



AGH

Ćwiczenie 1

Algorytmy macierzowe

Autorzy:

Gabriel Kaźmierczak

Dariusz Piwowarski

Spis treści

1	Implementacja mnożenia macierzy metodą rekurencyjną	3
1.1	Pseudokod algorytmu rekurencyjnego	3
1.2	Fragmenty kodu programu	5
2	Testy implementacji	7
2.1	Procedura testowa	7
2.2	Test poprawności działania algorytmu	7
2.3	Pomiar czasu wykonania	8
2.4	Pomiar ilości operacji zmiennoprzecinkowych	9
2.5	Szacowana złożoność obliczeniowa	10

1 Implementacja mnożenia macierzy metodą rekurencyjną

1.1 Pseudokod algorytmu rekurencyjnego

Procedura *split*(**A**):

- Wejście: Macierz kwadratowa **A**
- Jeśli przetwarzana macierz jest wymiaru 1x1:
 - Zwróć **A**
- W przeciwnym przypadku:
 - Podziel macierz na ćwiartki **A**₁₁, **A**₁₂, **A**₂₁, **A**₂₂
 - Zwróć **A**₁₁, **A**₁₂, **A**₂₁, **A**₂₂

Procedura *binet*(**A**,**B**):

- Wejście: Macierze kwadratowe **A**, **B**
- Jeśli przetwarzane macierze są wymiaru 1x1:
 - Zwróć wynik **A**_{1,1} · **B**_{1,1}
- W przeciwnym wypadku:
 - Wykonaj procedury *split*(**A**) oraz *split*(**B**)
 - Wykonaj procedury *binet*(**A**₁₁, **B**₁₁), *binet*(**A**₁₁, **B**₁₂), *binet*(**A**₁₂, **B**₁₁), *binet*(**A**₁₂, **B**₁₂), *binet*(**A**₂₁, **B**₁₁), *binet*(**A**₂₁, **B**₁₂), *binet*(**A**₂₂, **B**₁₁), *binet*(**A**₂₂, **B**₁₂), a następnie zsumuj wyniki dla każdej z grup:
 - * **A**₁₁, **B**₁₁ oraz **A**₁₂, **B**₂₁
 - * **A**₁₁, **B**₁₂ oraz **A**₁₂, **B**₂₂
 - * **A**₂₁, **B**₁₁ oraz **A**₂₂, **B**₂₁
 - * **A**₂₁, **B**₁₂ oraz **A**₂₂, **B**₂₂
 - Połącz uzyskane macierze w nową macierz **C**
 - Zwróć **C**

Procedura *strassen*(**A**,**B**):

- Wejście: Macierze kwadratowe **A**, **B**
- Jeśli przetwarzane macierze są wymiaru 1x1:
 - Zwróć wynik $\mathbf{A}_{1,1} \cdot \mathbf{B}_{1,1}$
- W przeciwnym wypadku:
 - Wykonaj procedury *split*(**A**) oraz *split*(**B**)
 - Wykonaj dodawanie oraz procedury *strassen*() dla każdej z grup:
 - * $A_{11} + A_{22}, B_{11} + B_{22}$
 - * $A_{21} + A_{22}, B_{11}$
 - * $A_{11}, B_{12} - B_{22}$
 - * $A_{22}, B_{21} - B_{11}$
 - * $A_{11} + A_{12}, B_{22}$
 - * $A_{21} - A_{11}, B_{11} + B_{12}$
 - * $A_{12} - A_{22}, B_{21} + B_{22}$
 - Na uzyskane macierze $M_1 \dots M_7$ wykonaj następujące operacje:
 - * $M_1 + M_4 - M_5 + M_7$
 - * $M_3 + M_5$
 - * $M_2 + M_4$
 - * $M_1 - M_2 + M_3 + M_6$
 - Połącz uzyskane macierze w nową macierz **C**

1.2 Fragmenty kodu programu

```

1 def binet_with_count(A: np.ndarray, B: np.ndarray):
2     if A.shape == (1, 1):
3         return A * B, 1
4
5     A11, A12, A21, A22 = split_matrix(A)
6     B11, B12, B21, B22 = split_matrix(B)
7
8     C11a, count1a = binet_with_count(A11, B11)
9     C11b, count1b = binet_with_count(A12, B21)
10    C12a, count2a = binet_with_count(A11, B12)
11    C12b, count2b = binet_with_count(A12, B22)
12    C21a, count3a = binet_with_count(A21, B11)
13    C21b, count3b = binet_with_count(A22, B21)
14    C22a, count4a = binet_with_count(A21, B12)
15    C22b, count4b = binet_with_count(A22, B22)
16    C1 = C11a + C11b
17    C2 = C12a + C12b
18    C3 = C21a + C21b
19    C4 = C22a + C22b
20
21    count = count1a + count1b + count2a + count2b + count3a + count3b +
22    count4a + count4b + 4 * math.prod(A.shape)
23
24    return np.vstack((np.hstack((C1, C2)), np.hstack((C3, C4)))), count

