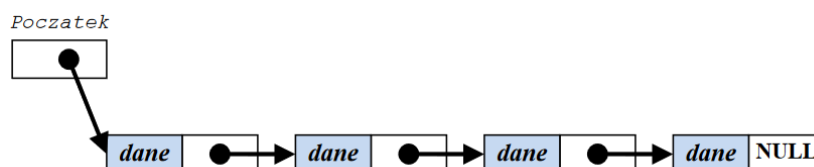
- stos,
- kolejka,
- kopiec,
- drzewo,
- tablica,
- graf
- listy.

Każda struktura danych ma charakterystyczne dla siebie właściwości. Na przykład dodanie elementu na początek tablicy ma złożoność obliczeniową $O(n)$. Ta sama operacja dla listy związanej ma złożoność $O(1)$. Te właściwości sprawiają, że użycie konkretnej struktury może uprościć rozwiązanie niektórych problemów. Możemy powiedzieć, że czasami lepiej jest użyć tablicy a w innym przypadku lista wiązana jest lepszym rozwiązaniem. Wszystko zależy od problemu, który próbujemy rozwiązać. Strukturę danych dopasowuje się do problemu. Lista jest strukturą danych służącą do przechowywania nieznanej z góry ilości informacji tego samego typu. Składa się z węzłów (ang. nodes), które zawierają dane przechowywane w liście oraz wskaźnik do kolejnego lub dodatkowo także do poprzedniego elementu.

• Lista jednokierunkowa

Lista jednokierunkowa jest strukturą pozwalającą na zapamiętanie danych w postaci uporządkowanej, a także na bardzo szybkie wstawianie i usuwanie elementów do i z listy. Pamiętana jest w postaci „pojemników” zawierających porcję danych oraz wskaźnik (adres) następnego „pojemnika”. W ten sposób wystarczy pamiętać wskaźnik do pierwszego elementu listy, by pamiętać całą listę.

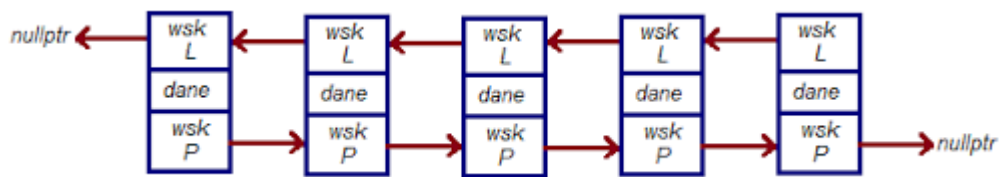
Aby zaimplementować listę jednokierunkową w języku C/C++ trzeba zdefiniować strukturę pełniącą rolę węzłów stanowiących kolejne elementy listy. Taka struktura składa się z dwóch części: pola (lub kilku pól), w którym pamiętane są przechowywane elementy (dane użytkowe) oraz pola wskaźnikowego, w którym będzie pamiętany wskazanie do następnego węzła. Dostęp do całej listy wymaga utworzenia zmiennej wskaźnikowej, zawierającej wskazanie na pierwszy węzeł listy lub wartość NULL jeśli lista jest pusta (tzn. nie zawiera żadnych węzłów).



Rys. Lista jednokierunkowa zawierająca 4 elementy

- **Lista dwukierunkowa**

Dwukierunkowa lista wiązana od listy jednokierunkowej różni się tym, że każdy z węzłów zawiera wskaźnik na poprzedni i następny element. Sama lista zawiera też atrybuty wskazujące pierwszy i ostatni węzeł w liście. Implementacja tej metody polega na przechodzeniu po wszystkich elementach od początku do żadanego indeksu. W przypadku listy dwukierunkowej węzły zawierają dwa wskaźniki. Operacje modyfikujące taką listę wymagają przebiecia każdego z tych wskaźników.



Rys. Lista dwukierunkowa zawierająca

Takie połączenie elementów umożliwia przeglądanie listy w obu kierunkach oraz daje możliwość dopisywania kolejnego elementu w dowolnym miejscu listy. Należy pamiętać aby pierwszy element listy pokazywał z lewej strony kolejki na wskaźnik pusty oraz ostatni element kolejki wskazywał z prawej strony na wskaźnik pusty. Takie rozwiązanie jest zabezpieczeniem (wartownikiem) przed przekroczeniem zakresu przeglądania listy z lewej i prawej strony.

Cel laboratorium nr 3

Celem laboratorium jest sprawdzenie czasu pracy oraz złożoności obliczeniowej operacji na liście jednokierunkowej i dwukierunkowej (m.in. dodawanie, usuwanie, pobieranie elementów do list). Na podstawie wyników czasów generowania list należy zbudować charakterystyki złożoności obliczeniowej.

Wszystkie pliki projektu znajdują się w publicznym repozytorium zdalnym Gita na moim koncie na Githubie. Link do niego:

https://github.com/Yaviena/Algorithms_Lab_3_part_1_Magda_Szafranska

Wykonanie ćwiczenia cz. I: lista jednokierunkowa

Implementacja listy jednokierunkowej

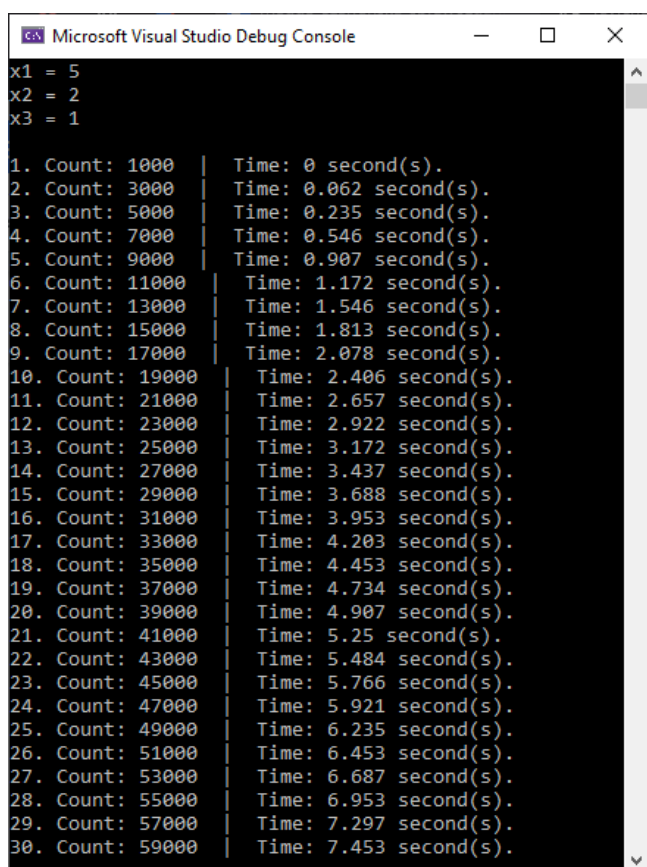
Zaimplementowałam kod obsługujący działanie listy jednokierunkowej. Zawiera on podstawowe funkcje operacji na listach takie jak: dodawanie elementu do listy, usuwanie z niej czy przeszukiwanie. Istotną rolę pełnią wskaźniki na pierwszy i ostatni element listy.

Cały kod programu znajduje się na końcu tego laboratorium. Dostępny jest także w repozytorium zdalnym (link powyżej).

- **Funkcje obliczające czasy wykonywania.**

Do pomiaru czasów wykonywania użyłam podobnie jak przy laboratorium nr 1 funkcji `GetTickCount64()`. Zwraca ona typ całkowity (`int`) jako liczbę milisekund, która upłynęła pomiędzy dwoma wydarzeniami. Aby uzyskać lepszą czytelność wynik pomnożyłam przez 0.001 pokazując go tym samym w sekundach.

