

**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

<b>Wstęp</b>	<b>3</b>
Użyte technologie	3
Zastosowana konwencja w pisaniu kodu projektu	3
Pliki źródłowe	3
<b>Cel laboratoriów</b>	<b>3</b>
<b>Laboratorium nr 1</b>	<b>5</b>
<b>Wstęp</b>	<b>5</b>
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
<b>Laboratoria nr 2</b>	<b>14</b>
<b>Wstęp</b>	<b>14</b>
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
<b>Laboratoria nr 3</b>	<b>30</b>
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 iterujący elementy na liście - wersja podstawowa	39
Algorytm iterujący elementy na liście - wersja ulepszona	43
<b>Wnioski</b>	<b>52</b>
Dot. laboratorium nr 1	52
Dot. laboratorium nr 2	52
Dot. laboratorium nr 3	52



# Wstęp

## Użyte technologie

- Na wszystkich laboratoriach korzystałam z następujących narzędzi:
  - język programowania C++
  - GIT
  - Github
  - Git Bash
  - Google Documents
- Dodatkowo specyficznie dla wybranych laboratoriów:
  - Laboratorium nr 1 i 3:
    - Microsoft Visual Studio Community 2019 (wersja 16.11.1)
    - Arkusze Google Sheets
  - 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 poprawność 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 operacji na macierzach, 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

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

## Cel laboratorium nr 1

Pliki projektu znajduje się w publicznym repozytorium zdalnym Gita w Githubie.

Link do niego: [https://github.com/Yaviena/Matrix\\_operations\\_AHNS\\_Algorithms\\_Lab\\_1](https://github.com/Yaviena/Matrix_operations_AHNS_Algorithms_Lab_1).

Algorytmy do zaimplementowania 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

## 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** tabl = CreateTab(rowCount, colCount);
```

```
int main()
{
    int rowCount = 5;
    int colCount = 4;
}
```

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

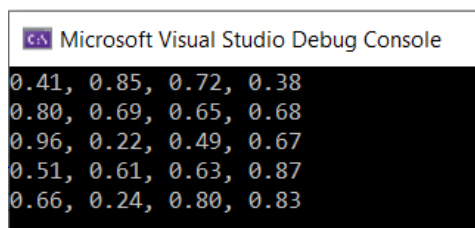
```
RandomTab(rowCount, colCount, tab1);
PrintTab(rowCount, colCount, tab1);
DeleteTab(rowCount, colCount, tab1);
```

- 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.



## 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);
```

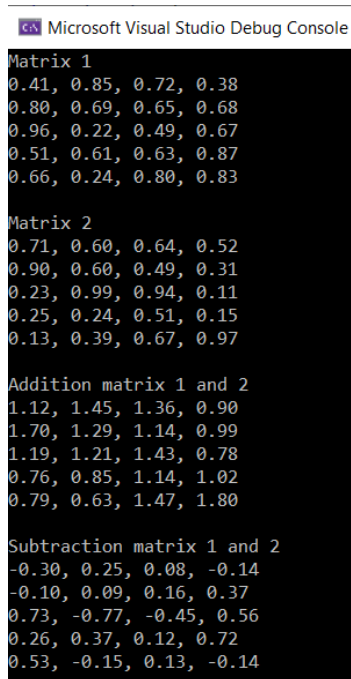
```
void SubtractTab(int rowCount, int colCount, double** tab1, double** tab2, double** tab3);
```

- 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 x 2    These should be equal    2 x 3    3 x 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);
```

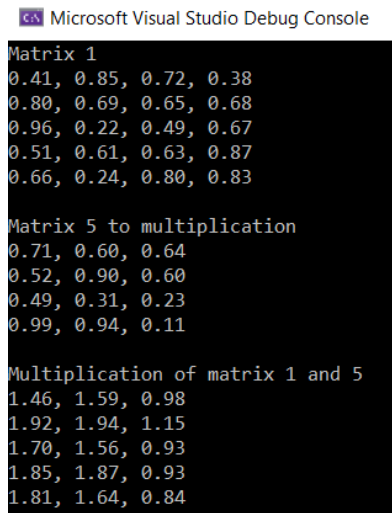
Wywołałam funkcję podając za argumenty kolejno zgodnie z definicją:

- 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 zmieniałam kolor wyświetlanego wewnątrz komunikatu na żółty.

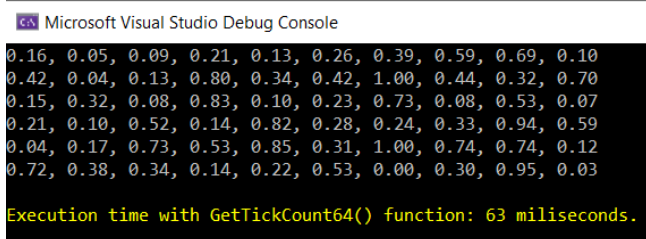
- Bezpośrednio przed wywołaniem funkcji wypełniającej tablicę oraz bezpośrednio po funkcji wypisującej dane na ekran wywołuję funkcję GetTickCount64() na zmiennych lokalnych i zapisuję w nich wynik (zmierzony czas).

```
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

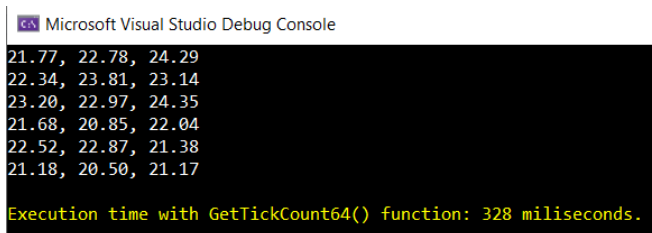
- 31 ms
- 47 ms
- 47 ms

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.

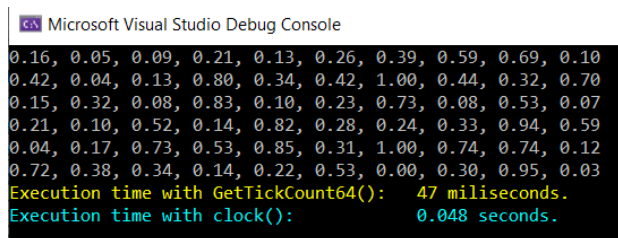
- Bezpośrednio przed wywołaniem funkcji wypełniającej tablicę oraz bezpośrednio po wypisaniu danych na ekran wywołuję funkcję GetTickCount64() na zmiennych lokalnych i zapisuję w nich wynik (zmierzony czas).

```
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.