```

Fragment 1: Metoda *binet_with_count* implementująca algorytm Bineta

```

1 def strassen_with_count(A: np.ndarray, B: np.ndarray):
2     if A.shape == (1, 1):
3         return A * B, 1
4
5     A11, A12, A21, A22 = split_matrix(A)
6     B11, B12, B21, B22 = split_matrix(B)
7
8     M1, count1 = strassen_with_count(A11 + A22, B11 + B22)
9     M2, count2 = strassen_with_count(A21 + A22, B11)
10    M3, count3 = strassen_with_count(A11, B12 - B22)
11    M4, count4 = strassen_with_count(A22, B21 - B11)
12    M5, count5 = strassen_with_count(A11 + A12, B22)
13    M6, count6 = strassen_with_count(A21 - A11, B11 + B12)
14    M7, count7 = strassen_with_count(A12 - A22, B21 + B22)
15
16    count = count1 + count2 + count3 + count4 + count5 + count6 + count7 +
17    18 * math.prod(A.shape)
18
19    return np.vstack((np.hstack((M1 + M4 - M5 + M7, M3 + M5)), np.hstack((
20    M2 + M4, M1 - M2 + M3 + M6)))), count

```

Fragment 2: Metoda *strassen_with_count* implementująca algorytm Strassena

```
1 def split_matrix(M: np.ndarray):  
2     n = M.shape[0] // 2  
3     return M[:n, :n], M[:n, n:], M[n:, :n], M[n:, n:]
```

Fragment 3: Metoda *split_matrix* dzieląca macierz

```
1 def test_for_random_2_to_power_k(k, with_count=False):  
2     n = 2 ** k  
3     A = np.random.rand(n, n)  
4     B = np.random.rand(n, n)  
5  
6     if with_count:  
7         C, count = binet_with_count(A, B)  
8         return count  
9  
10    else:  
11        start = perf_counter_ns()  
12        C = binet(A, B)  
13        return perf_counter_ns() - start  
14  
15 if __name__ == '__main__':  
16     max_k=9  
17     x = np.arange(1, max_k + 1)  
18     y1 = [test_for_random_2_to_power_k(k) for k in x]  
19     y2 = [test_for_random_2_to_power_k(k, with_count=True) for k in x]
```

Fragment 4: Fragment skryptu testującego implementację

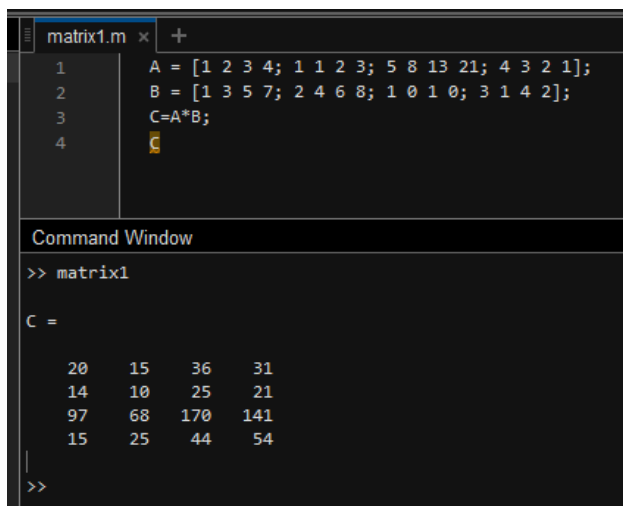
2 Testy implementacji

2.1 Procedura testowa

Dla każdego z implementowanych algorytmów wykonano test poprawności działania algorytmu na przykładowych macierzach 4x4, a następnie przeprowadzono pomiar wykonanych operacji zmiennoprzecinkowych i czasu wykonania w zależności od rozmiaru macierzy. Wyniki testów zostały zebrane i przedstawione na załączonych wykresach.

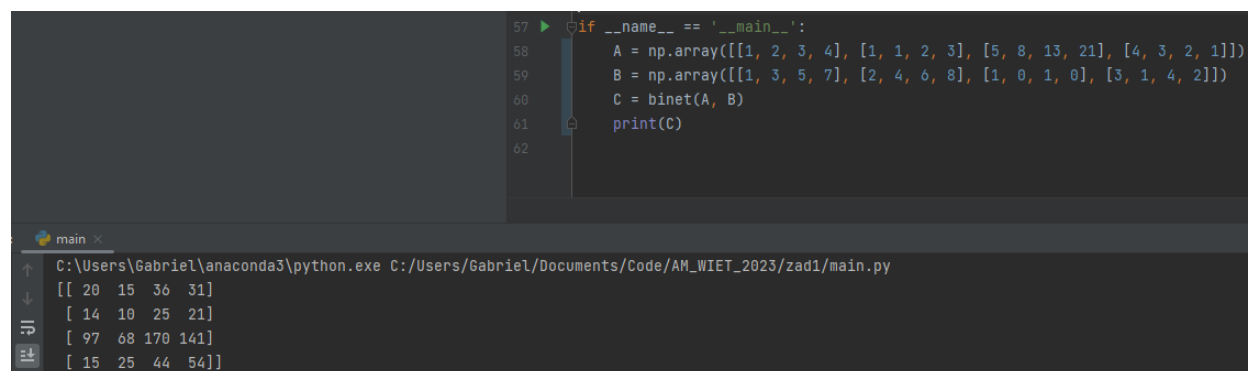
2.2 Test poprawności działania algorytmu