Bezpośrednio przed wywołaniem w funkcji `main()` kluczowej pętli `“while”` oraz bezpośrednio po pętli wywoływałam funkcję `GetTickCount64()` na zmiennych lokalnych i zapisywałam w nich za każdym razem bieżące wyniki wypełniania listy losowymi wartościami. Proces ten powtarzał się przez 30 iteracji zwiększając za każdym razem wielkość licznika o 2000 (od 1000 do 59000). Poniżej zrzut ekranu z uzyskanymi pomiarami.



```
Microsoft Visual Studio Debug Console
x1 = 5
x2 = 2
x3 = 1

1. Count: 1000 | Time: 0 second(s).
2. Count: 3000 | Time: 0.062 second(s).
3. Count: 5000 | Time: 0.235 second(s).
4. Count: 7000 | Time: 0.546 second(s).
5. Count: 9000 | Time: 0.907 second(s).
6. Count: 11000 | Time: 1.172 second(s).
7. Count: 13000 | Time: 1.546 second(s).
8. Count: 15000 | Time: 1.813 second(s).
9. Count: 17000 | Time: 2.078 second(s).
10. Count: 19000 | Time: 2.406 second(s).
11. Count: 21000 | Time: 2.657 second(s).
12. Count: 23000 | Time: 2.922 second(s).
13. Count: 25000 | Time: 3.172 second(s).
14. Count: 27000 | Time: 3.437 second(s).
15. Count: 29000 | Time: 3.688 second(s).
16. Count: 31000 | Time: 3.953 second(s).
17. Count: 33000 | Time: 4.203 second(s).
18. Count: 35000 | Time: 4.453 second(s).
19. Count: 37000 | Time: 4.734 second(s).
20. Count: 39000 | Time: 4.907 second(s).
21. Count: 41000 | Time: 5.25 second(s).
22. Count: 43000 | Time: 5.484 second(s).
23. Count: 45000 | Time: 5.766 second(s).
24. Count: 47000 | Time: 5.921 second(s).
25. Count: 49000 | Time: 6.235 second(s).
26. Count: 51000 | Time: 6.453 second(s).
27. Count: 53000 | Time: 6.687 second(s).
28. Count: 55000 | Time: 6.953 second(s).
29. Count: 57000 | Time: 7.297 second(s).
30. Count: 59000 | Time: 7.453 second(s).
```

- **Implementacja zapisu pomiarów do plików**

Aby zebrać dane potrzebne do sporządzania wykresów, dodałam funkcje zapisu do plików z biblioteki *fstream*. Dla wygody późniejszego kopiowania danych do arkusza Google Sheets, pomiary czasów oraz adekwatne dla nich wartości zmiennej `“Count”` zapisałam w osobnym pliku (z parametrem nadpisywania). Poniżej screen zapisów z pliku.

count.txt	time.txt
1 1000	1 0
2 3000	2 0.062
3 5000	3 0.235
4 7000	4 0.546
5 9000	5 0.907
6 11000	6 1.172
7 13000	7 1.546
8 15000	8 1.813
9 17000	9 2.078
10 19000	10 2.406
11 21000	11 2.657
12 23000	12 2.922
13 25000	13 3.172
14 27000	14 3.437
15 29000	15 3.688
16 31000	16 3.953
17 33000	17 4.203
18 35000	18 4.453
19 37000	19 4.734
20 39000	20 4.907
21 41000	21 5.25
22 43000	22 5.484
23 45000	23 5.766
24 47000	24 5.921
25 49000	25 6.235
26 51000	26 6.453
27 53000	27 6.687
28 55000	28 6.953
29 57000	29 7.297
30 59000	30 7.453

- **Dane do wykresu charakterystyki złożoności.**

Zapisane w plikach tekstowych pomiary skopiowałam do dokumentu Google Sheets. Posłużyły mi one do wygenerowania charakterystyki złożoności obliczeniowej.

Count	Time [s]
1000	0
3000	0.062
5000	0.235
7000	0.546
9000	0.907
11000	1.172
13000	1.546
15000	1.813
17000	2.078
19000	2.406
21000	2.657
23000	2.922
25000	3.172
27000	3.437
29000	3.688
31000	3.953

33000	4.203
35000	4.453
37000	4.734
39000	4.907
41000	5.25
43000	5.484
45000	5.766
47000	5.921
49000	6.235
51000	6.453
53000	6.687
55000	6.953
57000	7.297
59000	7.453

- **Dopasowanie skali osi.**

Maksymalna wartość na osi czasu to 8 sekund. Liczba dodawanych elementów do list (Count) waha się w moich pomiarach od 1000 do 59000 (z krokiem co 2000), które są jednocześnie moimi skrajnymi wartościami. Oś czasu podzieliłam adekwatnie do powyższego proporcjonalnie według skali liniowej na odcinki czasowe co 0.3 s.

- **Wykres charakterystyki złożoności dla listy jednokierunkowej**

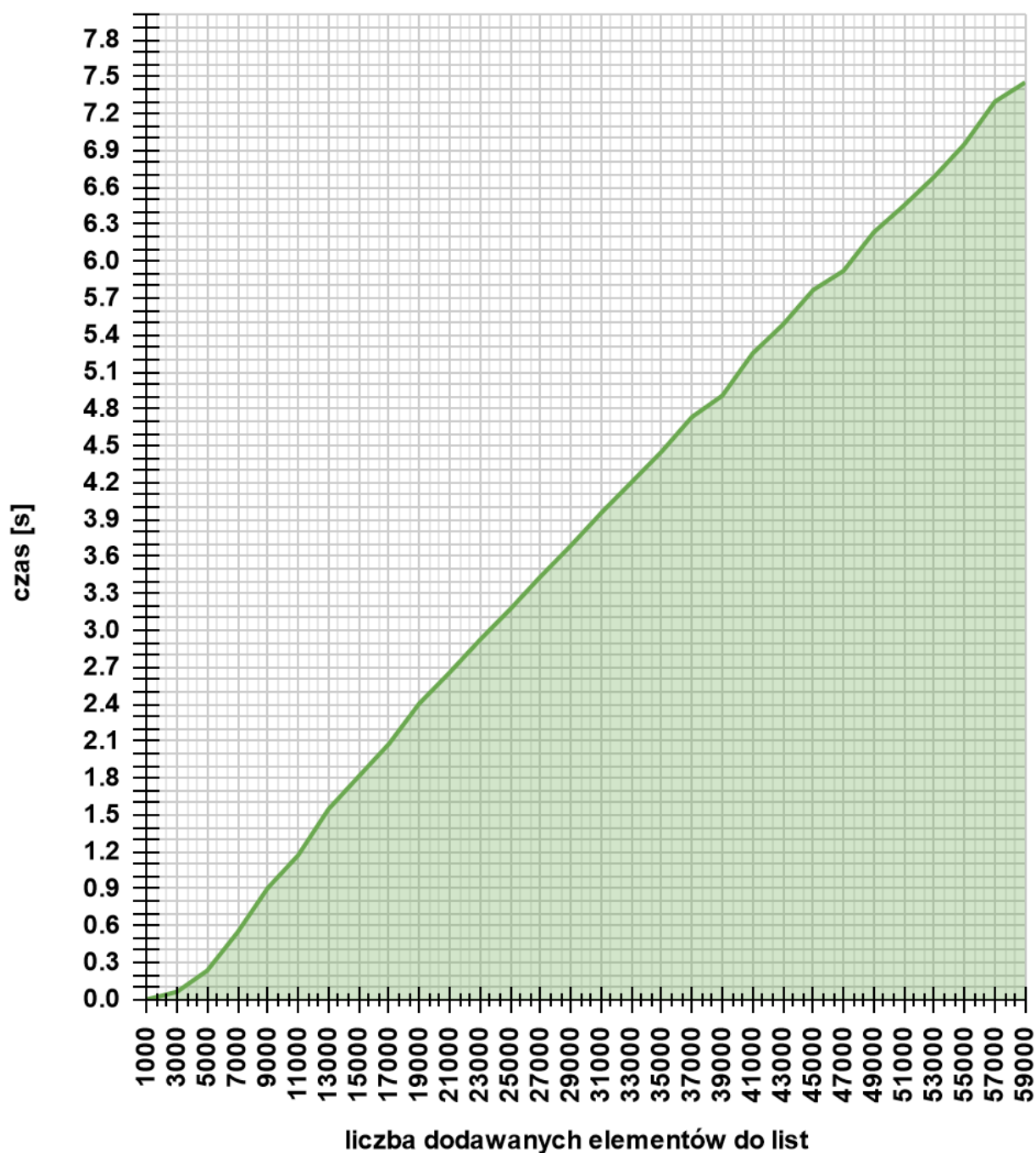
Na podstawie uzyskanych pomiarów stworzyłam wykres charakterystyki złożoności czasowej dodawania niepowtarzających się wartości do listy.

