

# 

# 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 Szafrańska, nr albumu: 18345

# **Spis treści**

[**Wstęp**](#_96ogqibcquzr) **2**

[Użyte technologie](#_erqqk5t2kxkv) 2

[Zastosowana konwencja w pisaniu kodu projektu](#_6wx5bkm6qre5) 2

[Pliki źródłowe](#_c742m1flyfc3) 2

[**Cel laboratoriów**](#_et9tvfdsuaba) **3**

[**Laboratorium nr 1**](#_2p8p06ww51xv) **4**

[**Wstęp**](#_mzlbwkcor3mw) **4**

[Cel laboratorium nr 1](#_k2d9byv1uiar) 5

[Wykonanie ćwiczenia cz. I: implementacja algorytmów](#_6bf1p3z6fgw3) 5

[Create, RandomFill, Print, Delete](#_skf6u4i19jdp) 5

[Adding, Subtraction](#_4dag8l1oop5i) 6

[Copy](#_yk7351p2xt5e) 7

[Trans](#_fgg41177qgu9) 8

[Multiplication](#_nwnzhi8jl6uo) 9

[Wykonanie ćwiczenia cz. II: pomiar czasów wykonywania](#_aztjxeh7aqne) 10

[Funkcja GetTickCount()](#_47i27mj4n1b4) 10

[Funkcja clock()](#_pufxetwg0wlv) 12

[**Laboratoria nr 2**](#_b5sv99l86asd) **14**

[**Wstęp**](#_5m6fohacxpuw) **14**

[Cel laboratorium nr 2](#_32tt8xjltyl4) 14

[Wykonanie ćwiczenia cz. I: czasy wykonywania](#_j20bw2sjx46) 14

[Uruchomienie bazowej aplikacji](#_ovqt0yhtbynk) 14

[Pomiar czasów wykonywania](#_c5itmzw8gycd) 20

[Wykonanie ćwiczenia cz. II: charakterystyki złożoności](#_ps6upvmselfc) 21

[Generowanie wyników aplikacji](#_jm1chev5tcxi) 21

[Tworzenie wykresu dla sortowania bąbelkowego, przez wybór i przez wstawianie](#_m3zmmwrbq9ae) 22

[Tworzenie wykresu dla sortowania przez scalanie, szybkiego i przez zliczanie](#_3gl57ugetaax) 26

[**Laboratoria nr 3**](#_ireonqn763v3) **30**

[Wstęp](#_i2xpio86jos2) 30

[Cel laboratorium nr 3](#_22q29xxi2i3q) 31

[Wykonanie ćwiczenia cz. I: lista jednokierunkowa](#_83znd7qzd0im) 31

[Implementacja listy jednokierunkowej](#_gc2tuky7v6v1) 31

[Wykonanie ćwiczenia cz. II: lista dwukierunkowa](#_8zxvlpguw48x) 39

[Algorytm sumujący elementy na liście - wersja podstawowa](#_g9i29pi9b7b5) 39

[Algorytm sumujący elementy na liście - wersja ulepszona](#_at404rv1v73g) 43

[**Wnioski**](#_dh5ye5htrm90) **54**

[Dot. laboratorium nr 1](#_ywp9git68x1g) 54

[Dot. laboratorium nr 2](#_1b8hrom32vzo) 54

[Dot. laboratorium nr 3](#_52iv6957fecy) 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](https://github.com/Yaviena/Algorithms_Lab_1_Matrix_operations_Magda_Szafranska)
* [Laboratorium nr 2](https://github.com/Yaviena/Algorithms_Lab_2_Sorting_diagrams_Magda_Szafranska)
* Laboratorium nr 3
  + [część I](https://github.com/Yaviena/Algorithms_Lab_3_part_1_Magda_Szafranska) - lista jednokierunkowa
  + [część II](https://github.com/Yaviena/Algorithms_Lab_3_part_2_Magda_Szafranska) - lista dwukierunkowa

Link do niniejszego sprawozdania dostępny jest [tutaj](https://yaviena.github.io/Algorithms_report_AHNS/).

# **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 x 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 A3 x 4 posiada trzy wiersze i 4 kolumny. Elementy tej macierzy są następujące:

|  |  | a0,0 | ,a0,1 | ,a0,2 | ,a0,3 |  |
| --- | --- | --- | --- | --- | --- | --- |
| **A**3 x 4 = |  | a1,0 | ,a1,1 | ,a1,2 | ,a1,3 |  |
|  |  | a2,0 | ,a2,1 | ,a2,2 | ,a2,3 |  |

Element a2,1 leży w wierszu o numerze 2 (wiersz trzeci z uwagi na strat 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

### 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); |
| --- |

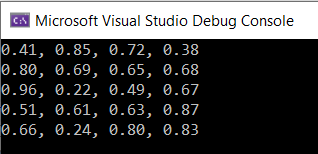
Na stworzonej tablicy wywołałam w funkcji main() metody do wypełnienia jej wartościami, wypisania w oknie konsoli oraz usunięcia tablicy.

| 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.  
  


### 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 |
| --- |

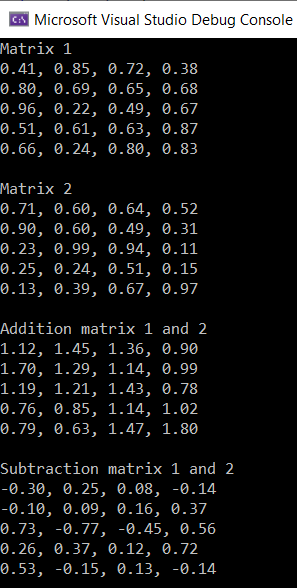
Utworzyłam prototypy funkcji dodawania i odejmowania podane w deklaracji.

| 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.



### 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.

### 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.

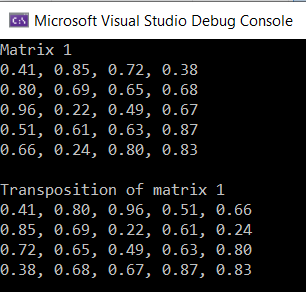
| void TransTab(int rowCount, int colCount, double\*\* tab, double\*\* temp\_tab); |
| --- |

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.

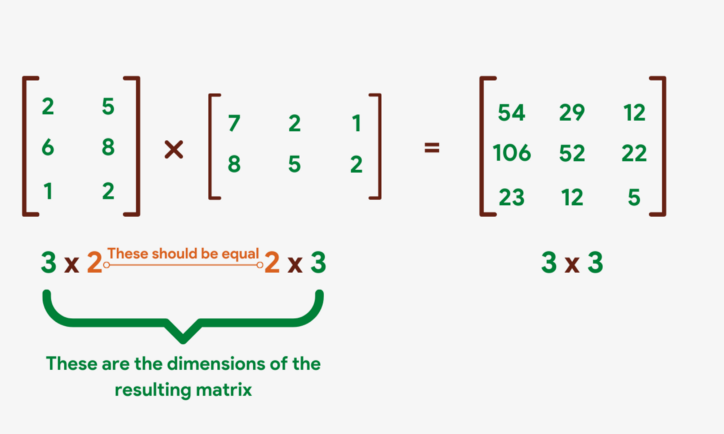
| cout << endl << "Transposition of matrix 1" << endl;  TransTab(rowCount, colCount, tab1, transTab);  PrintTab(colCount, rowCount, transTab); |
| --- |

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



### 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.



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.

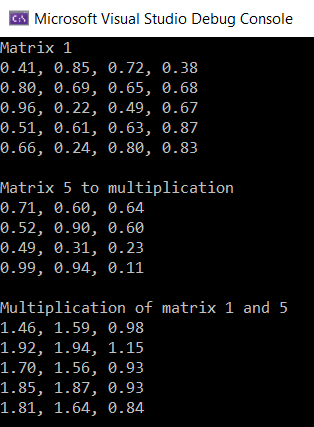
| cout << endl << "Matrix 5 to multiplication" << endl;  RandomTab(colCount, colCount2, tab5);  PrintTab(colCount, colCount2, tab5); |
| --- |

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) przechowująca 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.  
  