```
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(): 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

## 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\\_Fuctions\\_graphs\\_Magda\\_Szafranska](https://github.com/Yaviena/Algorithms_Lab_2_Fuctions_graphs_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 Wykładowcy 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 ofstream 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 całkowitych >>>\n");
    if (argc != 2) {
        printf("Schemat wywołania 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("Nieprawidłowy 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 b1belkowe (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("\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("\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;
    }
//-----

```

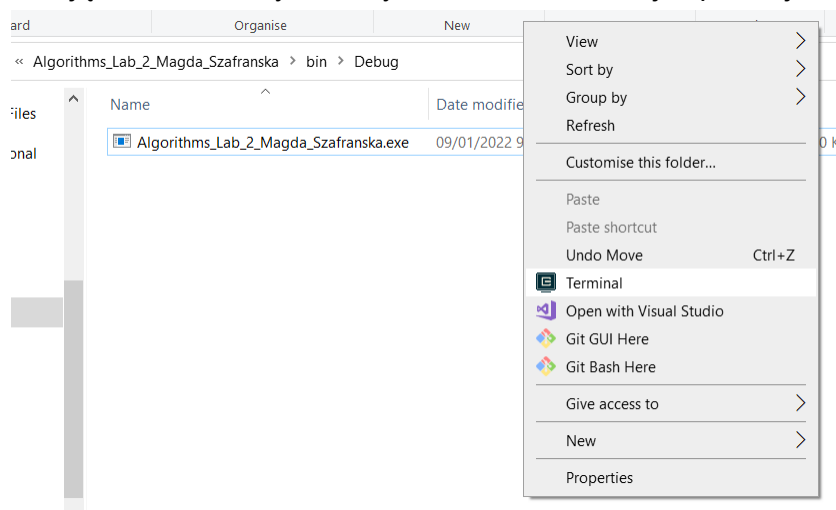
## Pomiar czasów wykonywania

- **Uruchomienie aplikacji**

Aby uruchomić plik .exe przeszedłam 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)
A Algorithms_Lab_2_Magda_Szafranska.exe 1000
<<< Test algorytm w sortowania tablicy liczb calkowitych >>>
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, 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 calkowitych >>>
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, 3, 4, 4, 4, 4, 5, 6, 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 calkowitych >>>
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, 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, 209
76, 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, 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

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).

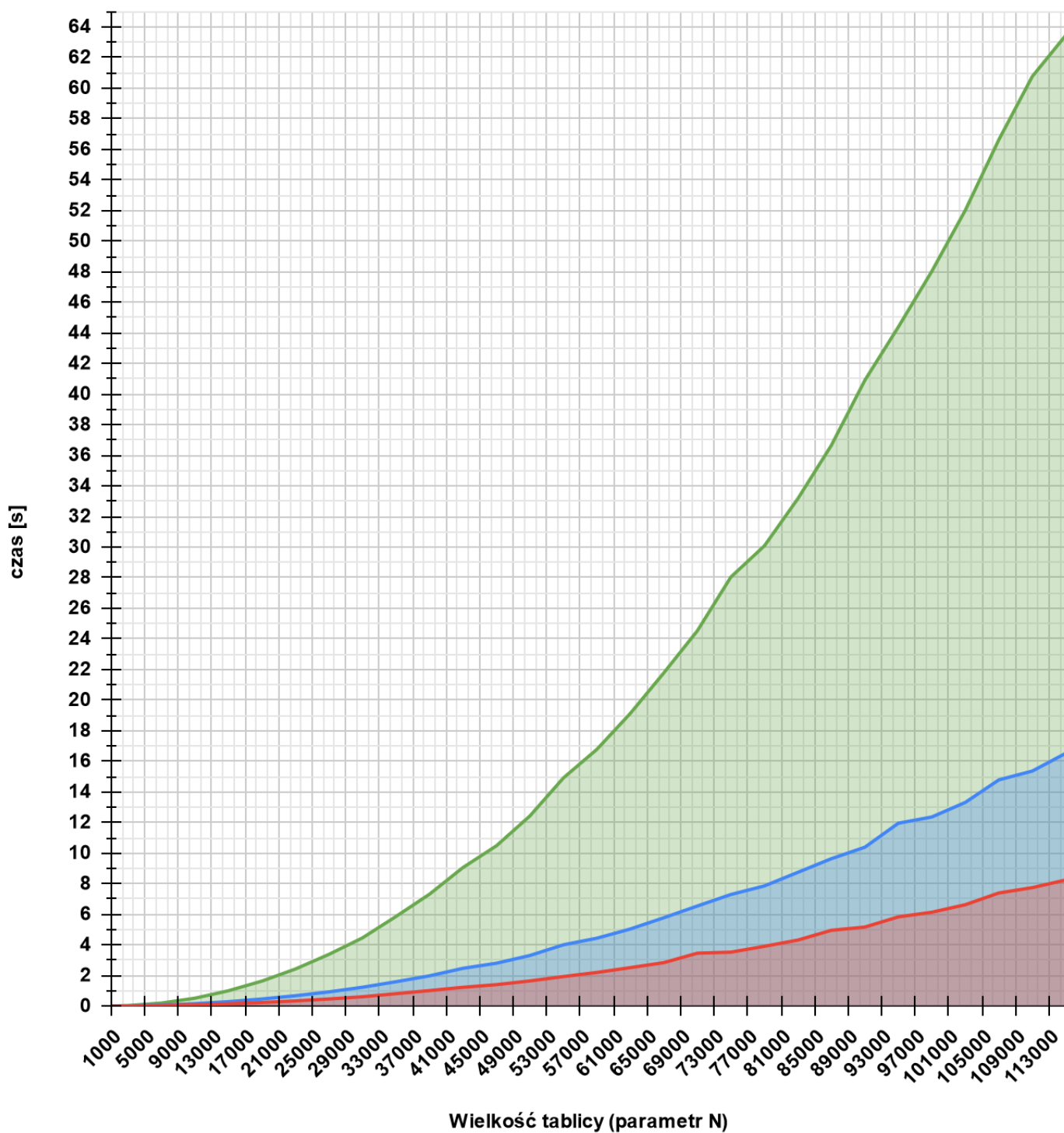
Na osi poziomej oznaczyłam N, czyli długość tablicy - parametr podawany podczas wywoływania programu.

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

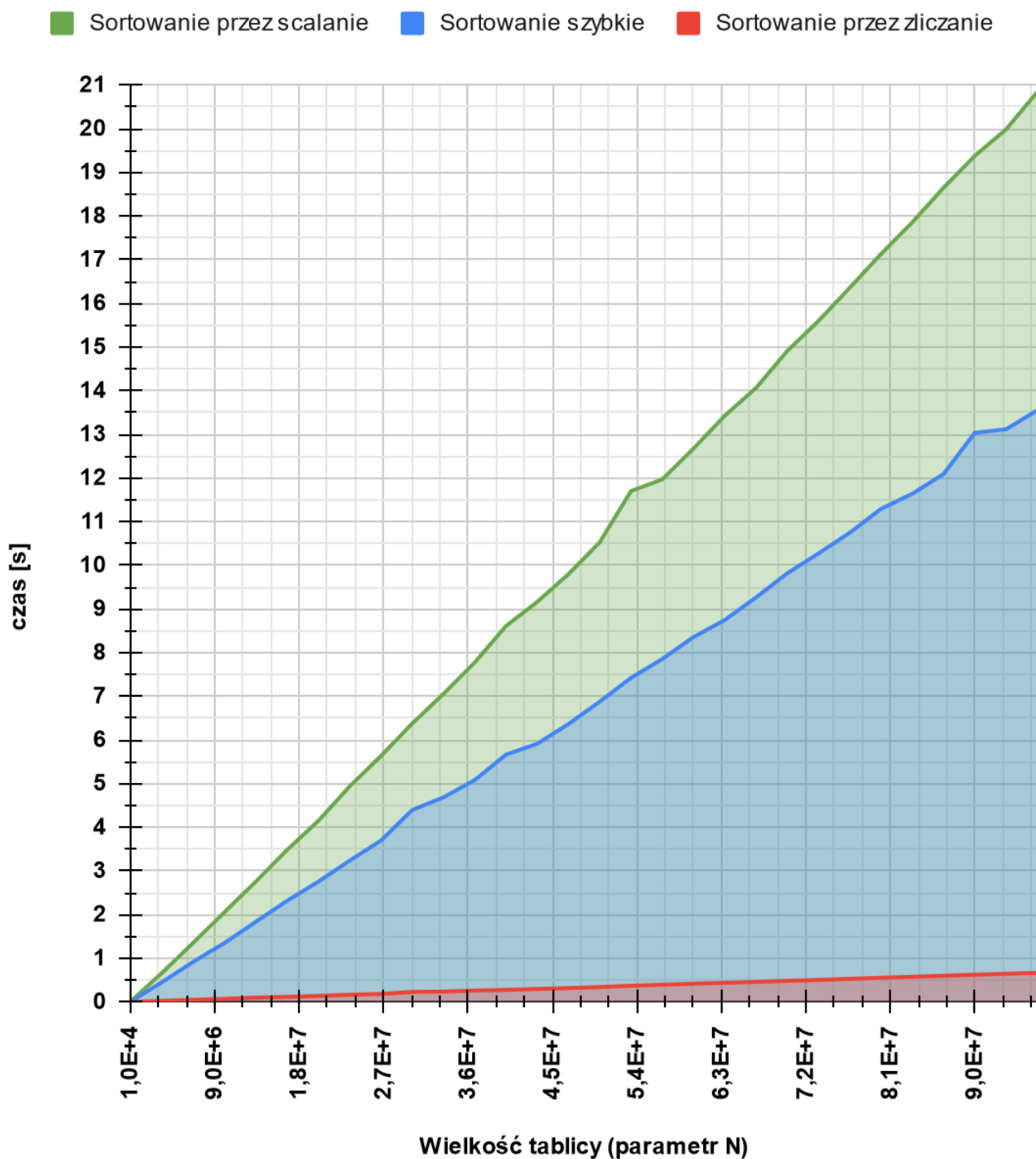
- **Pomiary użyte do wykresu.**

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:

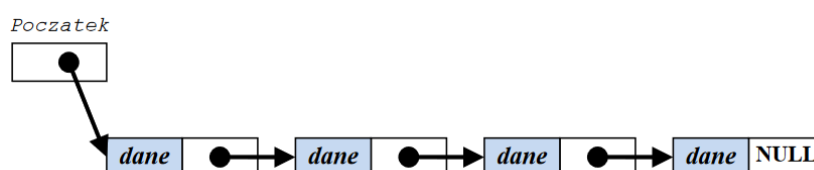
- stos,
- kolejka,
- kopiec,
- drzewo,
- tablica,
- graf
- listy.

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.

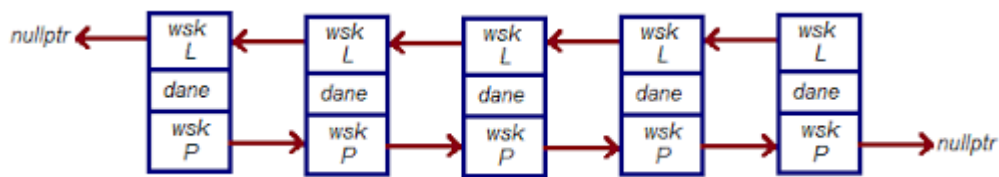