Na osi poziomej oznaczyłam liczbę dodawanych elementów do list. Oś pionowa to czas, w jakim są one dodawane.

Charakterystyka złożoności obliczeniowej

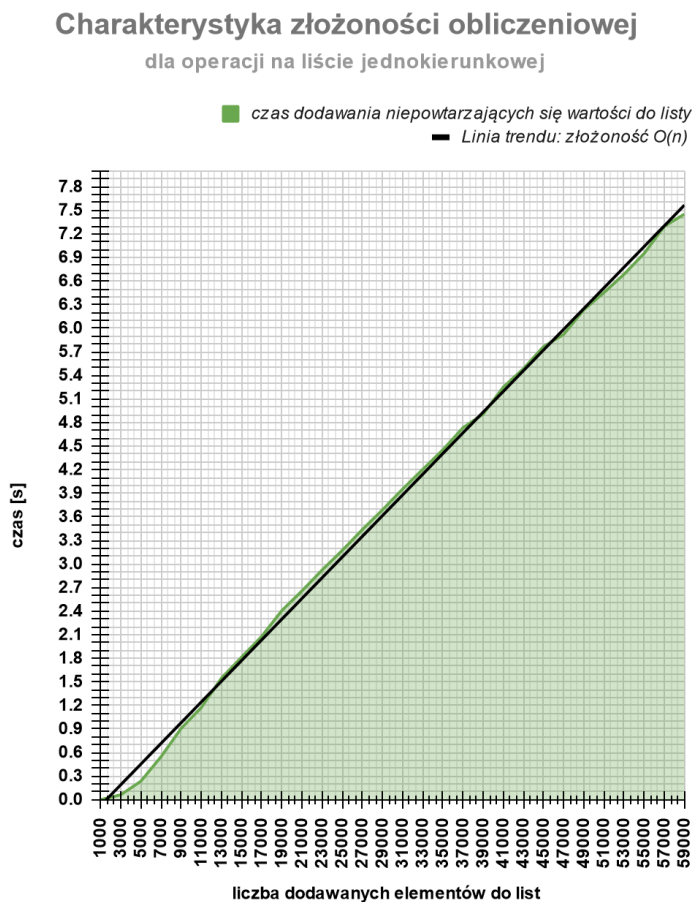
dla operacji na liście jednokierunkowej

■ czas dodawania niepowtarzających się wartości do listy



- **Podsumowanie**

Nałożyłam na wykres dodatkowo linię trendu (rysunek poniżej).



Zarówno z dokumentacji niniejszego sprawozdania jak i z teorii złożoności obliczeniowej wynika, że zbadanie, czy element znajduje się na liście wykonuje się w czasie liniowym. Na linii trendu wyraźnie widać, że złożoność jest rzędu liniowego czyli $O(n)$.

- **Kod programu.**

Poniżej znajduje się cały kod programu. Jest do pobrania również w zdalnym repozytorium razem z całym projektem. Link do niego znajduje się na początku sekcji Laboratorium nr 3.

```
// Autor: Magda Szafranska, nr indeksu AHNS: 18345
// Informatyka NST, rok 2, sem. 3
// Algorytmy, laboratoria nr 3 - part I

#include <iostream>
#include <Windows.h>
#include <fstream>

using namespace std;

class TItem
{
```

```

public: int FData;
       TItem* FNext;

public:
    TItem(TItem* ANext, int AData)
    {
        FData = AData;
        FNext = ANext;
    }
    ~TItem()
    {
    }
};

class TList
{
private:
    TItem* FFirst;

public:
    TList(void);
    ~TList(void);

    int Pop(void);
    void Push(int AData);
    bool IsExist(int AData);
};

TList::TList(void)
{
    FFirst = NULL;
}
TList::~~TList(void)
{
    while (FFirst) Pop();
}
void TList::Push(int AData)
{
    FFirst = new TItem(FFirst, AData);
}
int TList::Pop(void)
{
    int AData = FFirst -> FData;
    TItem* AItem = FFirst;
    FFirst = FFirst -> FNext;
    delete AItem;
    return AData;
}
bool TList::IsExist(int AData)
{
    TItem* Item = FFirst;
    while (Item)
        if (Item -> FData == AData) return true;
        else Item = Item -> FNext;
    return false;
}

```

```

int main()
{
    TList list;
    list.Push(1);
    list.Push(2);
    list.Push(5);

    int x1 = list.Pop();
    int x2 = list.Pop();
    int x3 = list.Pop();

    cout << "x1 = " << x1 << endl;
    cout << "x2 = " << x2 << endl;
    cout << "x3 = " << x3 << endl << endl;

    // writing to the text file
    fstream fTime, fCount;
    fTime.open("time.txt", ios::out | ios::app);
    fCount.open("count.txt", ios::out | ios::app);

    int T1, T2;
    int Count = 1000;
    int b = 2000;

    for (int j = 1; j <= 30; j++)
    {
        fCount << Count << endl;
        cout << j << ". Count: " << Count;

        T1 = (int)GetTickCount64();
        while (--Count)
        {
            int Value = rand();
            if (!list.IsExist(Value))
                list.Push(Value);
        }
        T2 = (int)GetTickCount64();

        cout << " | Time: " << (T2 - T1) * 0.001 << " second(s)." << endl;
        fTime << (T2 - T1) * 0.001 << endl;
        Count = 1000 + b;
        b += 2000;
    }

    fTime.close();
    fCount.close();
}

fTime.close();
fCount.close();
}

```


Wykonanie ćwiczenia cz. II: lista dwukierunkowa

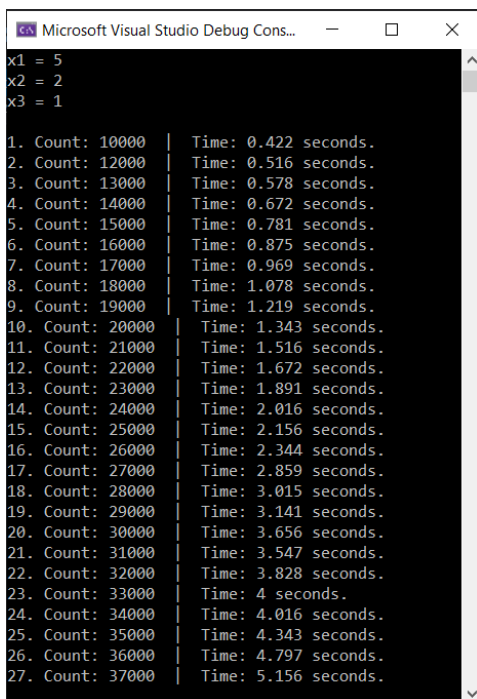
W drugiej części zajęć zajęliśmy się algorytmem dodającym elementy do listy dwukierunkowej i sumującym je. Celem było sprawdzenie jak szybko elementy te są iterowane, czas pracy algorytmu oraz jego złożoność czasowa.

Następnie należało wprowadzić pewne poprawki, aby poprawić tę złożoność.