## 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.

### 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.

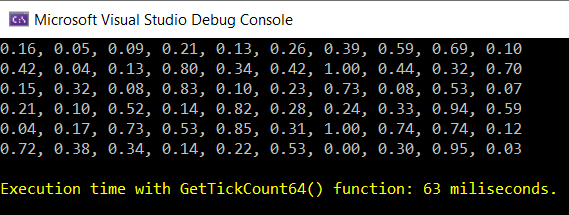
Bezpośrednio przed wywołaniem funkcji wypełniającej tablicę oraz bezpośrednio po funkcji wypisującej dane na ekran wywołuję funkcję GetTickCount64() na zmiennych lokalnych i zapisuję w nich wynik (zmierzony czas).

| 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.



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:

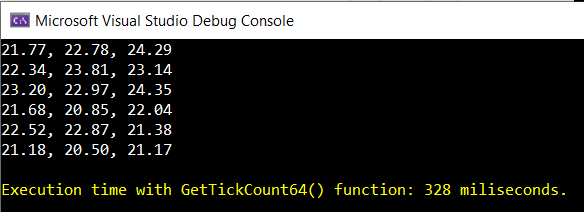
* + 62 ms
  + 78 ms
  + 47 ms
  + 62 ms
  + 31 ms
  + 47 ms
  + 47 ms

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**.



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.

### 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.

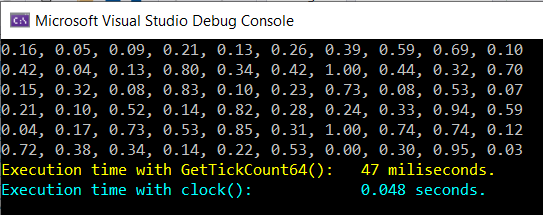
Bezpośrednio przed wywołaniem funkcji wypełniającej tablicę oraz bezpośrednio po wypisaniu danych na ekran wywołuję funkcję GetTickCount64() na zmiennych lokalnych i zapisuję w nich wynik (zmierzony czas).

| 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.



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

Kompletne wnioski znajdują się na końcu sprawozdania w sekcji [Wnioski](#_d0wsm6nzjy1y).

# 

# **Laboratoria nr 2**

## Wstęp

Informacja, ile czasu dany algorytm był wykonywany jest niewystarczający. 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(n2)
* sortowanie przez wstawianie (ang. insertion sort) O(n2)
* sortowanie przez scalanie (ang. merge sort) O(nlogn)
  + wymaga dodatkowej pamięci O(n)
* sortowanie przez zliczanie (ang. counting sort) O(n+k)
  + wymaga dodatkowej pamięci
* sortowanie kubełkowe (ang. bucket sort) O(n)
  + wymaga dodatkowej pamięci
* sortowanie biblioteczne (ang. library sort) O(nlogn)
* sortowanie przez wybieranie (ang. selection sort) O(n2)
* sortowanie Shella – (ang. shellsort) O(nlogn)
* sortowanie szybkie – (ang. quicksort) O(nlogn)
* sortowanie przez kopcowanie – (ang. heapsort) O(nlogn)

## 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).

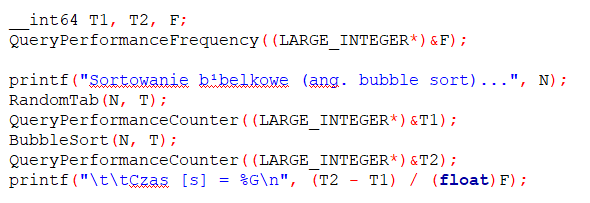
* **Funkcje obliczające czasy sortowania.**

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:

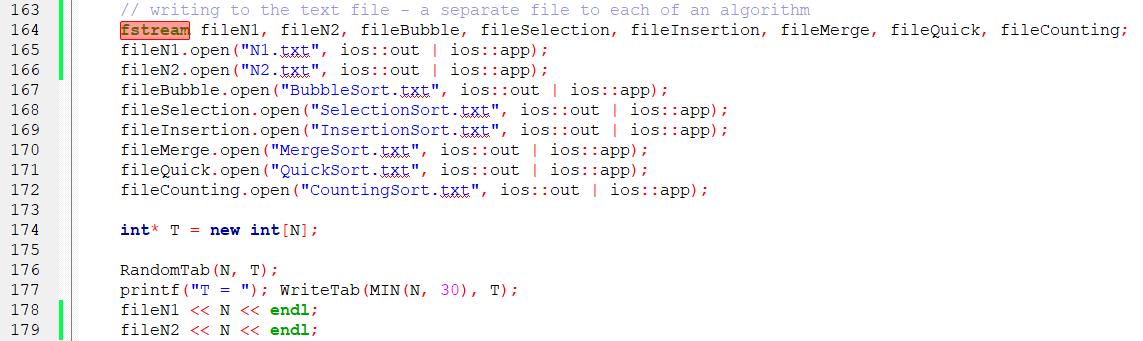


Liczba cykli wykonywanych przez procesor zwracana jest do zmiennej całkowitej 64-bitowej T1. Jeżeli odczytamy taki parametr (T1) tuż przed uruchomieniem danego algorytmu oraz tuż po jego wykonaniu (T2), to następnie różnica tych dwóch wielkości (T2-T1) da nam liczbę taktów procesora, które były potrzebne 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 arkuszu Google Sheets, pomiary czasów dla każdego z algorytmów zapisywałam w osobnym pliku (z opcją nadpisywania). Poniżej adekwatny wycinek kodu.



* **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 tablcy 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 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;  }  //--------------------------------------------------------------------------- |
| --- |

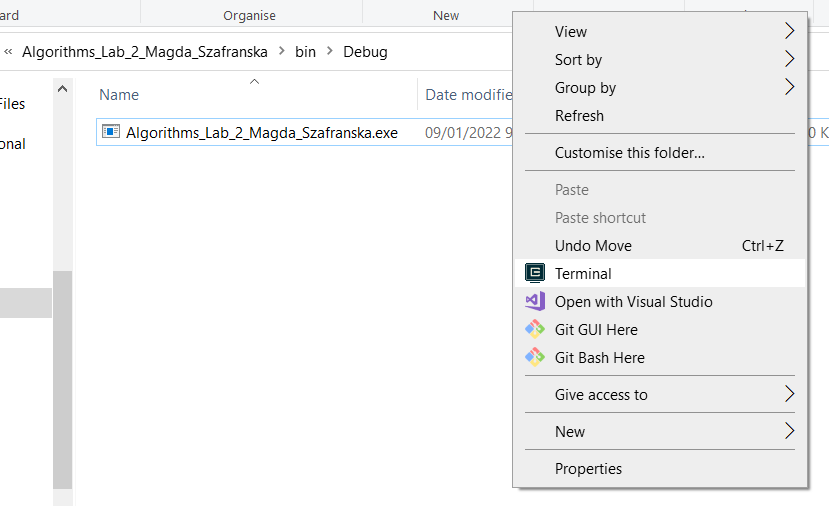
### Pomiar czasów wykonywania

* **Uruchomienie aplikacji**

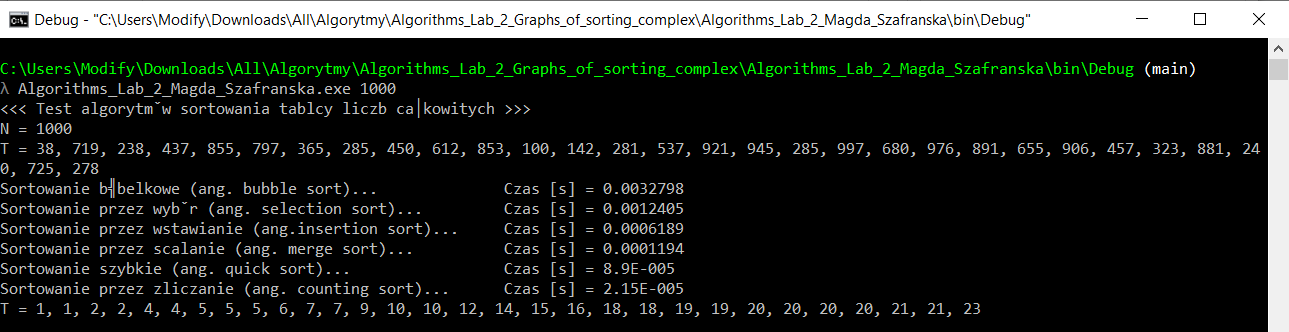
Aby uruchomić plik .exe przeszłam do folderu “*Debug*”, w którym się on znajdował:

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.ex*e” podając po odstępie po nazwie pliku przykładowy parametr 1000 określający wielkość tablicy do posortowania.