Aby zaimplementować listę jednokierunkową w języku C/C++ trzeba zdefiniować strukturę pełniącą rolę węzłów stanowiących kolejne elementy listy. Taka struktura składa się z dwóch części: pola (lub kilku pól), w którym pamiętane są przechowywane elementy (dane użytkowe) oraz pola wskaźnikowego, w którym 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 przełączenia 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 w Githubie. Link do niego: [https://github.com/Yaviena/Algorithms\\_Lab\\_3\\_part\\_1\\_Magda\\_Szafranska](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).

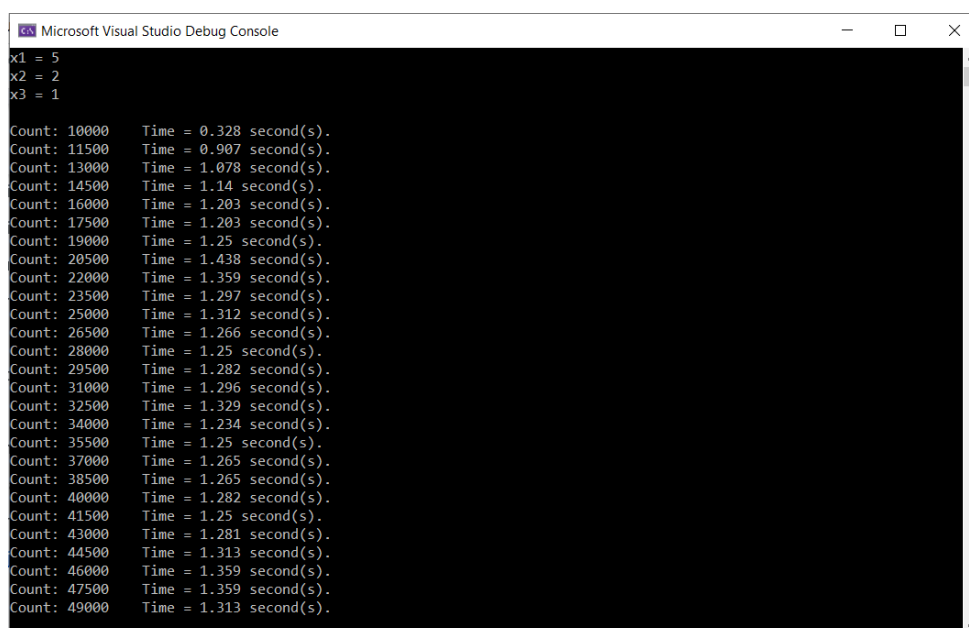


- **Funkcje obliczające czasy wykonywania.**

Do pomiaru czasów wykonywania użyłam podobnie jak przy laboratorium nr 1 funkcji `GetTickCount64()`. Funkcja ta wymaga użycia biblioteki `windows.h`. 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”` z funkcją oraz bezpośrednio po pętli wywoływałam funkcję `GetTickCount64()` na zmiennych lokalnych i zapisywałam w nich za każdym razem bieżące wyniki. Proces powtarzał się przez 27 iteracji zwiększając za każdym razem wielkość licznika o 1500 (od 10000 do 49000).

Poniżej zrzut ekranu z uzyskanymi pomiarami.



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

Count: 10000    Time = 0.328 second(s).
Count: 11500    Time = 0.907 second(s).
Count: 13000    Time = 1.078 second(s).
Count: 14500    Time = 1.14 second(s).
Count: 16000    Time = 1.203 second(s).
Count: 17500    Time = 1.203 second(s).
Count: 19000    Time = 1.25 second(s).
Count: 20500    Time = 1.438 second(s).
Count: 22000    Time = 1.359 second(s).
Count: 23500    Time = 1.297 second(s).
Count: 25000    Time = 1.312 second(s).
Count: 26500    Time = 1.266 second(s).
Count: 28000    Time = 1.25 second(s).
Count: 29500    Time = 1.282 second(s).
Count: 31000    Time = 1.296 second(s).
Count: 32500    Time = 1.329 second(s).
Count: 34000    Time = 1.234 second(s).
Count: 35500    Time = 1.25 second(s).
Count: 37000    Time = 1.265 second(s).
Count: 38500    Time = 1.265 second(s).
Count: 40000    Time = 1.282 second(s).
Count: 41500    Time = 1.25 second(s).
Count: 43000    Time = 1.281 second(s).
Count: 44500    Time = 1.313 second(s).
Count: 46000    Time = 1.359 second(s).