Algorytm sumujący elementy na liście - wersja podstawowa

- **Obliczenie czasu wykonywania.**

Do pomiarów i stworzenia charakterystyki przyjąłm za punkt startowy 10000 elementów tworzonych i sumowanych na liście. Następnie za każdym kolejnym razem zwiększałam ich ilość o 1000 mierząc czas. Poniżej screen z liczbą tworzonych na liście i sumowanych elementów oraz czasem wykonywania tych operacji na listach.



```
Microsoft Visual Studio Debug Cons...
x1 = 5
x2 = 2
x3 = 1

1. Count: 10000 | Time: 0.422 seconds.
2. Count: 12000 | Time: 0.516 seconds.
3. Count: 13000 | Time: 0.578 seconds.
4. Count: 14000 | Time: 0.672 seconds.
5. Count: 15000 | Time: 0.781 seconds.
6. Count: 16000 | Time: 0.875 seconds.
7. Count: 17000 | Time: 0.969 seconds.
8. Count: 18000 | Time: 1.078 seconds.
9. Count: 19000 | Time: 1.219 seconds.
10. Count: 20000 | Time: 1.343 seconds.
11. Count: 21000 | Time: 1.516 seconds.
12. Count: 22000 | Time: 1.672 seconds.
13. Count: 23000 | Time: 1.891 seconds.
14. Count: 24000 | Time: 2.016 seconds.
15. Count: 25000 | Time: 2.156 seconds.
16. Count: 26000 | Time: 2.344 seconds.
17. Count: 27000 | Time: 2.859 seconds.
18. Count: 28000 | Time: 3.015 seconds.
19. Count: 29000 | Time: 3.141 seconds.
20. Count: 30000 | Time: 3.656 seconds.
21. Count: 31000 | Time: 3.547 seconds.
22. Count: 32000 | Time: 3.828 seconds.
23. Count: 33000 | Time: 4 seconds.
24. Count: 34000 | Time: 4.016 seconds.
25. Count: 35000 | Time: 4.343 seconds.
26. Count: 36000 | Time: 4.797 seconds.
27. Count: 37000 | Time: 5.156 seconds.
```

- **Implementacja zapisu pomiarów do plików**

Tak jak uprzednio, zapisałam do plików pomiary czasów i adekwatne dla nich wartości zmiennej "Count".

time_1.txt	count_1.txt
1 0.422	1 10000
2 0.516	2 12000
3 0.578	3 13000
4 0.672	4 14000
5 0.781	5 15000
6 0.875	6 16000
7 0.969	7 17000
8 1.078	8 18000
9 1.219	9 19000
10 1.343	10 20000
11 1.516	11 21000
12 1.672	12 22000
13 1.891	13 23000
14 2.016	14 24000
15 2.156	15 25000
16 2.344	16 26000
17 2.859	17 27000
18 3.015	18 28000
19 3.141	19 29000
20 3.656	20 30000
21 3.547	21 31000
22 3.828	22 32000
23 4	23 33000
24 4.016	24 34000
25 4.343	25 35000
26 4.797	26 36000
27 5.156	27 37000

- **Dane do wykresu charakterystyki złożoności.**

Zapisane w plikach tekstowych pomiary skopiowałam do dokumentu Google Sheets. Posłużyły mi one do wygenerowania charakterystyki złożoności obliczeniowej.

Count	Time [s] - before
10000	0.422
12000	0.516
13000	0.578
14000	0.672
15000	0.781
16000	0.875
17000	0.969
18000	1.078
19000	1.219
20000	1.343
21000	1.516
22000	1.672
23000	1.891
24000	2.016
25000	2.156
26000	2.344
27000	2.859
28000	3.015
29000	3.141

30000	3.656
31000	3.547
32000	3.828
33000	4
34000	4.016
35000	4.343
36000	4.797
37000	5.156

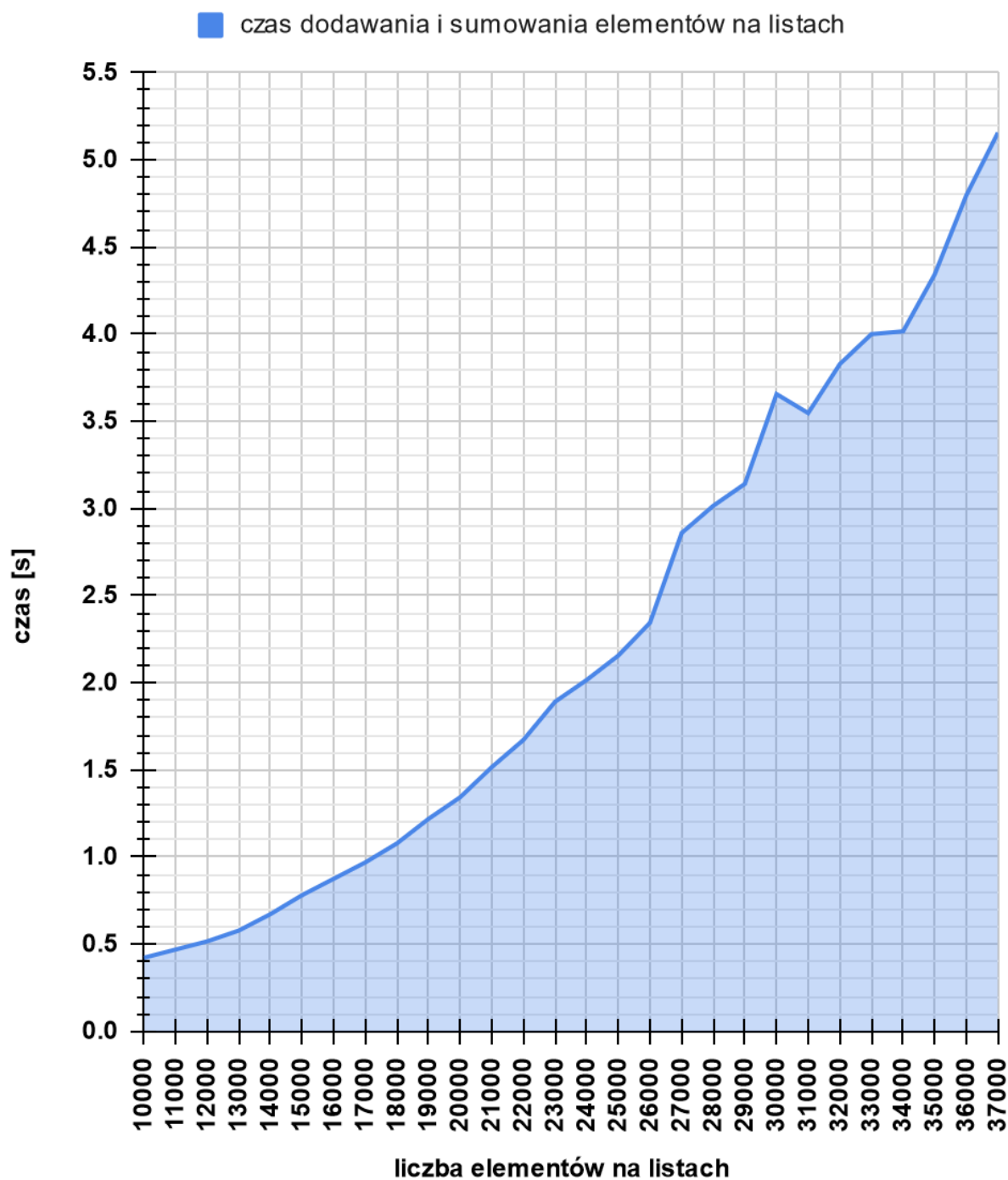
- **Dopasowanie skali osi.**

Maksymalna wartość na osi czasu to 5.5 sekund. Podzieliłam ją proporcjonalnie według skali liniowej na odcinki czasowe co 0.5 sekundy. Liczba tworzonych i sumowanych elementów na listach waha się w moich pomiarach od 10000 do 37000 (z krokiem co 1000), które są jednocześnie moimi skrajnymi wartościami. Oś czasu podzieliłam więc co 1000 elementów.