Kolejne pomiary znajdują się poniżej w drugiej części zadania.

## Wykonanie ćwiczenia cz. II: charakterystyki złożoności

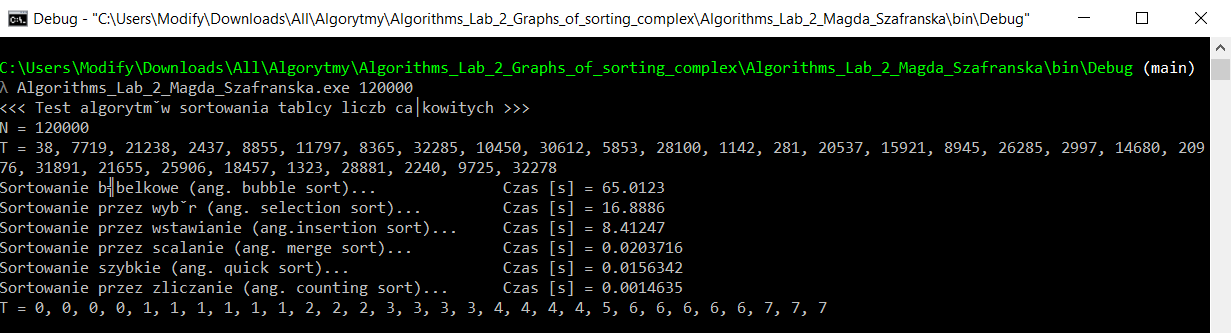
### Generowanie wyników aplikacji

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.

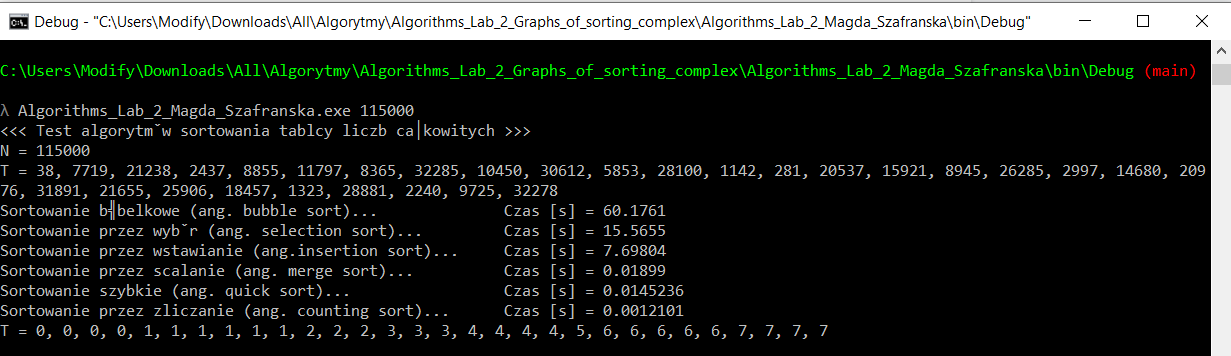
Pierwszy pomiar dla N = 1000 elementów okazał się za mały dlatego ponownie wywołałam program używając za każdym razem innej wielkości N uzależnionej od otrzymywanego czasu wykonywania dla algorytmu sortowania bąbelkowego.

Poniżej zrzuty ekranu kilku przykładowych wyników:

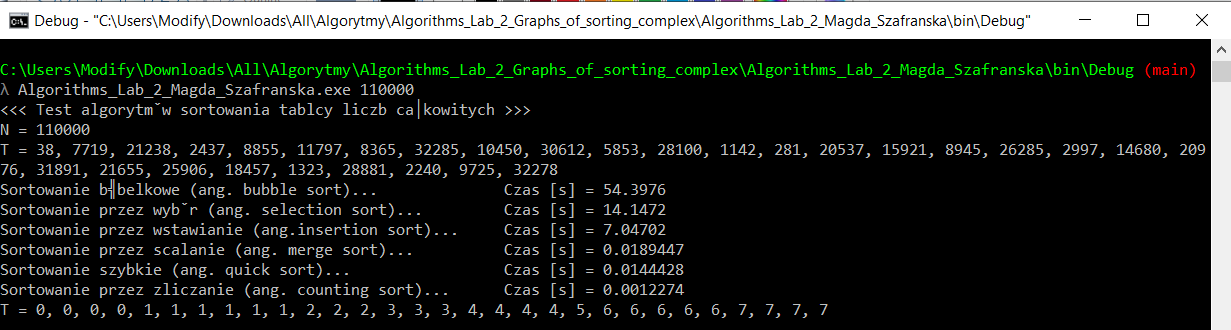
* dla N = 120 000, czas (sortowania bąbelkowego) =~ 65 s



* dla N = 115 000, czas =~ 60 s



* dla N = 110 000, czas =~ 54 s



### 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 krótszym.

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:

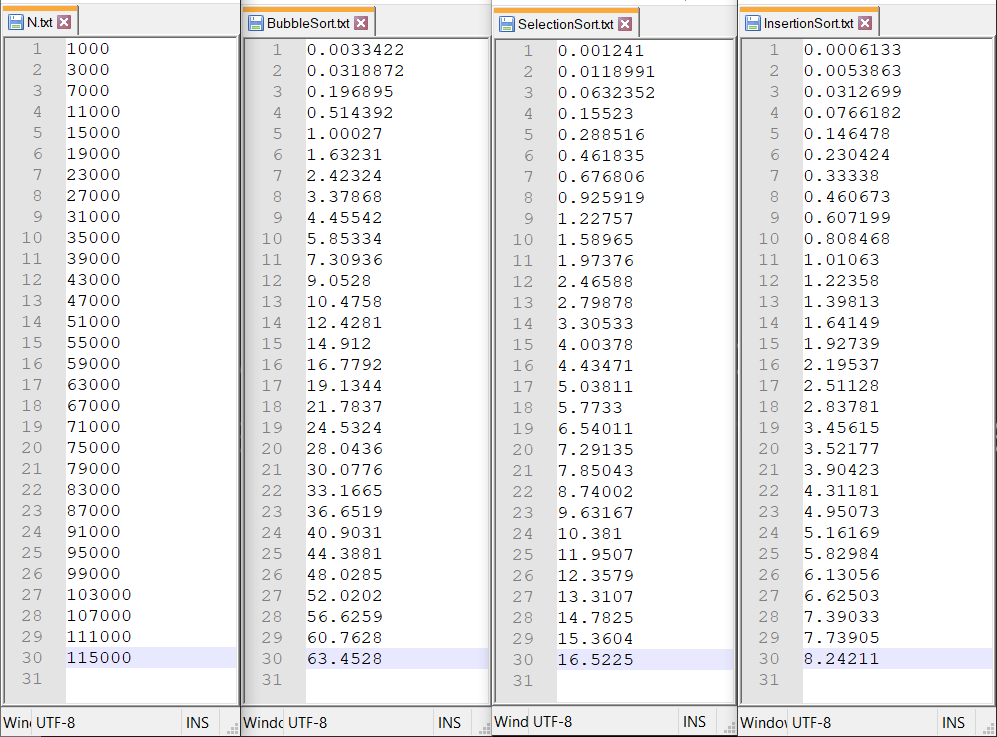
* jeden rysunek dla trzech pierwszych rodzajów sortowania:
  + sortowanie bąbelkowe (ang. bubble sort)
  + sortowanie przez wybór (ang. selection sort)
  + sortowanie przez wstawianie (ang.insertion sort)
* drugi dla trzech kolejnych (przez scalanie, szybkie, przez zliczanie)
  + sortowanie przez scalanie (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 przyjęłam 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.