Count: 47500    Time = 1.359 second(s).
Count: 49000    Time = 1.313 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).

Przy pierwszym uruchomieniu programu podałam wartość `Count` równą 10000 i za każdym razem zwiększałam ten parametr o 1500. Poniżej screen zapisów z pliku.

count.bt	time.bt
1 10000	1 0.328
2 11500	2 0.907
3 13000	3 1.078
4 14500	4 1.14
5 16000	5 1.203
6 17500	6 1.203
7 19000	7 1.25
8 20500	8 1.438
9 22000	9 1.359
10 23500	10 1.297
11 25000	11 1.312
12 26500	12 1.266
13 28000	13 1.25
14 29500	14 1.282
15 31000	15 1.296
16 32500	16 1.329
17 34000	17 1.234
18 35500	18 1.25
19 37000	19 1.265
20 38500	20 1.265
21 40000	21 1.282
22 41500	22 1.25
23 43000	23 1.281
24 44500	24 1.313
25 46000	25 1.359
26 47500	26 1.359
27 49000	27 1.313
28	28

- **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]
10000	0,328
11500	0,907
13000	1,078
14500	1,14
16000	1,203
17500	1,203
19000	1,25
20500	1,438
22000	1,359
23500	1,297
25000	1,312
26500	1,266
28000	1,25
29500	1,282
31000	1,296
32500	1,329
34000	1,234
35500	1,25
37000	1,265

38500	1,265
40000	1,282
41500	1,25
43000	1,281
44500	1,313
46000	1,359
47500	1,359
49000	1,313

- **Dopasowanie skali osi.**

Maksymalna wartość na osi czasu to 1,5 sekundy. Liczba dodawanych elementów do list (Count) waha się w moich pomiarach od 10000 do 49000 (z krokiem co 1500), które są jednocześnie moimi skrajnymi wartościami. Oś czasu podzieliłam adekwatnie do powyższego proporcjonalnie według skali liniowej na odcinki czasowe co 0.05 s (co 1 cm).

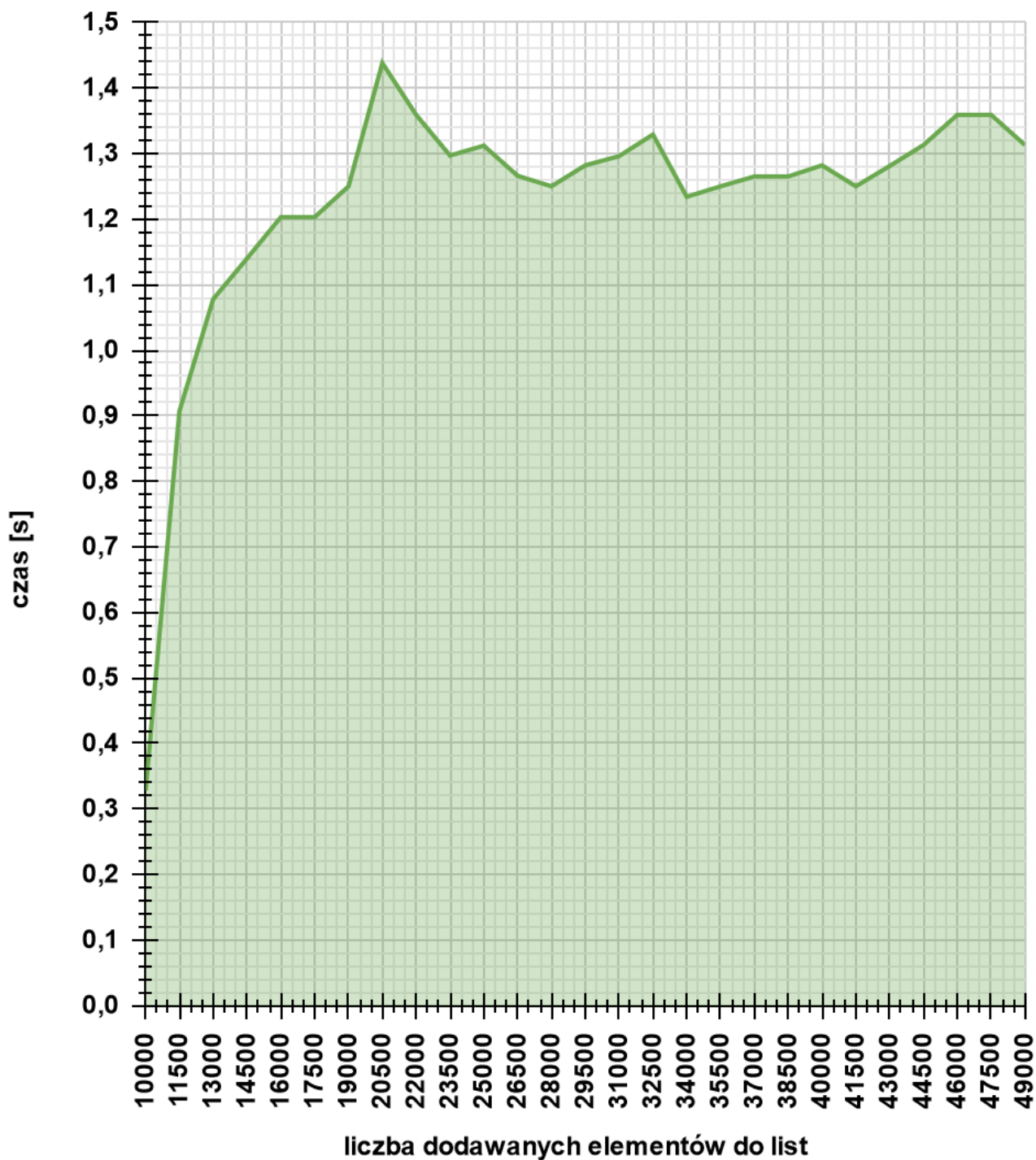
- **Wykres charakterystyki złożoności dla listy jednokierunkowej**

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



- **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 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();

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

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

    int T1, T2;

    for (int i = 10000; i <= 50000; i += 1500)
    {
        int Count = 10000;
        fCount << i << endl;

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

        cout << "Count: " << i << "\tTime = " << (T2 - T1) * 0.001 << " second(s)."
<< endl;
        fTime << (T2 - T1) * 0.001 << endl;
    }
}