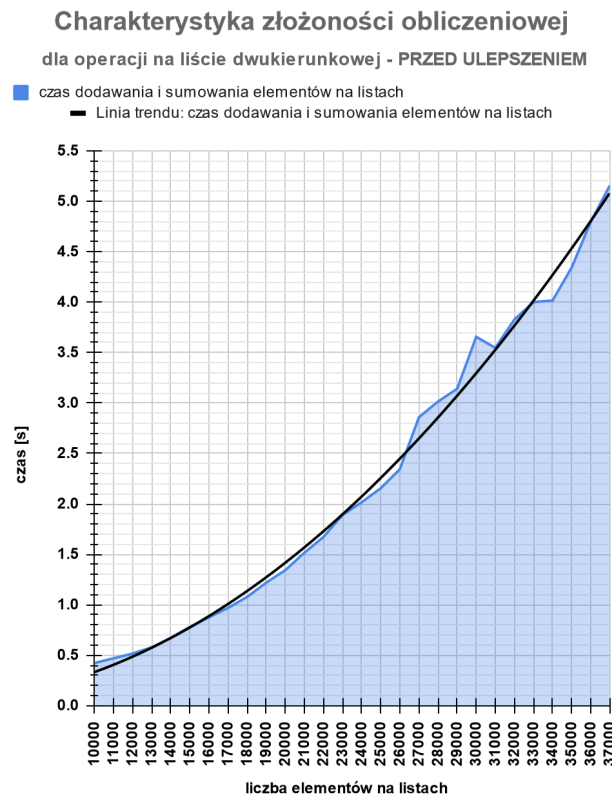
- **Wykres charakterystyki złożoności dla listy dwukierunkowej**

Na podstawie uzyskanych pomiarów stworzyłam wykres charakterystyki złożoności czasowej dodawania i sumowania elementów na liście dwukierunkowej. Na osi poziomej oznaczyłam liczbę dodawanych (a później też sumowanych) elementów do list. Oś pionowa to czas, w jakim się to dzieje dla poszczególnych list różniących się większą liczbą elementów od poprzedniej listy.

Charakterystyka złożoności obliczeniowej dla operacji na liście dwukierunkowej - PRZED ULEPSZENIEM



Po nałożeniu na wykres dodatkowej linii trendu (rysunek poniżej) wyraźnie widać, że złożoność jest wielomianowa czyli $O(n^x)$, gdzie n to liczba danych wejściowych a x jest dowolną stałą.

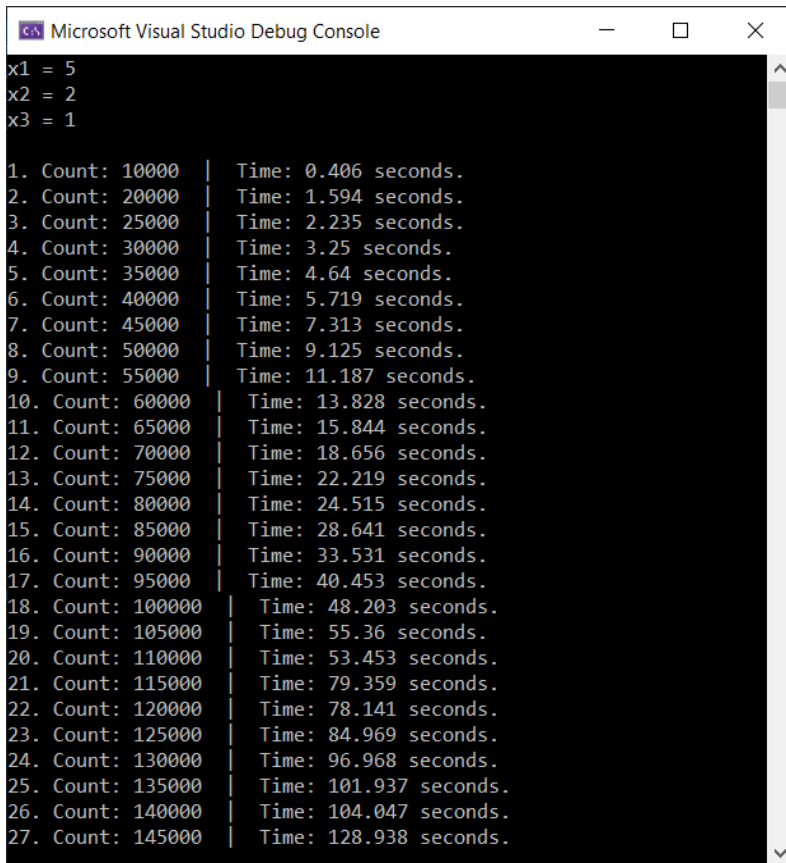


- **Kod programu.**

Kompletny kod programu znajduje się na końcu sekcji dotyczącej niniejszego laboratorium. Jest do pobrania również w zdalnym repozytorium razem z całym projektem (link na początku sekcji Laboratorium nr 3).

Algorytm sumujący elementy na liście - wersja ulepszona

Poprzedni algorytm (bez ulepszenia) już dla 145000 elementów miał bardzo duży czas wykonywania. Screen z wynikami zamieszczam poniżej.



```
Microsoft Visual Studio Debug Console
x1 = 5
x2 = 2
x3 = 1

1. Count: 10000 | Time: 0.406 seconds.
2. Count: 20000 | Time: 1.594 seconds.
3. Count: 25000 | Time: 2.235 seconds.
4. Count: 30000 | Time: 3.25 seconds.
5. Count: 35000 | Time: 4.64 seconds.
6. Count: 40000 | Time: 5.719 seconds.
7. Count: 45000 | Time: 7.313 seconds.
8. Count: 50000 | Time: 9.125 seconds.
9. Count: 55000 | Time: 11.187 seconds.
10. Count: 60000 | Time: 13.828 seconds.
11. Count: 65000 | Time: 15.844 seconds.
12. Count: 70000 | Time: 18.656 seconds.
13. Count: 75000 | Time: 22.219 seconds.
14. Count: 80000 | Time: 24.515 seconds.
15. Count: 85000 | Time: 28.641 seconds.
16. Count: 90000 | Time: 33.531 seconds.
17. Count: 95000 | Time: 40.453 seconds.
18. Count: 100000 | Time: 48.203 seconds.
19. Count: 105000 | Time: 55.36 seconds.
20. Count: 110000 | Time: 53.453 seconds.
21. Count: 115000 | Time: 79.359 seconds.
22. Count: 120000 | Time: 78.141 seconds.
23. Count: 125000 | Time: 84.969 seconds.
24. Count: 130000 | Time: 96.968 seconds.
25. Count: 135000 | Time: 101.937 seconds.
26. Count: 140000 | Time: 104.047 seconds.
27. Count: 145000 | Time: 128.938 seconds.
```

Aby poprawić jego wydajność, w kilku miejscach wstawiłam dodatkowy fragment kodu.

Dodałam dwie zmienne:

- *int FIndex*; - przechowuje index (pozycję) przetwarzanego obecnie elementu na liście
- *TItem* FCurr*; - to skrót od Current, czyli obecny, ta zmienna przechowuje zatem w pamięci aktualnie przetwarzany element.

```
int FIndex;
TItem* FCurr;
```

Zmodyfikowałam także funkcję `Get()`. W ulepszonej wersji operuję na indexach listy a nie bezpośrednio na obiekcie.

Poniżej porównanie wersji przed oraz po modyfikacji.