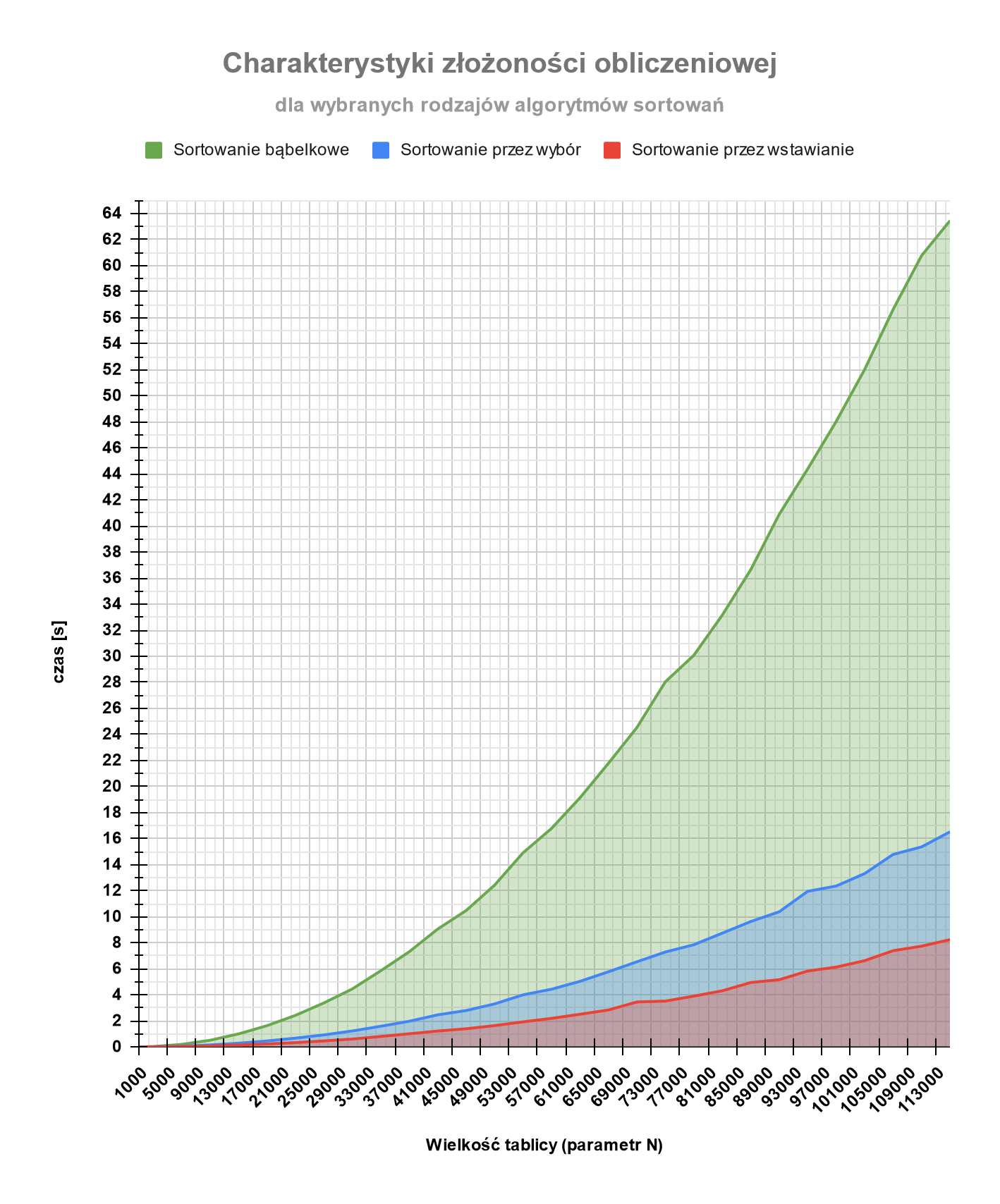
Uzyskane w wyniku zapisu do plików pomiary skopiowałam do odpowiednich kolumn w arkuszu Google Sheets.

| **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.

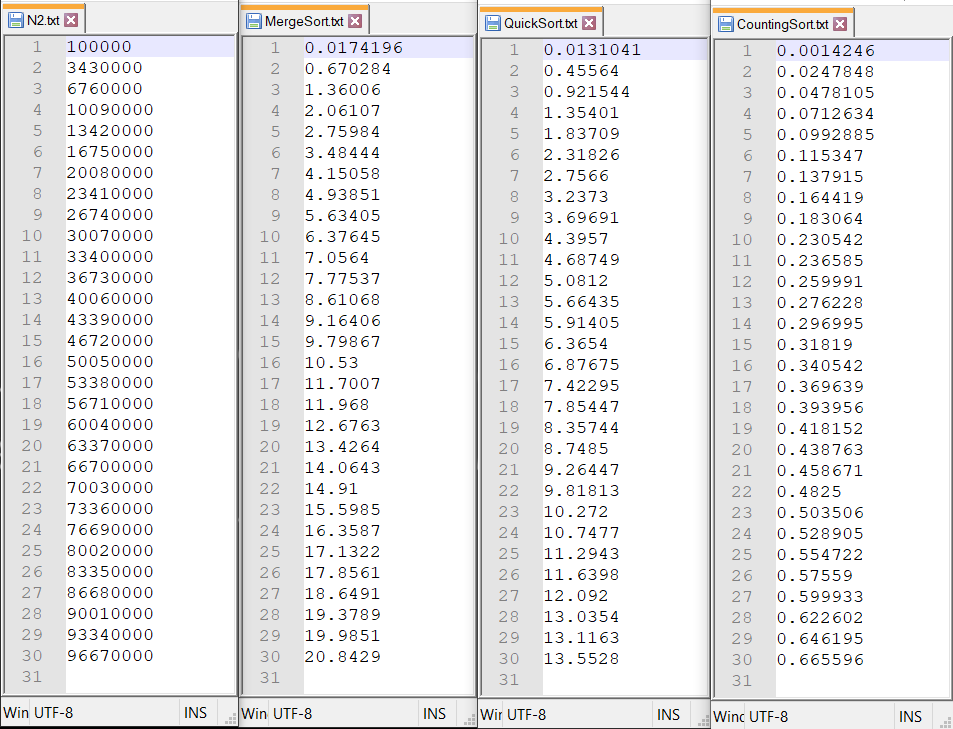


### Tworzenie wykresu dla sortowania przez scalanie, szybkiego i przez zliczanie

* 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.