```

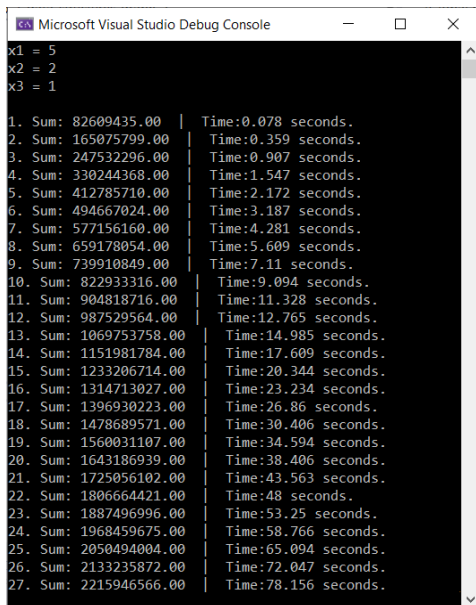
```
}  
  
fTime.close();  
fCount.close();  
}
```

## Wykonanie ćwiczenia cz. II: lista dwukierunkowa

W drugiej części zajęć zajęliśmy się algorytmem iterującym elementy na liście dwukierunkowej. 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 iterujący elementy na liście - wersja podstawowa



```
Microsoft Visual Studio Debug Console
x1 = 5
x2 = 2
x3 = 1
1. Sum: 82609435.00 | Time:0.078 seconds.
2. Sum: 165075799.00 | Time:0.359 seconds.
3. Sum: 247532296.00 | Time:0.907 seconds.
4. Sum: 330244368.00 | Time:1.547 seconds.
5. Sum: 412785710.00 | Time:2.172 seconds.
6. Sum: 494667024.00 | Time:3.187 seconds.
7. Sum: 577156160.00 | Time:4.281 seconds.
8. Sum: 659178054.00 | Time:5.609 seconds.
9. Sum: 739910849.00 | Time:7.11 seconds.
10. Sum: 822933316.00 | Time:9.094 seconds.
11. Sum: 904818716.00 | Time:11.328 seconds.
12. Sum: 987529564.00 | Time:12.765 seconds.
13. Sum: 1069753758.00 | Time:14.985 seconds.
14. Sum: 1151981784.00 | Time:17.609 seconds.
15. Sum: 1233206714.00 | Time:20.344 seconds.
16. Sum: 1314713027.00 | Time:23.234 seconds.
17. Sum: 1396930223.00 | Time:26.86 seconds.
18. Sum: 1478689571.00 | Time:30.406 seconds.
19. Sum: 1560031107.00 | Time:34.594 seconds.
20. Sum: 1643186039.00 | Time:38.406 seconds.
21. Sum: 1725056102.00 | Time:43.563 seconds.
22. Sum: 1806664421.00 | Time:48 seconds.
23. Sum: 1887496996.00 | Time:53.25 seconds.
24. Sum: 1968459675.00 | Time:58.766 seconds.
25. Sum: 2050494004.00 | Time:65.094 seconds.
26. Sum: 2133235872.00 | Time:72.047 seconds.
27. Sum: 2215946566.00 | Time:78.156 seconds.
```

- **Implementacja zapisu pomiarów do plików**

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



sum1.txt	time1.txt
1 10000	1 0.078
2 11500	2 0.359
3 13000	3 0.907
4 14500	4 1.547
5 16000	5 2.172
6 17500	6 3.187
7 19000	7 4.281
8 20500	8 5.609
9 22000	9 7.11
10 23500	10 9.094
11 25000	11 11.328
12 26500	12 12.765
13 28000	13 14.985
14 29500	14 17.609
15 31000	15 20.344
16 32500	16 23.234
17 34000	17 26.86
18 35500	18 30.406
19 37000	19 34.594
20 38500	20 38.406
21 40000	21 43.563
22 41500	22 48
23 43000	23 53.25
24 44500	24 58.766
25 46000	25 65.094
26 47500	26 72.047
27 49000	27 78.156
28	28

- **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	0.078
11500	0.359
13000	0.907
14500	1.547
16000	2.172
17500	3.187
19000	4.281
20500	5.609
22000	7.11
23500	9.094
25000	11.328
26500	12.765
28000	14.985
29500	17.609
31000	20.344
32500	23.234

34000	26.86
35500	30.406
37000	34.594
38500	38.406
40000	43.563
41500	48
43000	53.25
44500	58.766
46000	65.094
47500	72.047
49000	78.156

- **Dopasowanie skali osi.**

Maksymalna wartość na osi czasu to 80 sekund. Podzieliłam ją proporcjonalnie według skali liniowej na odcinki czasowe co 4 sekundy. Liczba iterowanych elementów do list waha się w moich pomiarach od 10000 do 49000 (z krokiem co 1500), które są jednocześnie moimi skrajnymi wartościami. Oś czasu podzieliłam co 1500 elementów.