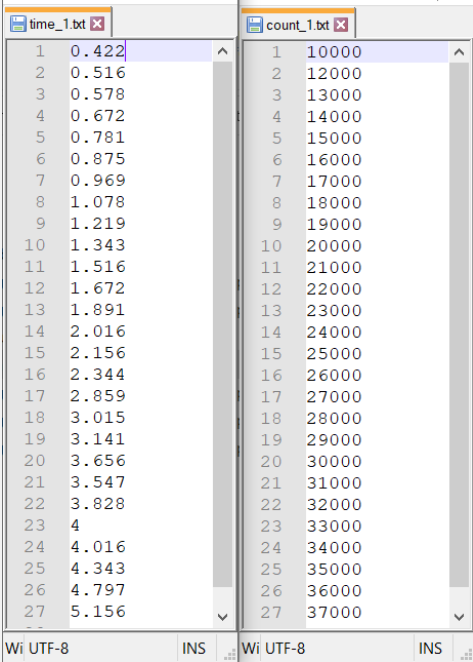
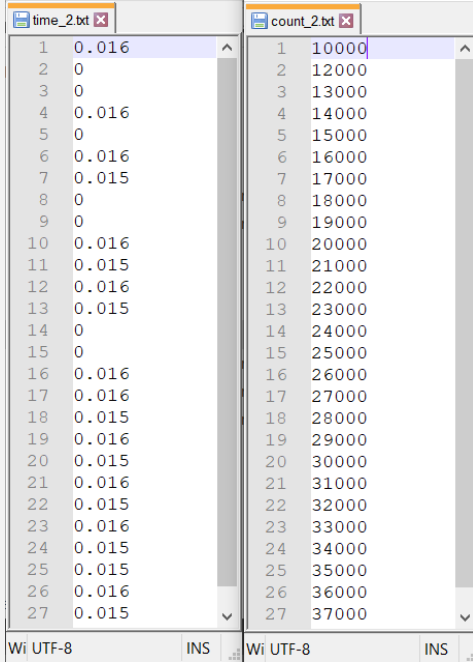
Działanie pętli *while* po modyfikacji:

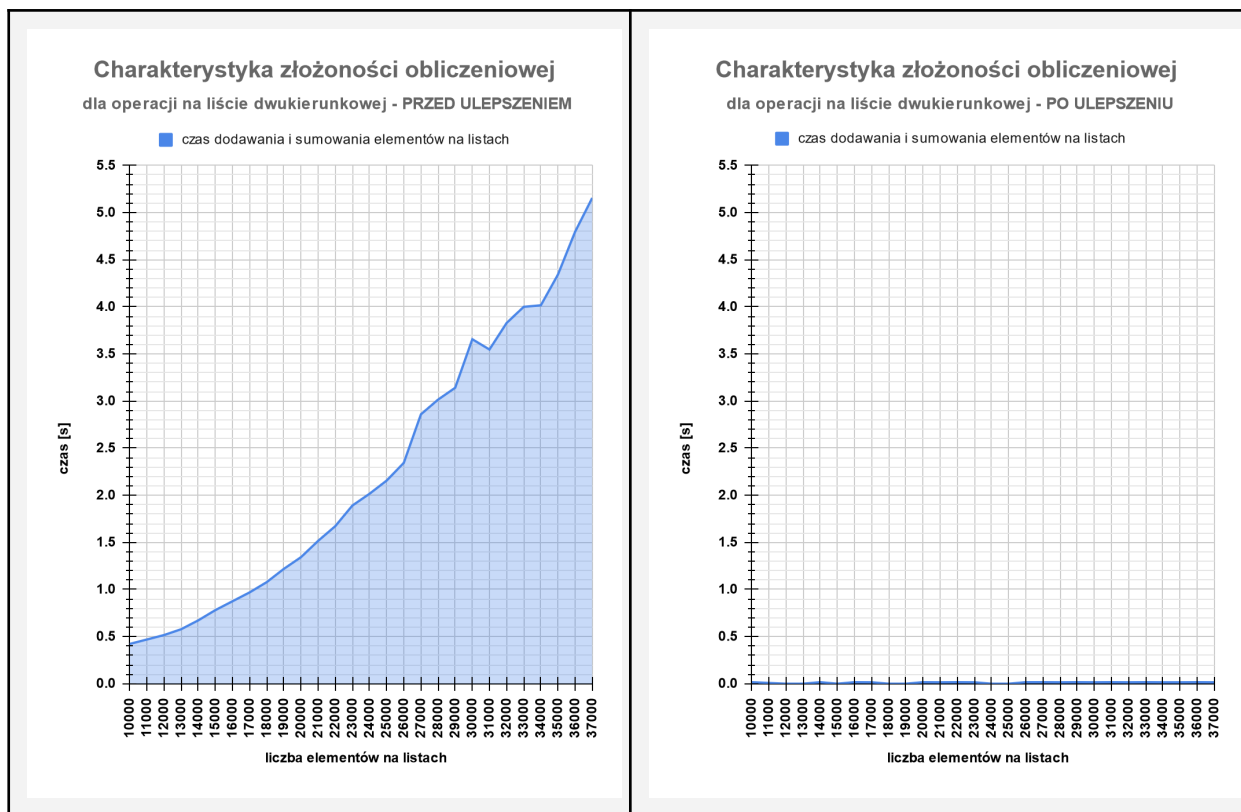
- jeżeli indeks < 0 to pierwszy indeks ustawiany jest na element nr 0 (czyli pierwszy element)
- jeżeli w pętli wybrany indeks (przekazany w parametrze funkcji *AIndex*) jest mniejszy od indexu aktualnie przetwarzanego elementu, to zmienia aktualnie przetwarzany element na poprzedni (*FPrev*)
- jeżeli w pętli wybrany indeks (przekazany w parametrze funkcji *AIndex*) jest większy od indexu aktualnie przetwarzanego elementu, to zmienia aktualnie przetwarzany element na następny (*FNext*).

PRZED ulepszeniem	PO ulepszeniu
<pre> 74 TItem* Get(int Index) 75 { 76 TItem* Item = FFirst; 77 78 while (Item && Index--> 79 Item = Item->FNext; 80 81 return Item; 82 }</pre>	<pre> 80 TItem* Get(int AIndex) 81 { 82 if (FIndex < 0) 83 { 84 FIndex = 0; 85 FCurr = FFirst; 86 } 87 while (AIndex < FIndex) 88 { 89 FCurr = FCurr->FPrev; 90 FIndex--; 91 } 92 while (AIndex > FIndex) 93 { 94 FCurr = FCurr->FNext; 95 FIndex++; 96 } 97 return FCurr; 98 }</pre>

PRZED ulepszeniem pętla przetwarzała każdy z obiektów pojedynczo. Czas jej wykonywania był tak długi, ponieważ każdy obiekt był bezpośrednio zaczytywany do pamięci. W ulepszonej wersji pętla przetwarza jedynie indeksy obiektów.

Po poprawieniu algorytmu, przy zachowaniu tych samych parametrów co uprzednio, jego złożoność zauważalnie zmalała. Pomiary ulepszonej wersji również zapisałam do plików (*time_2.txt* oraz *count_2.txt*), innych niż przed ulepszeniem - aby móc je następnie porównać niezależnie. Wyniki poniżej - przedstawione na identycznych parametrach.

PRZED ulepszeniem	PO ulepszeniu
	



- **Wersja ulepszona PRO**

Zastosowanie tych samych parametrów spowodowało nieczytelność wykresu po ulepszeniu go fragmentami kodu. Algorytm po ulepszeniu jest o wiele efektywniejszy więc wprowadziłam o wiele większe ilości elementów dodawanych i sumowanych na listach. Dla lepszego zobrazowania tego, jak bardzo poprawiła się wydajność algorytmu po “ulepszeniu”, wykonałam jeszcze jedną kompilację, tym razem podając na wejściu większe wartości.

```

Microsoft Visual Studio Debug Console
x1 = 5
x2 = 2
x3 = 1

1. Count: 1000 | Time: 0 seconds.
2. Count: 1001000 | Time: 0.406 seconds.
3. Count: 1501000 | Time: 0.625 seconds.
4. Count: 2001000 | Time: 1.109 seconds.
5. Count: 2501000 | Time: 0.922 seconds.
6. Count: 3001000 | Time: 1.078 seconds.
7. Count: 3501000 | Time: 1.265 seconds.
8. Count: 4001000 | Time: 1.453 seconds.
9. Count: 4501000 | Time: 1.672 seconds.
10. Count: 5001000 | Time: 1.906 seconds.
11. Count: 5501000 | Time: 2.218 seconds.
12. Count: 6001000 | Time: 2.688 seconds.
13. Count: 6501000 | Time: 2.625 seconds.
14. Count: 7001000 | Time: 2.656 seconds.
15. Count: 7501000 | Time: 2.765 seconds.
16. Count: 8001000 | Time: 3.031 seconds.
17. Count: 8501000 | Time: 3.485 seconds.
18. Count: 9001000 | Time: 3.719 seconds.
19. Count: 9501000 | Time: 3.718 seconds.
20. Count: 10001000 | Time: 4.203 seconds.
21. Count: 10501000 | Time: 4 seconds.
22. Count: 11001000 | Time: 4.094 seconds.
23. Count: 11501000 | Time: 4.297 seconds.
24. Count: 12001000 | Time: 4.453 seconds.
25. Count: 12501000 | Time: 4.828 seconds.
26. Count: 13001000 | Time: 4.828 seconds.
27. Count: 13501000 | Time: 5.359 seconds.