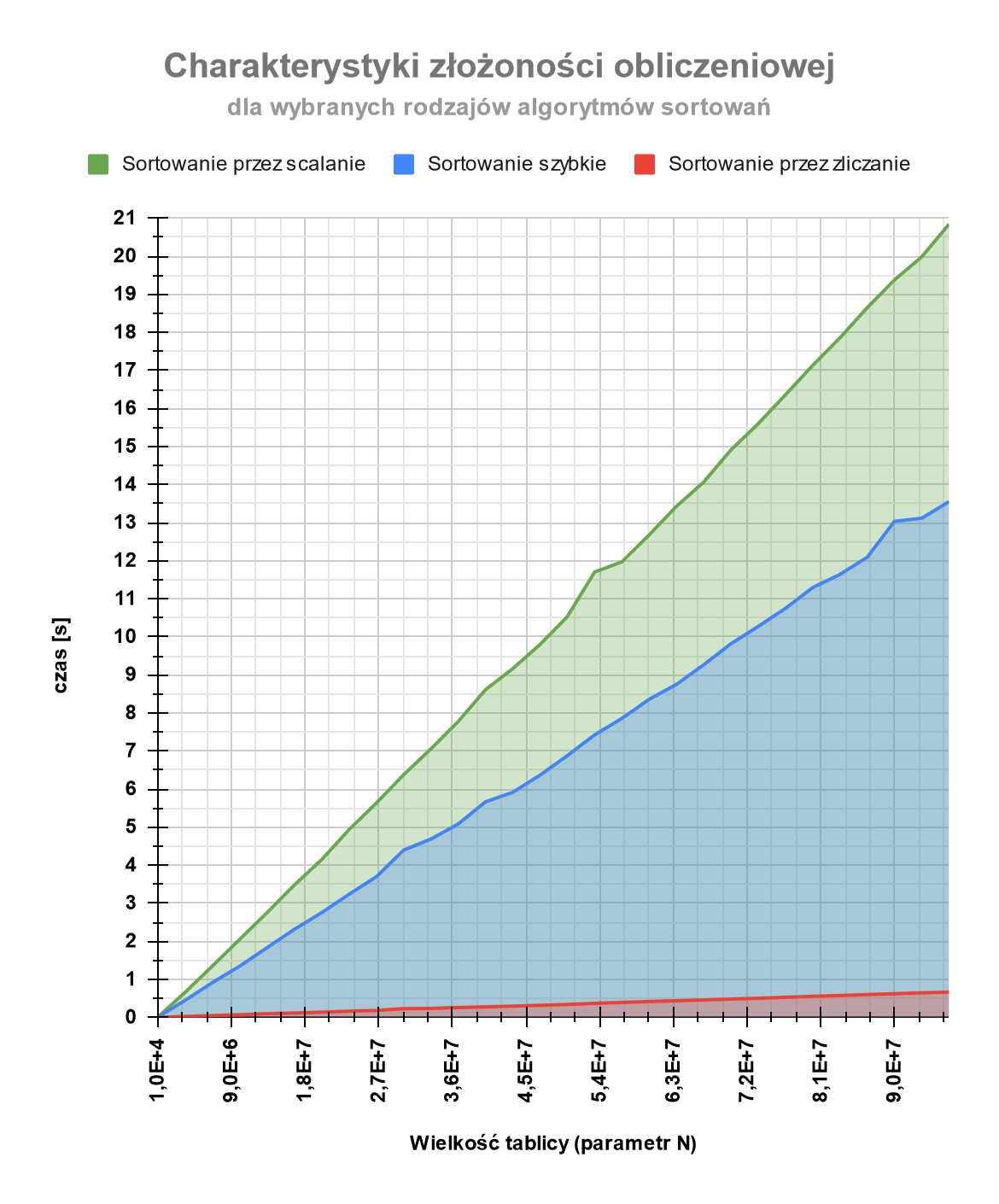


* **Pomiary użyte do wykresu.**

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).



* **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(n2)**
* Sortowanie przez wybór: **O(n2)**
* Sortowanie przez wstawianie: **O(n2)**
* Sortowanie przez scalanie: **O(n·log n)**
* Sortowanie szybkie: **O(n·log n)**
* Sortowanie przez zliczanie (ang. counting sort): **O(n)**

Kompletne wnioski po przeanalizowaniu charakterystyk znajdują się na końcu sprawozdania w sekcji [Wnioski](#_d0wsm6nzjy1y).

# 

# **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 Ο(n). Ta sama operacja dla listy wiązanej ma złożoność Ο(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

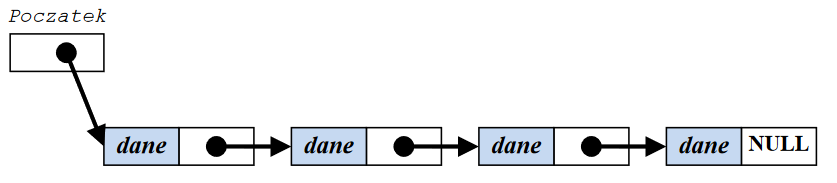
części: pola (lub kilku pól), w którym pamiętane są przechowywane elementy (dane

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 list pusta (tzn. nie zawiera

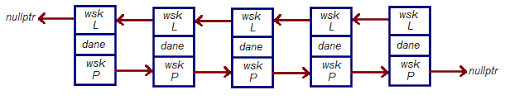
żadnych węzłów).



*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 żądanego indeksu. W przypadku listy dwukierunkowej węzły zawierają dwa wskaźniki. Operacje modyfikujące taką listę wymagają przepięcia 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 rozwiązania 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.

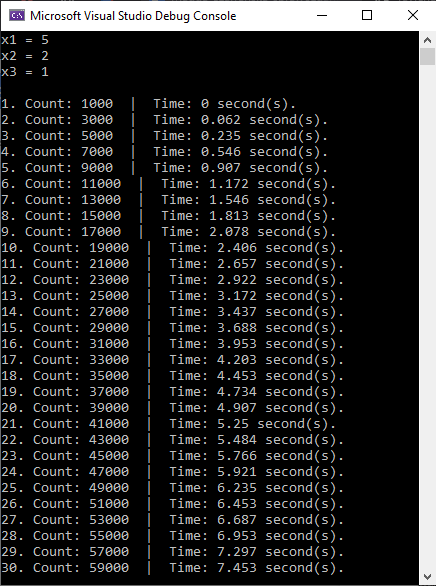
Cały kod programu znajduje się na końcu tego laboratorium. Dostępny jest także w repozytorium zdalnym (link powyżej).

* **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.

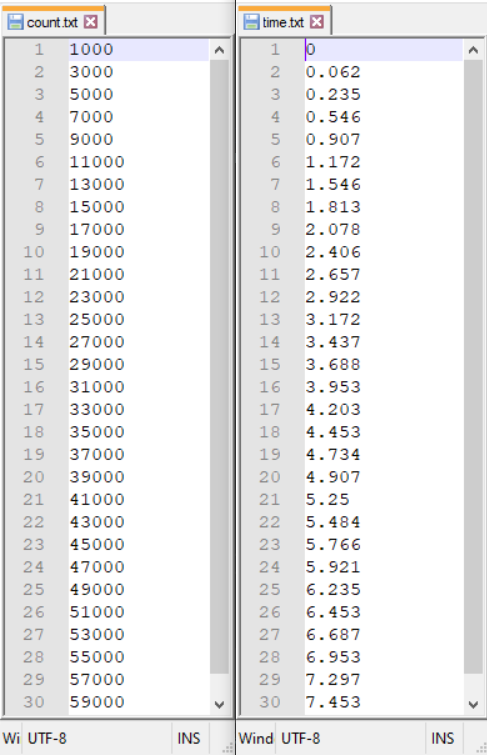
Bezpośrednio przed wywołaniem w funkcji main() kluczowej pętli “while” oraz bezpośrednio po pętli wywoływałam funkcję GetTickCount64() na zmiennych lokalnych i zapisywałam w nich za każdym razem bieżące wyniki wypełniania listy losowymi wartościami. Proces ten powtarzał się przez 30 iteracji zwiększając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

Poniżej zrzut ekranu z uzyskanymi pomiarami.

****

* **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.



