

**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

Wydział Studiów Strategicznych i Technicznych	0
Kierunek: Informatyka, rok II, semestr III (2021/2022)	0
Spis treści	1
Wstęp teoretyczny	2
Użyte technologie	2
Zastosowana konwencja w pisaniu kodu projektu	2
Pliki źródłowe	2
Cel laboratoriów	2
Laboratorium nr 1	3
Cel pośredni	3
Implementacja algorytmów	3
Create, RandomFill, Print, Delete	3
Adding, Subtraction	4
Copy	5
Trans	6
Multiplication	7
Pomiar czasów wykonywania	8
Funkcja GetTickCount()	8
Funkcja clock()	10
Laboratoria nr 2	11
Laboratoria nr 3	11
Wnioski	11

Wstęp teoretyczny

aa

Użyte technologie

Na wszystkich laboratoriach korzystałam z tego samego środowiska programistycznego oraz pozostałych narzędzi:

- Microsoft Visual Studio Community 2019 (wersja 16.11.1)
- C++
- GIT
- Github
- Git Bash
- Google Documents

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

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.

Dodatkowo implementacja kodu w języku C++ pozwoliła go utrwalić i odkryć swoje detale tego języka programistycznego.

Laboratorium nr 1

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 pośredni

Pliki projektu znajdują się w publicznym repozytorium zdalnym Gita w Githubie. Link do niego: 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** tab1 = 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.

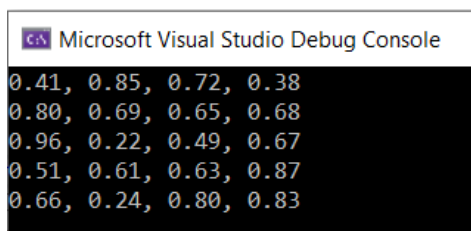
```
RandomTab(rowCount, colCount, tab1);
PrintTab(rowCount, colCount, tab1);
```

```
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.



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

2. Adding, Subtraction

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

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

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

- Utworzyłam prototypy funkcji dodawania i odejmowania podane w deklaracji.

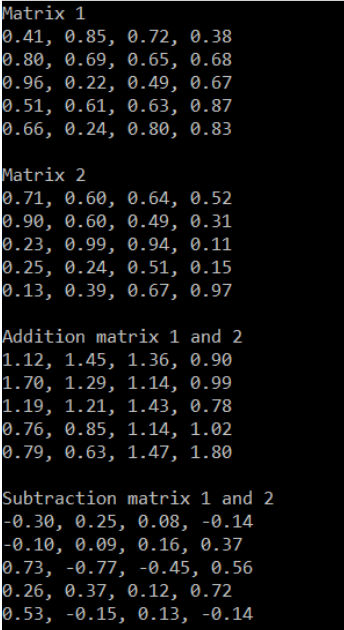
```
void AddTab(int rowCount, int colCount, double** tab1, double** tab2,  
double** tab3);  
void SubtractTab(int rowCount, int colCount, double** tab1, double** tab2,  
double** tab3);
```

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

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

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

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



```
Microsoft Visual Studio Debug Console

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

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

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

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

3. Copy

Kolejnym krokiem było kopiowanie macierzy.

- W funkcji main() stworzyłam kolejną tablicę, która będzie przechowywać skopiowane wartości z innej tablicy.

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

- Utworzyłam prototyp funkcji kopiującej podany w deklaracji.

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

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

- Wywołałam funkcję przechowując wynik operacji kopiowania macierzy tab1 w tab4.

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

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

4. Trans

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

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

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

- Utworzyłam prototyp funkcji kopiującej podany w deklaracji.


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

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

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

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

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

 Microsoft Visual Studio Debug Console

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

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

5. Multiplication

Aby pomnożyć dwie macierze muszą one spełniać warunek, iż ilość kolumn pierwszej macierzy jest równa ilości wierszy drugiej. Macierzą wynikową będzie macierz o ilości

rzędów takiej jak pierwsza z mnożonych macierzy oraz z ilością kolumn taką samą jak druga mnożona macierz.

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

3 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);
```

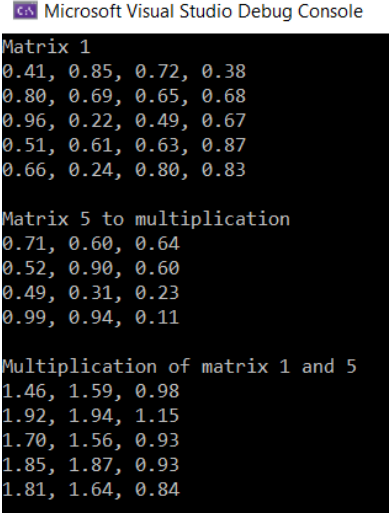
Wywołałam funkcję podając za argumenty kolejno zgodnie z definicją:

- 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 zmieniłam kolor wyświetlanego wewnątrz komunikatu na żółty.

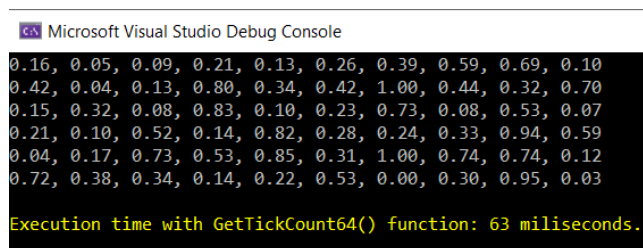
- Bezpośrednio przed wywołaniem funkcji wypełniającą tablicę oraz bezpośrednio po funkcji wypisującej dane na ekran wywołuję funkcję GetTickCount64() na zmiennych lokalnych i zapisuję w nich wynik (zmierzony czas).

```
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

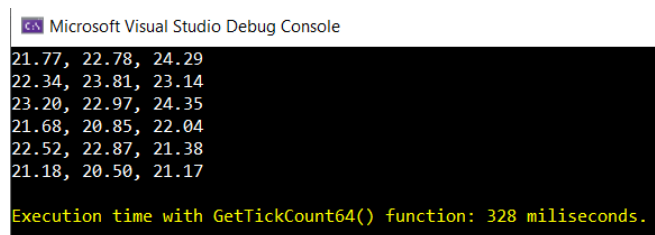
- 31 ms
- 47 ms
- 47 ms

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.

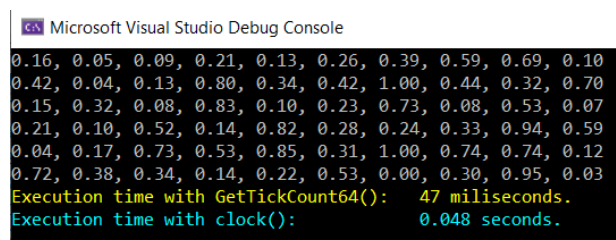
- Bezpośrednio przed wywołaniem funkcji wypełniającą tablicę oraz bezpośrednio po wypisaniu danych na ekran wywołuję funkcję GetTickCount64() na zmiennych lokalnych i zapisuję w nich wynik (zmierzony czas).

```
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.

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

Laboratoria nr 2

Cel pośredni

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

Aa

Wykonanie ćwiczenia cz. I: implementacja algorytmów

Subpart 1

Wykonanie ćwiczenia cz. II: wykresy

Subpart 1

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

Laboratoria nr 3

Cel pośredni

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

Aa

Wykonanie ćwiczenia cz. I: implementacja algorytmów

Subpart 1

Wykonanie ćwiczenia cz. II: wykresy

Subpart 1

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.