```


- **Implementacja zapisu pomiarów do plików**

Zapisałam pomiary do plików z dopiskiem “pro”.

Count	Time 2 [s] - PRO
1000	0
1001000	0.406
1501000	0.625
2001000	1.109
2501000	0.922
3001000	1.078
3501000	1.265
4001000	1.453
4501000	1.672
5001000	1.906
5501000	2.218
6001000	2.688
6501000	2.625
7001000	2.656
7501000	2.765
8001000	3.031
8501000	3.485
9001000	3.719
9501000	3.718
10001000	4.203
10501000	4
11001000	4.094
11501000	4.297
12001000	4.453
12501000	4.828
13001000	4.828
13501000	5.359

- **Dane do wykresu charakterystyki złożoności.**

Zapisane w plikach tekstowych pomiary (z dopiskiem “pro”) skopiowałam do dokumentu Google Sheets. Posłużyły mi one do wygenerowania charakterystyki złożoności obliczeniowej.

Count	Time 2 [s] - PRO
1000	0
1001000	0.406
1501000	0.625
2001000	1.109
2501000	0.922
3001000	1.078
3501000	1.265
4001000	1.453
4501000	1.672
5001000	1.906
5501000	2.218
6001000	2.688

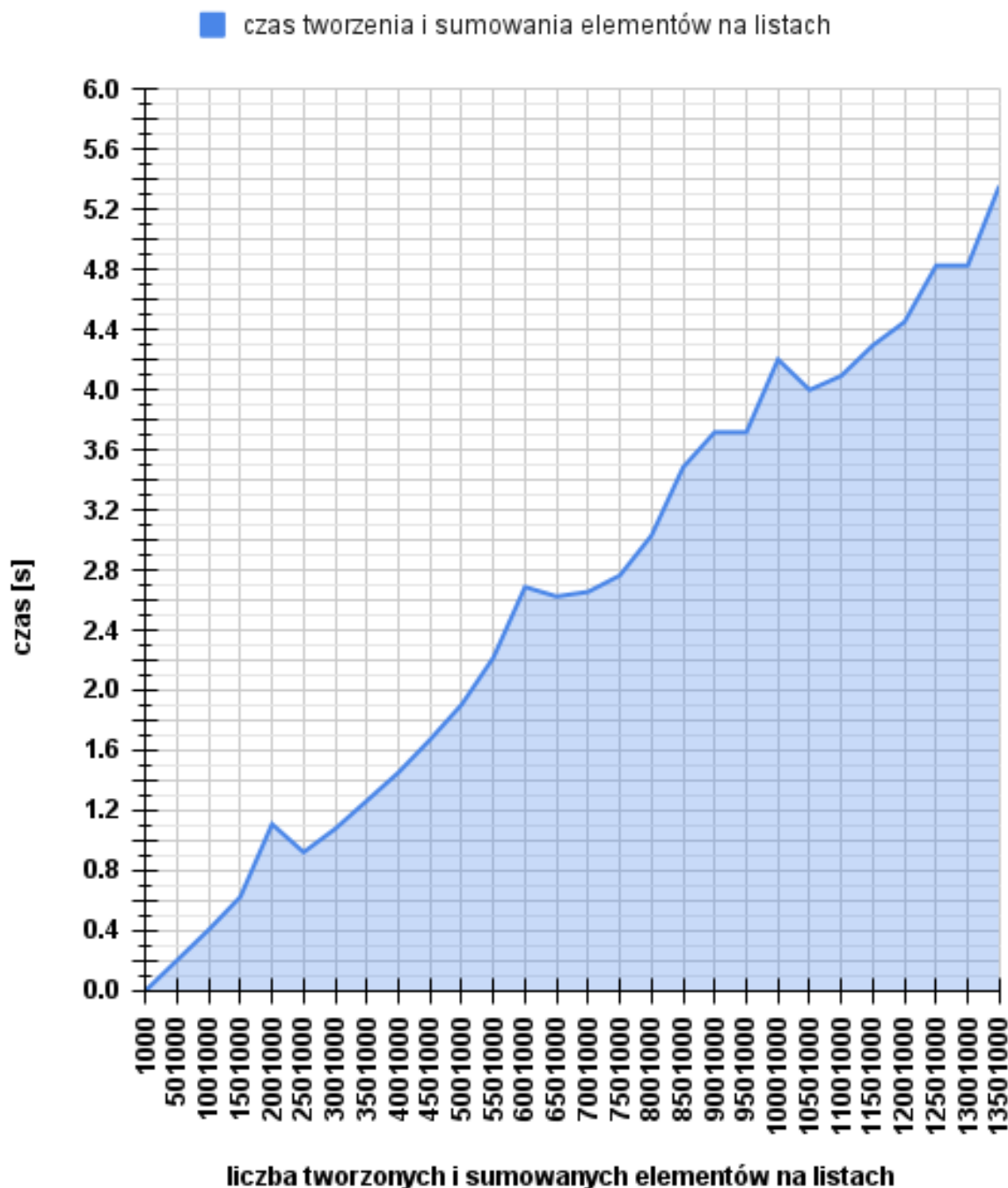
6501000	2.625
7001000	2.656
7501000	2.765
8001000	3.031
8501000	3.485
9001000	3.719
9501000	3.718
10001000	4.203
10501000	4
11001000	4.094
11501000	4.297
12001000	4.453
12501000	4.828
13001000	4.828
13501000	5.359

- **Wykres charakterystyki złożoności dla listy dwukierunkowej.**

Na podstawie uzyskanych pomiarów stworzyłam wykres charakterystyki złożoności czasowej na liście dwukierunkowej po “ulepszeniu”, dostosowując skale na obu osiach.

Charakterystyka złożoności obliczeniowej

dla operacji na liście dwukierunkowej - PO ULEPSZENIU (PRO)



- **Kod programu.**

Kompletny kod programu znajduje się poniżej. Jest również do pobrania w zdalnym repozytorium razem z całym projektem (link na początku sekcji Laboratorium nr 3).

```
// Autor: Magda Szafranska, nr indeksu AHNS: 18345
// Informatyka NST, rok 2, sem. 3
// Algorytmy, laboratoria nr 3 - part II

#include <iostream>
#include <Windows.h>
#include <fstream>

using namespace std;

class TItem
{
    friend class TList;
public: int FData;
        TItem* FNext;
        TItem* FPrev;

    public:
        TItem(TItem* APrev, TItem* ANext, int AData)
        {
            FPrev = APrev;
            FNext = ANext;
            FData = AData;

            if (FPrev) FPrev -> FNext = this;
            if (FNext) FNext -> FPrev = this;
        }
        ~TItem(void)
        {
            if (FPrev) FPrev->FNext = FNext;
            if (FNext) FNext->FPrev = FPrev;
        }
        TItem(TItem* ANext, int AData)
        {
            FData = AData;
            FNext = ANext;
        }
};

class TList
{
private:
    int FCount;
    TItem* FFirst;
    TItem* Flast;
    int FIndex;
    TItem* FCurr;
public:
    TList(void)
    {
        FCount = 0;
        FFirst = Flast = NULL;
        FIndex = -1;
        FCurr = NULL;
    }
    ~TList(void)
    {
        Clear();
    }
}
```

```

void Clear(void)
{
    while (Flast)
    {
        if (Flast->FNext)
            delete Flast->FNext;
        Flast = Flast->FPprev;
    }
    FFirst = NULL;
    FCount = 0;
    FIndex = -1;
    FCurr = NULL;
}

int Add(double AData)
{
    Flast = new TItem(Flast, NULL, AData);
    if (!FFirst) FFirst = Flast;
    return FCount++;
}

TItem* Get(int AIndex)
{
    if (FIndex < 0)
    {
        FIndex = 0;
        FCurr = FFirst;
    }
    while (AIndex < FIndex)
    {
        FCurr = FCurr->FPprev;
        FIndex--;
    }
    while (AIndex > FIndex)
    {
        FCurr = FCurr->FNext;
        FIndex++;
    }
    return FCurr;
}

void Del(int Index)
{
    TItem* Item = Get(Index);
    if (Item == FFirst) FFirst = Item->FNext;
    if (Item == Flast) Flast = Item->FPprev;
    delete Item;
    --FCount;
    FIndex = -1;
    FCurr = NULL;
}

int Count(void)
{
    return FCount;
}

int operator[](int Index)
{
    return Get(Index) -> FData;
}

int Pop(void);
void Push(int AData);
bool IsExist(int AData);
};

void TList::Push(int AData)
{
    FFirst = new TItem(FFirst, AData);
}