* **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 |

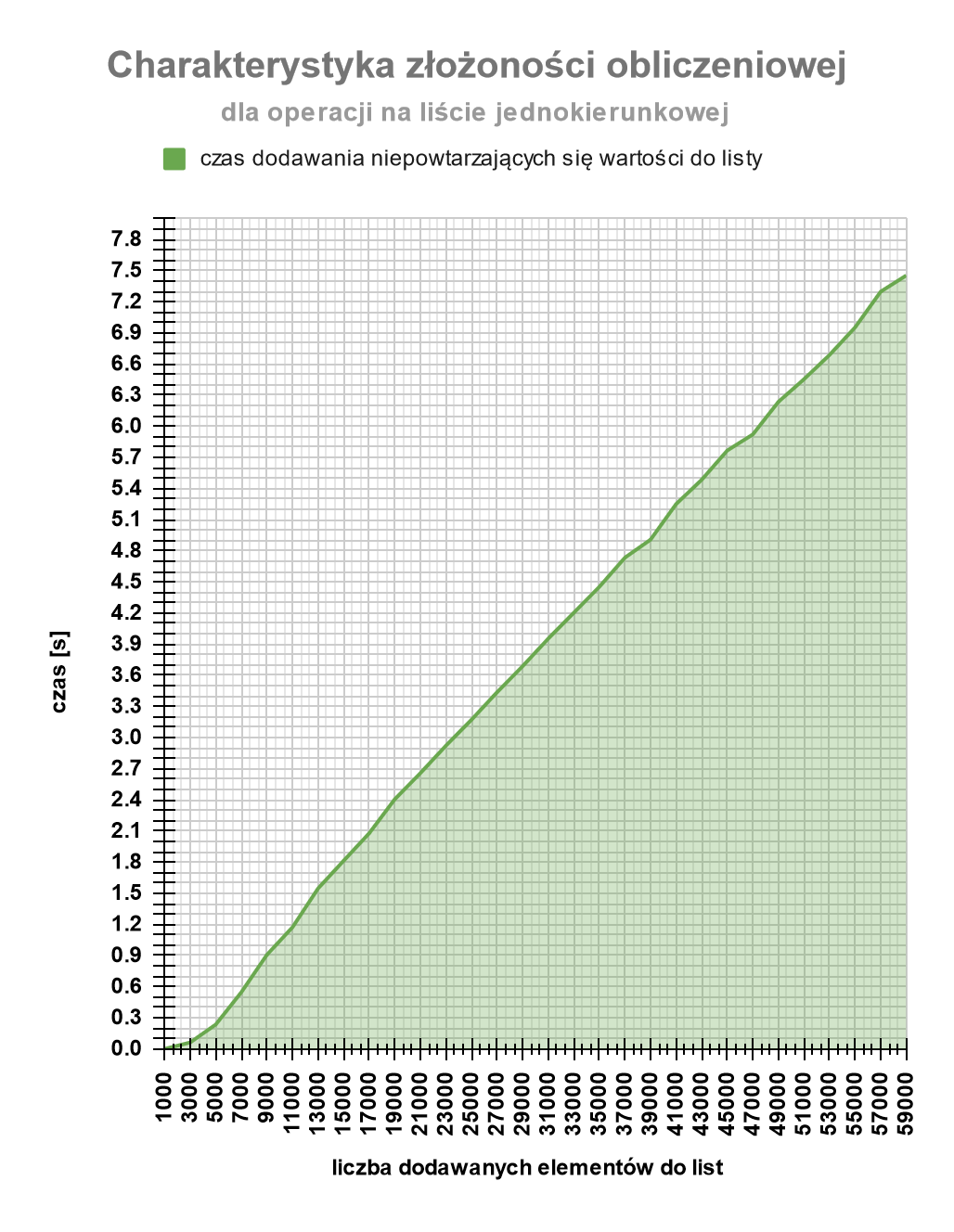
* **Dopasowanie skali osi.**

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**

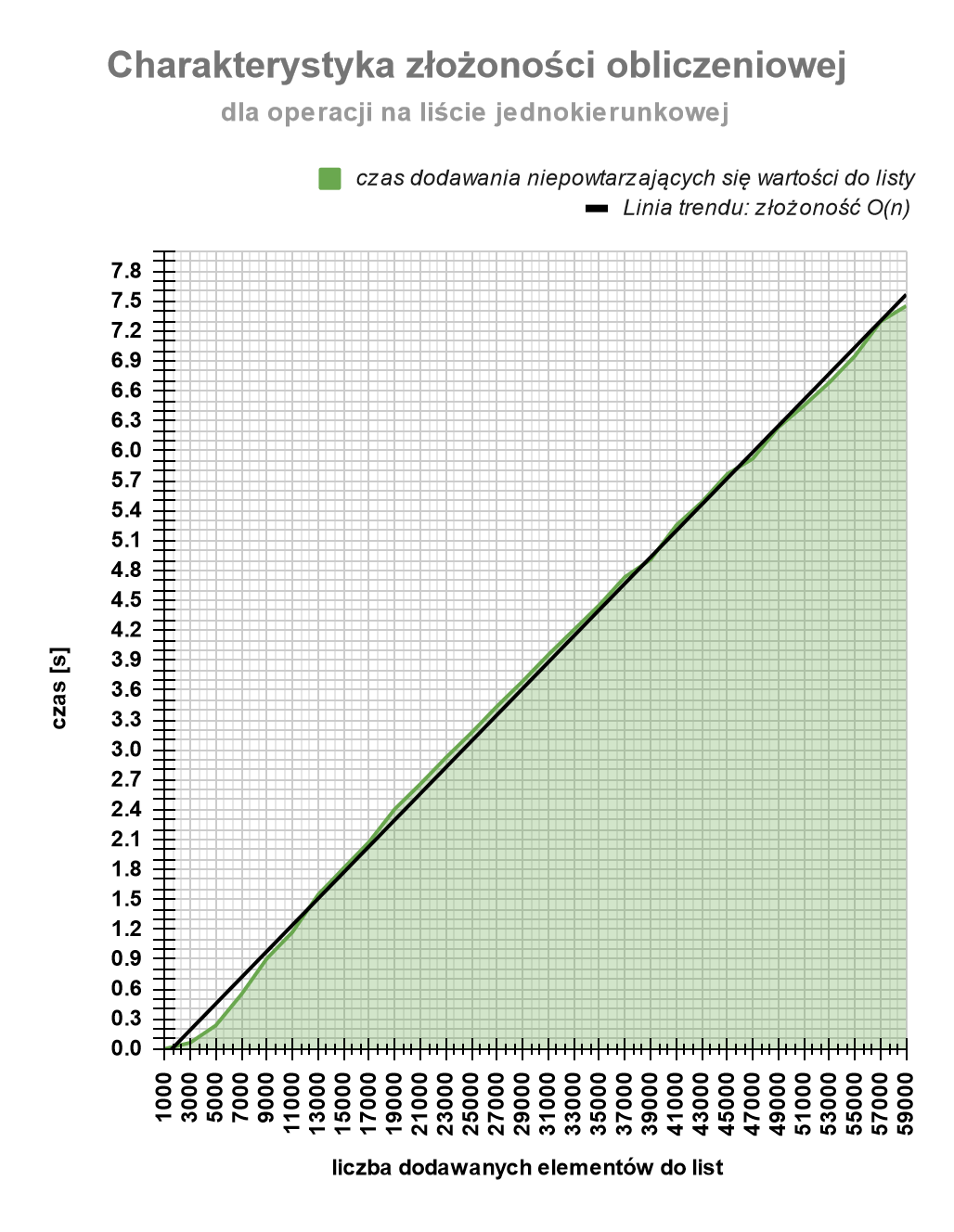
Na podstawie uzyskanych pomiarów stworzyłam wykres charakterystyki złożoności czasowej dodawania niepowtarzających się wartości do listy.

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



* **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 = 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

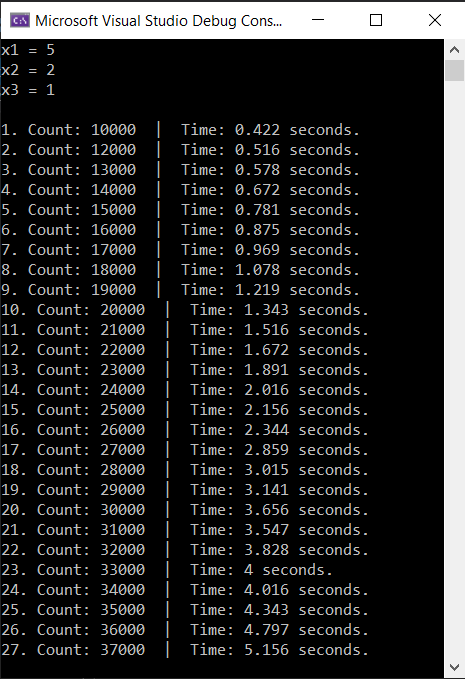
W drugiej części zajęć zajęliśmy się algorytmem dodającym elementy do listy dwukierunkowej i sumującym je. Celem było sprawdzenie jak szybko elementy te są iterowane, czas pracy algorytmu oraz jego złożoność czasowa.