- **Wykres charakterystyki złożoności dla listy jednokierunkowej**

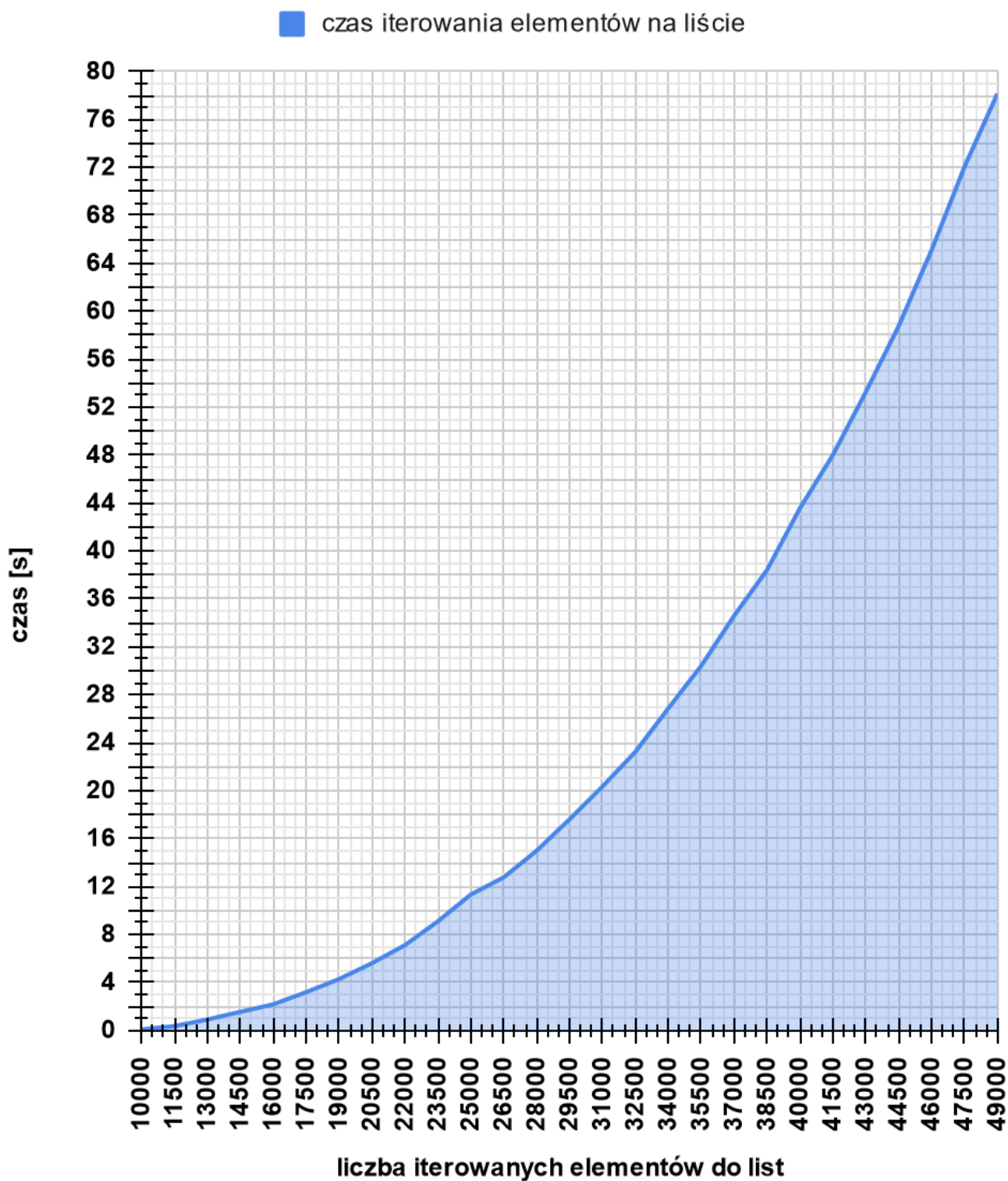
Na podstawie uzyskanych pomiarów stworzyłam wykres charakterystyki złożoności czasowej iterowania elementów na liście dwukierunkowej.

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

Wykres

# Charakterystyka złożoności obliczeniowej

## dla operacji na liście dwukierunkowej - PRZED ULEPSZENIEM



- **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 iterujący elementy na liście - wersja ulepszona

Poprzedni algorytm już dla niemal 50000 elementów miał bardzo duży czas wykonywania. Aby poprawić jego złożoność obliczeniową, w kilku miejscach wstawiłam dodatkowy fragment kodu. Ma on na celu ...

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

Dodatkowo zmodyfikowałam funkcję Get(). W ulepszonej wersji ...  
Poniżej porównanie wersji przed oraz po modyfikacji.

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

Po poprawieniu algorytmu jego złożoność zauważalnie zmalała. Poniżej screen z liczbą iterowanych elementów oraz czasem.

```

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

1. Sum: 822933316.00 | Time:0 seconds.
2. Sum: 1643186939.00 | Time:0.015 seconds.
3. Sum: 2461120012.00 | Time:0.015 seconds.
4. Sum: 3279216430.00 | Time:0.016 seconds.
5. Sum: 4101006728.00 | Time:0.016 seconds.
6. Sum: 4923796687.00 | Time:0.015 seconds.
7. Sum: 5740603624.00 | Time:0.032 seconds.
8. Sum: 6558804268.00 | Time:0.032 seconds.
9. Sum: 7379133874.00 | Time:0.031 seconds.
10. Sum: 8196278929.00 | Time:0.031 seconds.
11. Sum: 9014772964.00 | Time:0.047 seconds.
12. Sum: 9834056790.00 | Time:0.047 seconds.
13. Sum: 10653364755.00 | Time:0.047 seconds.
14. Sum: 11469105828.00 | Time:0.047 seconds.
15. Sum: 12290857983.00 | Time:0.062 seconds.
16. Sum: 13110175443.00 | Time:0.062 seconds.
17. Sum: 13932967114.00 | Time:0.078 seconds.
18. Sum: 14748730157.00 | Time:0.063 seconds.
19. Sum: 15568466044.00 | Time:0.062 seconds.
20. Sum: 16387983044.00 | Time:0.078 seconds.
21. Sum: 17204348882.00 | Time:0.078 seconds.
22. Sum: 18023987746.00 | Time:0.235 seconds.
23. Sum: 18845447350.00 | Time:0.172 seconds.
24. Sum: 19665529403.00 | Time:0.094 seconds.
25. Sum: 20484306341.00 | Time:0.11 seconds.
26. Sum: 21303813624.00 | Time:0.109 seconds.
27. Sum: 22126021282.00 | Time:0.094 seconds.