```

```

}
int TList::Pop(void)
{
    int AData = FFirst -> FData;
    TItem* AItem = FFirst;
    FFirst = FFirst -> FNext;
    delete AItem;
    return AData;
}
bool TList::IsExist(int AData)
{
    TItem* Item = FFirst;
    while (Item)
        if (Item -> FData == AData) return true;
        else Item = Item -> FNext;
    return false;
}

int main()
{
    TList list;
    list.Push(1);
    list.Push(2);
    list.Push(5);

    int x1 = list.Pop();
    int x2 = list.Pop();
    int x3 = list.Pop();

    cout << "x1 = " << x1 << endl;
    cout << "x2 = " << x2 << endl;
    cout << "x3 = " << x3 << endl << endl;

    // variables to writing to the text file
    fstream fTime, fCount;
    fTime.open("time_pro.txt", ios::out | ios::app);
    fCount.open("count_pro.txt", ios::out | ios::app);

    int T1, T2;
    int Count = 1000;
    int b = 500000;

    for (int j = 1; j <= 27; j += 1)    //the loop will execute 27 times
    {
        list.Clear();                // Clearing the list each time

        double Sum = 0.0;
        //Count = 10000000 +b;
        fCount << Count << endl;        // Writing current Count to the file
        cout << j << ". Count: " << Count;

        int T1 = GetTickCount64();

        while (Count--)
        {
            list.Add(rand());
        }

        for (int i = 0; i < list.Count(); ++i)
        {
            Sum += list[i];
        }
        int T2 = GetTickCount64();

        cout << "    | Time: " << (T2 - T1) * 0.001 << " seconds." << endl;
        fTime << (T2 - T1) * 0.001 << endl;
    }
}

```

```
        b += 500000;  
        Count = 1000 + b;  
    }  
    fTime.close();  
    fCount.close();  
}
```

- **Podsumowanie**

Dodatkowe fragmenty w kodzie (w wersji “ulepszonej”) sprawiły, że złożoność obliczeniowa algorytmu nieporównywalnie zmalała. Spowodowała to głównie zmiana przetwarzania elementów: z pamięci lub z indeksów.

Kompletne wnioski znajdują się na końcu sprawozdania w sekcji [Wnioski](#).

Wnioski

Dot. laboratorium nr 1

Procesor oblicza i wykonuje polecenia tak szybko, że czas wykonania jest niemal bliski zeru. Zachodzące obliczenia są niemal niewidoczne dla ludzkiego oka. Widzimy różnicę w wynikach przy pomiarze czasu wykonywania danej operacji na macierzy o wymiarach 50x10 oraz 500x100. Nie jest ona jednak tak znaczna jak można by było się spodziewać. Na przedstawionych przykładach wypełnienia i wypisywania macierzy oraz pomnożenia i wypisywania macierzy, czasy wykonywania sposobem z funkcją `GetTickCount64()` są identyczne co do drugiego miejsca po przecinku.

Analogiczne wnioski można wysnuć obserwując czas wykonywania analogicznych pomiarów drugim sposobem. Wynik wyrażony jest tam w sekundach, a więc po porównaniu obu wielkości widać niemal identyczne wyniki. Ta niewielka ewentualna różnica kilku milisekund jest spowodowana kolejnością wywołania rozpoczęcia i zakończenia pomiarów pomiędzy poszczególnymi metodami.

Dot. laboratorium nr 2

Analizowane przeze mnie algorytmy były podstawowymi przedstawicielami metod sortowania. W rezultacie pomiarów ich czasów wykonywania otrzymałam charakterystyki złożoności obliczeniowej wszystkich badanych algorytmów dzięki czemu mogłam porównać, który z nich jest najlepszy (tj. najefektywniejszy w użyciu), a który najgorszy. Wykresy charakterystyk świetnie zilustrowały te różnice.

Najwolniejsze ze wszystkich metod sortowania okazało się sortowanie bąbelkowe, a najszybszy był algorytm sortowania przez zliczanie. Wynika to bezpośrednio ze sposobu, w jaki każdy w nich wyznacza kolejne elementy w tablicy.

Sortowanie przez zliczanie ma złożoność obliczeniową $O(n+m)$, gdzie m to rozpiętość danych. Nie wykonuje ono żadnych porównań, dzięki czemu dużą zaletą jest stabilność tego algorytmu. Jego działanie, najogólniej ujmując, polega na zliczaniu ilości wystąpień poszczególnych elementów tablicy i zapisywaniu odpowiedniej ilości pól w dodatkowej tablicy. Jest to jednocześnie jego wadą gdyż konieczne jest użycie dodatkowej pamięci do przechowywania wystąpień elementów tablicy. Należy go zatem stosować z rozwagą adekwatnie do potrzeb i dostępnych zasobów.

Algorytm sortowania bąbelkowego z kolei opiera się na zasadzie maksimum, tj. każda liczba jest mniejsza lub równa od liczby maksymalnej. Porównując kolejne elementy i zamieniając je kolejnością finalnie otrzymujemy uporządkowany ciąg. Jest to jednak niezmiernie czasochłonne. Można go stosować tylko dla niewielkiej liczby elementów w sortowanym zbiorze (do około 5000). Przy większych zbiorach czas sortowania może być zbyt długi, tak jak doskonale dało się to zaobserwować w niniejszym laboratorium.

Analiza złożoności badanych algorytmów pozwala szacować czas ich wykonania. Ma ona jednak pewne ograniczenia, m.in. paralelizację - przede wszystkim bada ona jedynie sumaryczny czas potrzebny na wykonanie instrukcji, zaniedbując, czy da się ten czas rozłożyć na wiele rdzeni w zależności od sprzętu, na którym te algorytmy są wykonywane. Po drugie, analiza asymptotyczna "ignoruje stałe", traktując je jako szczegóły implementacji. W skrajnych przypadkach może mieć to jednak kluczowe znaczenie i przesunąć algorytm o jeden rząd wyżej lub niżej w rankingu efektywności.

W pierwszym laboratoryjnym przypadku dowiodłam, że złożoność operacji sprawdzania czy element znajduje się na liście jest rzędu liniowego czyli $O(n)$. Wynika to zarówno z dokumentacji sprawozdania jak i z teorii złożoności obliczeniowej.

Analizując drugi przykład z listą dwukierunkową, złożoność - nawet tak ogólnie pojętą (w przybliżeniu) - możemy poprawiać przebudowując dany algorytm. Tak też było w moim tym przypadku. Dodatkowe fragmenty w kodzie (w wersji "ulepszonej") sprawiły, że złożoność obliczeniowa algorytmu wstawiania elementów do list i ich sumowania nieporównywalnie zmalała. Spowodowała to zmiana sposobu przetwarzania elementów: z wczytywania obiektów z pamięci na przetwarzanie ich jako indeksy obiektów. Po poprawieniu algorytmu, przy zachowaniu tych samych parametrów co uprzednio, jego złożoność zauważalnie zmalała, co wyraźnie uwidoczniły wykresy charakterystyk złożoności.