Następnie należało wprowadzić pewne poprawki, aby poprawić tę złożoność.

### Algorytm sumujący elementy na liście - wersja podstawowa

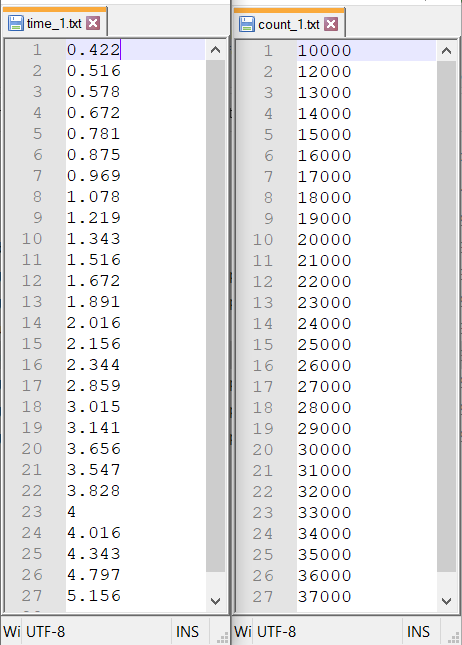
* **Obliczenie czasu wykonywania.**

Do pomiarów i stworzenia charakterystyki przyjęłam 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.



* **Implementacja zapisu pomiarów do plików**

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



* **Dane do wykresu charakterystyki złożoności.**

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

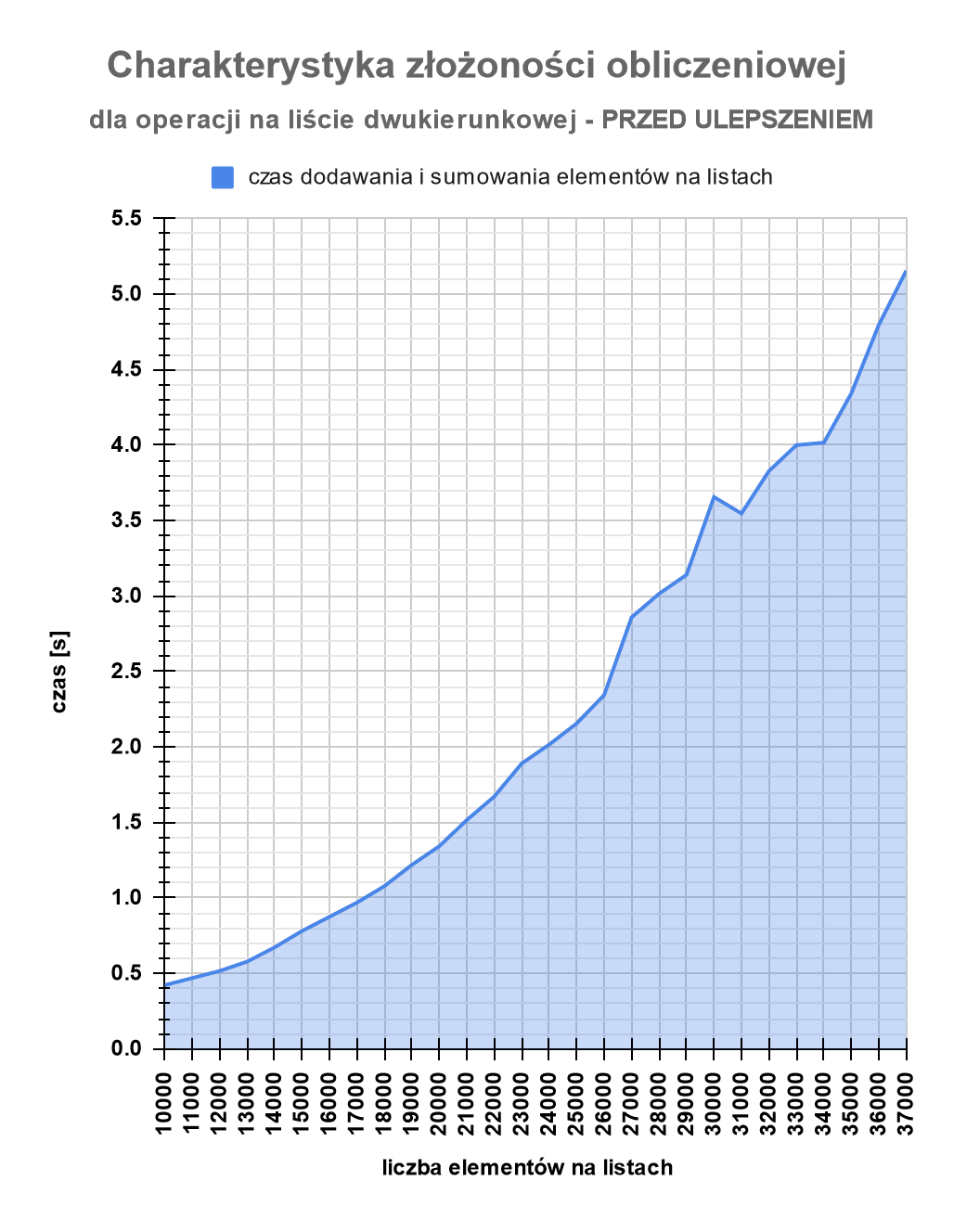
| **Count** | **Time [s] - before** |
| --- | --- |
| 10000 | 0.422 |
| 12000 | 0.516 |
| 13000 | 0.578 |
| 14000 | 0.672 |
| 15000 | 0.781 |
| 16000 | 0.875 |
| 17000 | 0.969 |
| 18000 | 1.078 |
| 19000 | 1.219 |
| 20000 | 1.343 |
| 21000 | 1.516 |
| 22000 | 1.672 |
| 23000 | 1.891 |
| 24000 | 2.016 |
| 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.**

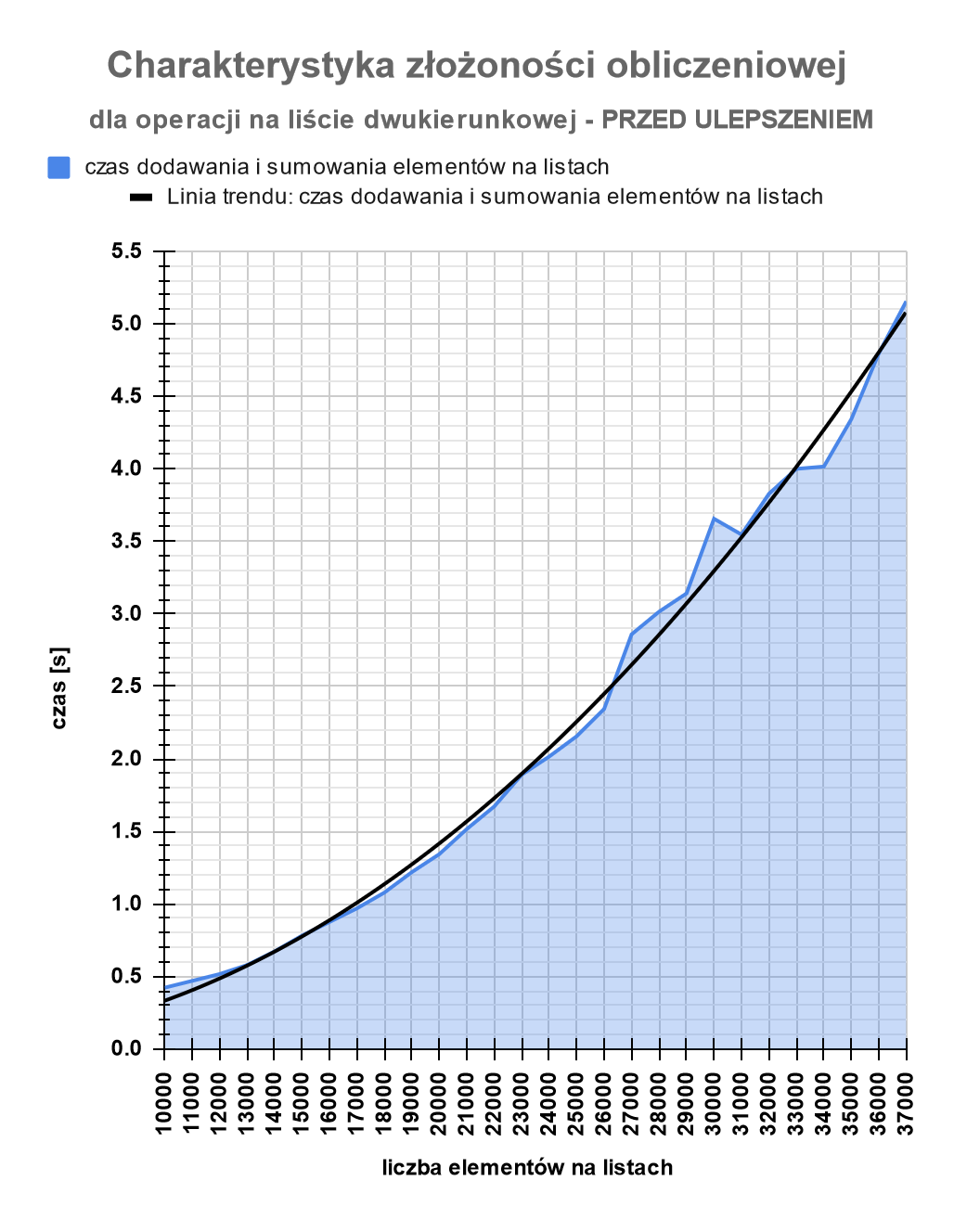
Maksymalna wartość na osi czasu to 5.5 sekund. Podzieliłam ją proporcjonalnie według skali liniowej na odcinki czasowe co 0.5 sekundy. Liczba tworzonych i sumowanych elementów na listach waha się w moich pomiarach od 10000 do 37000 (z krokiem co 1000), które są jednocześnie moimi skrajnymi wartościami.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