```

- **Implementacja zapisu pomiarów do plików**

Pomiary ulepszonej wersji również zapisałam do plików (*time2.txt* oraz *sum2.txt*), innych niż przed ulepszeniem - aby móc je następnie porównać niezależnie.

sum2.txt	time2.txt
1 10000	1 0
2 11500	2 0.015
3 13000	3 0.015
4 14500	4 0.016
5 16000	5 0.016
6 17500	6 0.015
7 19000	7 0.032
8 20500	8 0.032
9 22000	9 0.031
10 23500	10 0.031
11 25000	11 0.047
12 26500	12 0.047
13 28000	13 0.047
14 29500	14 0.047
15 31000	15 0.062
16 32500	16 0.062
17 34000	17 0.078
18 35500	18 0.063
19 37000	19 0.062
20 38500	20 0.078
21 40000	21 0.078
22 41500	22 0.235
23 43000	23 0.172
24 44500	24 0.094
25 46000	25 0.11
26 47500	26 0.109
27 49000	27 0.094
28	28

- **Dane do wykresu charakterystyki złożoności.**

Zapisane w plikach tekstowych pomiary (z dopiskiem "2") skopiowałam do dokumentu Google Sheets. Posłużyły mi one do wygenerowania charakterystyki złożoności obliczeniowej.

Count	Time 2 [s] (after)
10000	0
11500	0.015
13000	0.015
14500	0.016
16000	0.016
17500	0.015
19000	0.032
20500	0.032
22000	0.031
23500	0.031
25000	0.047
26500	0.047
28000	0.047
29500	0.047
31000	0.062
32500	0.062
34000	0.078
35500	0.063
37000	0.062
38500	0.078
40000	0.078
41500	0.235
43000	0.172
44500	0.094
46000	0.11
47500	0.109
49000	0.094

- **Dopasowanie skali osi.**

Dla uzyskania porównywalnych wyników dokonałam pomiarów w tych samych warunkach co przed modyfikacją kodu. Tym razem wyniki dla tych samych ilości elementów na liście były zdecydowanie korzystniejsze niemniej przy zachowaniu tej samej skali czasowej wyniki były całkowicie niewidoczne. Dostosowałam więc oś czasu do maksymalnej wartości. Maksymalna

wartość na osi czasu to 0.3 sekundy. Podzieliłam ją proporcjonalnie według skali liniowej na odcinki czasowe co 0.02 sekundy. Oś czasu pozostała bez zmian.

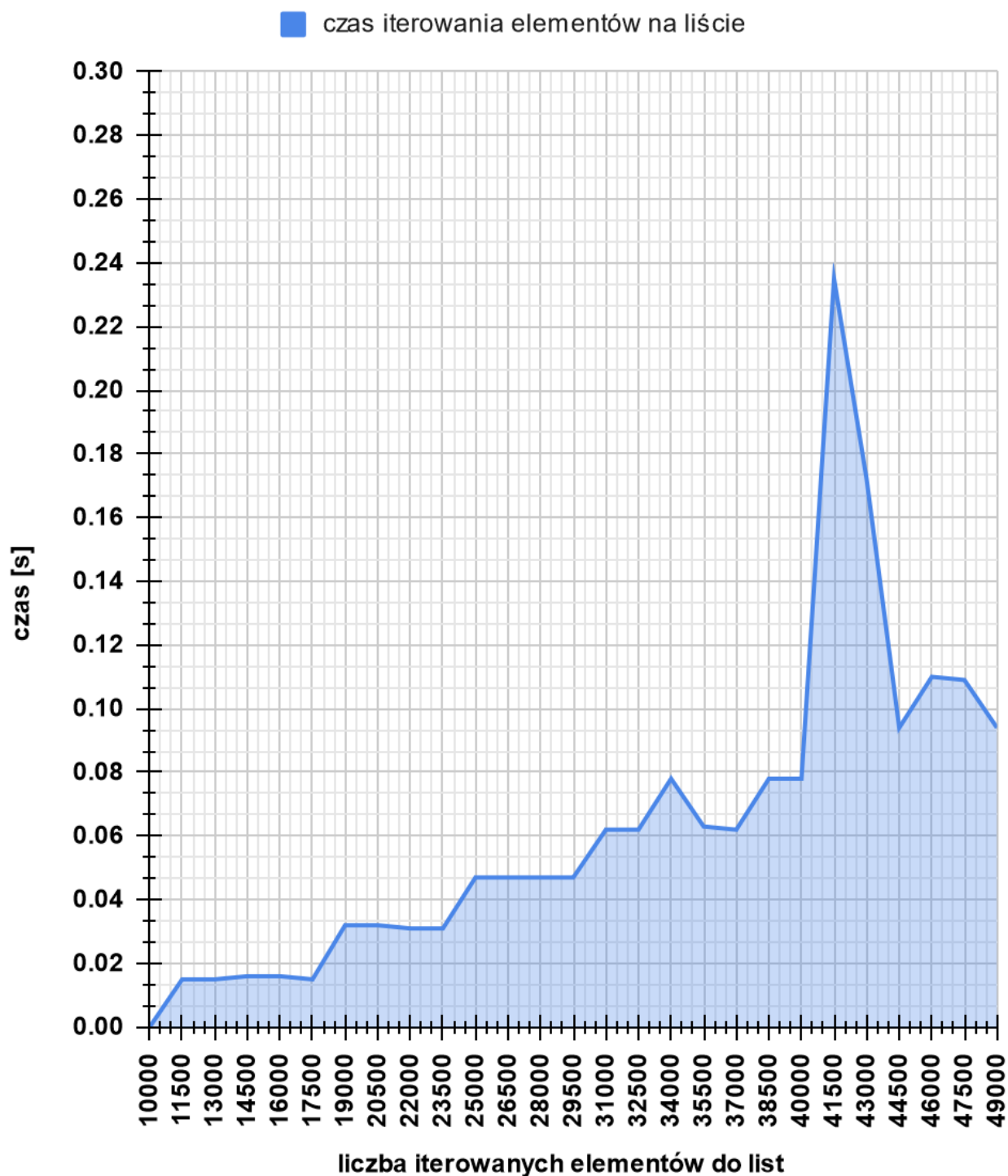
- **Wykres charakterystyki złożoności dla listy jednokierunkowej.**

Na podstawie uzyskanych pomiarów stworzyłam wykres charakterystyki złożoności czasowej iterowania elementów na liście dwukierunkowej.