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

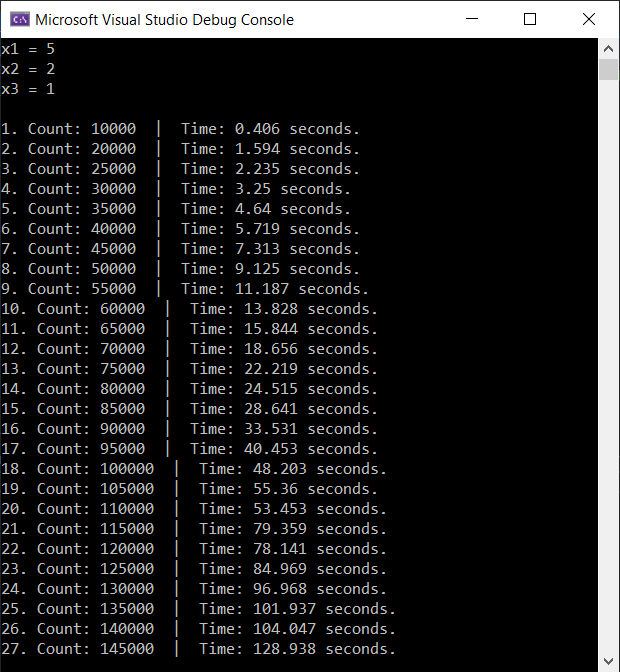


* **Kod programu.**

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.



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 |
| --- | --- |
|  |  |

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.

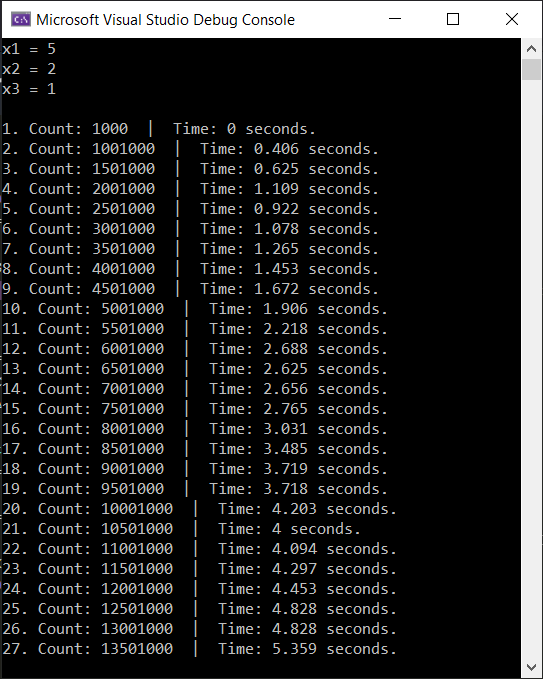
Po poprawieniu algorytmu, przy zachowaniu tych samych parametrów co uprzednio, jego złożoność zauważalnie zmalała. Pomiary ulepszonej wersji również zapisałam do plików (*time\_2.txt* oraz *count\_2.txt*), innych niż przed ulepszeniem - aby móc je następnie porównać niezależnie. Wyniki poniżej - przedstawione na identycznych parametrach.

| **PRZED** ulepszeniem | **PO** ulepszeniu |
| --- | --- |
|  |  |
| Wykres | Wykres |

* **Wersja ulepszona PRO**

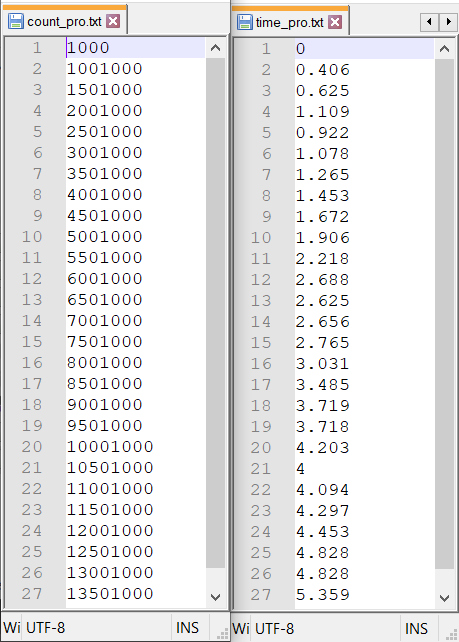
Zastosowanie tych samych parametrów spowodowało nieczytelność wykresu po ulepszeniu go fragmentami kodu. Algorytm po ulepszeniu jest o wiele efektywniejszy więc wprowadziłam o wiele większe ilości elementów dodawanych i sumowanych na listach.

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



* **Implementacja zapisu pomiarów do plików**

Zapisałam pomiary do plików z dopiskiem “*pro*”.



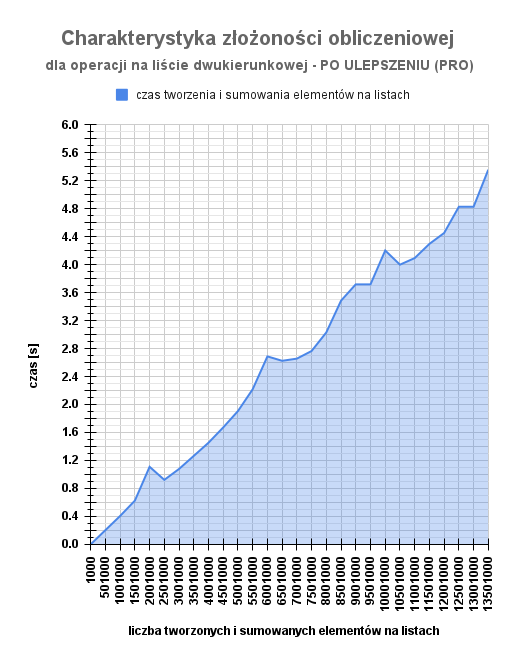
* **Dane do wykresu charakterystyki złożoności.**

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



* **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 = 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](#_d0wsm6nzjy1y).

# 

# **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.

#### Dot. laboratorium nr 3

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.