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

# Charakterystyka złożoności obliczeniowej

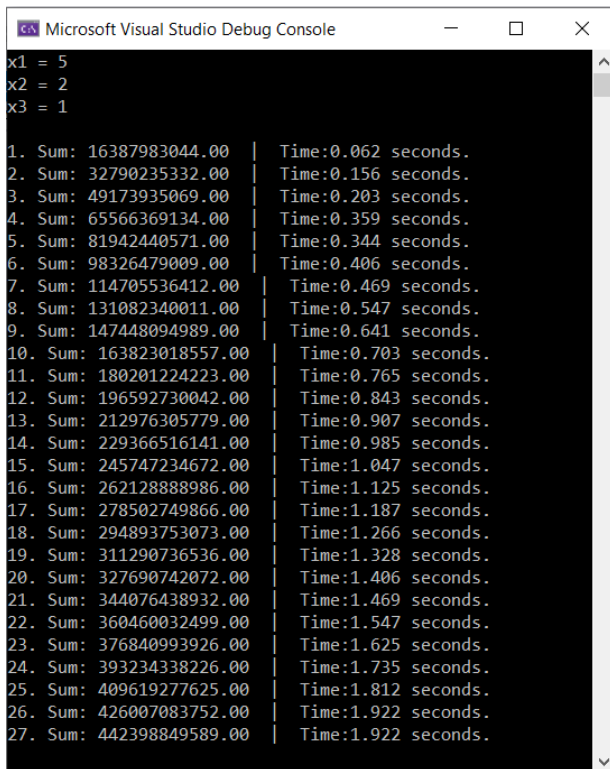
## dla operacji na liście dwukierunkowej - PO ULEPSZENIU





- **Wersja ulepszona PRO**

Dla lepszego zobrazowania tego, jak bardzo poprawiła się wydajność algorytmu po “ulepszeniu” go fragmentami kodu, wykonałam jeszcze jedną kompilację, tym razem podając na wejściu o wiele większe wartości.



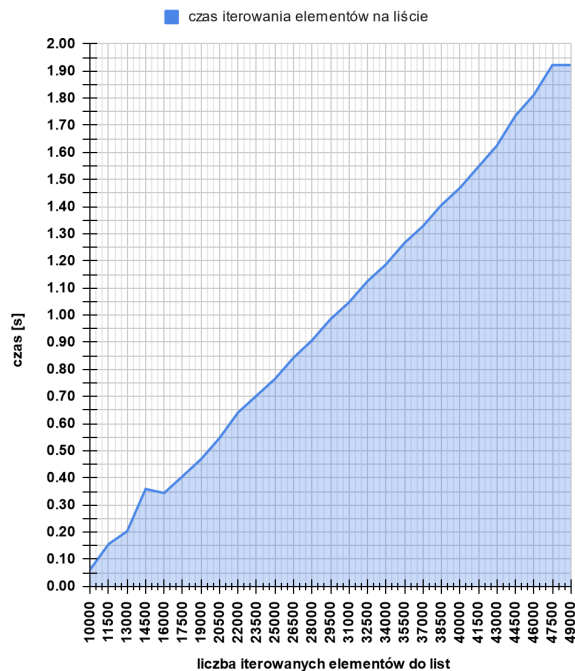
```
Microsoft Visual Studio Debug Console
x1 = 5
x2 = 2
x3 = 1

1. Sum: 16387983044.00 | Time:0.062 seconds.
2. Sum: 32790235332.00 | Time:0.156 seconds.
3. Sum: 49173935069.00 | Time:0.203 seconds.
4. Sum: 65566369134.00 | Time:0.359 seconds.
5. Sum: 81942440571.00 | Time:0.344 seconds.
6. Sum: 98326479009.00 | Time:0.406 seconds.
7. Sum: 114705536412.00 | Time:0.469 seconds.
8. Sum: 131082340011.00 | Time:0.547 seconds.
9. Sum: 147448094989.00 | Time:0.641 seconds.
10. Sum: 163823018557.00 | Time:0.703 seconds.
11. Sum: 180201224223.00 | Time:0.765 seconds.
12. Sum: 196592730042.00 | Time:0.843 seconds.
13. Sum: 212976305779.00 | Time:0.907 seconds.
14. Sum: 229366516141.00 | Time:0.985 seconds.
15. Sum: 245747234672.00 | Time:1.047 seconds.
16. Sum: 262128888986.00 | Time:1.125 seconds.
17. Sum: 278502749866.00 | Time:1.187 seconds.
18. Sum: 294893753073.00 | Time:1.266 seconds.
19. Sum: 311290736536.00 | Time:1.328 seconds.
20. Sum: 327690742072.00 | Time:1.406 seconds.
21. Sum: 344076438932.00 | Time:1.469 seconds.
22. Sum: 360460032499.00 | Time:1.547 seconds.
23. Sum: 376840993926.00 | Time:1.625 seconds.
24. Sum: 393234338226.00 | Time:1.735 seconds.
25. Sum: 409619277625.00 | Time:1.812 seconds.
26. Sum: 426007083752.00 | Time:1.922 seconds.
27. Sum: 442398849589.00 | Time:1.922 seconds.
```

Z zapisanych do pliku nowych danych utworzyłam analogiczny do poprzednich wykres, dostosowując skalę na 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->FPrev;
        }
        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->FPrev;
            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->FPrev;
    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;

    // writing to the text file

```

```

fstream fTime, fSum;
fTime.open("time2.txt", ios::out | ios::app);
fSum.open("sum2.txt", ios::out | ios::app);

int T1, T2;
int n = 1;

for (int i = 10000; i <= 50000; i += 1500)
{
    int Count = 50000;
    fSum << i << endl;

    while (Count-->0) list.Add(rand());
    int T1 = GetTickCount64();
    double Sum = 0.0;
    for (int i = 0; i < list.Count(); ++i)
        Sum += list[i];
    int T2 = GetTickCount64();

    cout << n << ". Sum: ";
    printf("%.2f", Sum);
    cout << " | Time:" << (T2 - T1) * 0.001 << " seconds." << endl;
    fTime << (T2 - T1) * 0.001 << endl;
    n++;
}

fTime.close();
fSum.close();
}

```

## - Podsumowanie

Wstawki w kodzie sprawiły, że złożoność obliczeniowa algorytmu nieporównywalnie zmalała.

1. Jaka jest charakterystyka i złożoność?
2. Dlaczego taka złożoność?
3. Co zrobić, żeby poprawić tę złożoność.

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 nie są widoczne 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 wypisania macierzy oraz pomnożenia i wypisania 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 algorytmów dzięki czemu mogłam porównać, który z nich jest najlepszy, 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 wyznaczania kolejnych elementów 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.

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.

## Dot. laboratorium nr 3