W celu zbadania poprawności działania algorytmów porównaliśmy wyniki uzyskane w wyniku mnożenia dwóch macierzy 4x4 przy pomocy naszej implementacji z wynikiem uzyskanym w programie MATLAB.



```
matrix1.m x +
1 A = [1 2 3 4; 1 1 2 3; 5 8 13 21; 4 3 2 1];
2 B = [1 3 5 7; 2 4 6 8; 1 0 1 0; 3 1 4 2];
3 C=A*B;
4
Command Window
>> matrix1
C =
    20    15    36    31
    14    10    25    21
    97    68   170   141
    15    25    44    54
>>
```

Rysunek 1: Wynik uzyskany z programu MATLAB



```
57 if __name__ == '__main__':
58     A = np.array([[1, 2, 3, 4], [1, 1, 2, 3], [5, 8, 13, 21], [4, 3, 2, 1]])
59     B = np.array([[1, 3, 5, 7], [2, 4, 6, 8], [1, 0, 1, 0], [3, 1, 4, 2]])
60     C = binet(A, B)
61     print(C)
62
```

```
main x
C:\Users\Gabriel\anaconda3\python.exe C:/Users/Gabriel/Documents/Code/AM_WIET_2023/zad1/main.py
[[ 20  15  36  31]
 [ 14  10  25  21]
 [ 97  68 170 141]
 [ 15  25  44  54]]
```

Rysunek 2: Wynik uzyskany przy użyciu implementacji algorytmu Bineta

```
57 if __name__ == '__main__':  
58     A = np.array([[1, 2, 3, 4], [1, 1, 2, 3], [5, 8, 13, 21], [4, 3, 2, 1]])  
59     B = np.array([[1, 3, 5, 7], [2, 4, 6, 8], [1, 0, 1, 0], [3, 1, 4, 2]])  
60     C = strassen(A, B)  
61     print(C)  
62
```

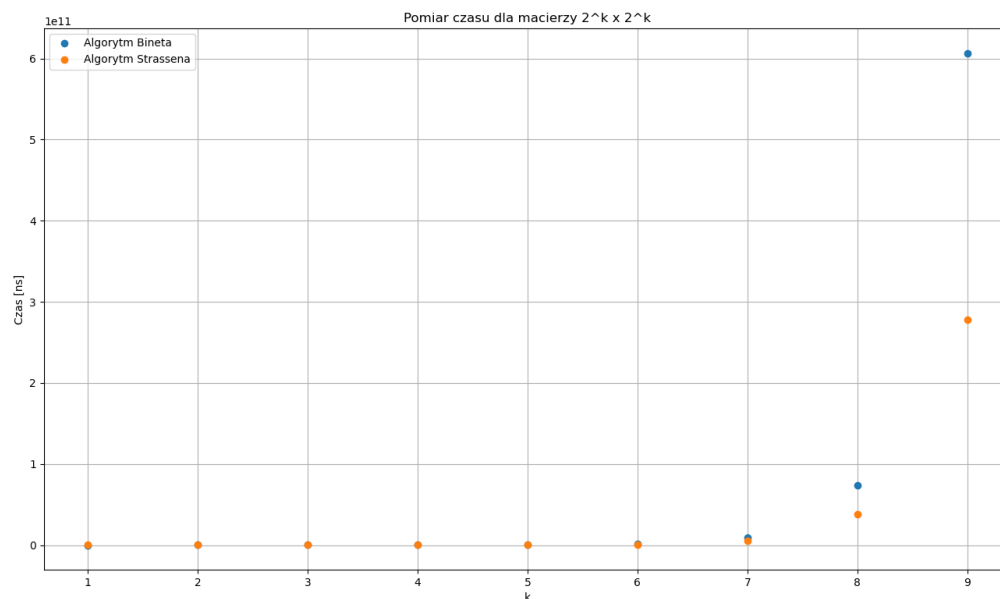
main x

```
C:\Users\Gabriel\anaconda3\python.exe C:/Users/Gabriel/Documents/Code/AM_WIET_2023/zad1/main.py  
[[ 20  15  36  31]  
 [ 14  10  25  21]  
 [ 97  68 170 141]  
 [ 15  25  44  54]]
```

Rysunek 3: Wynik uzyskany przy użyciu implementacji algorytmu Strassena

2.3 Pomiar czasu wykonania

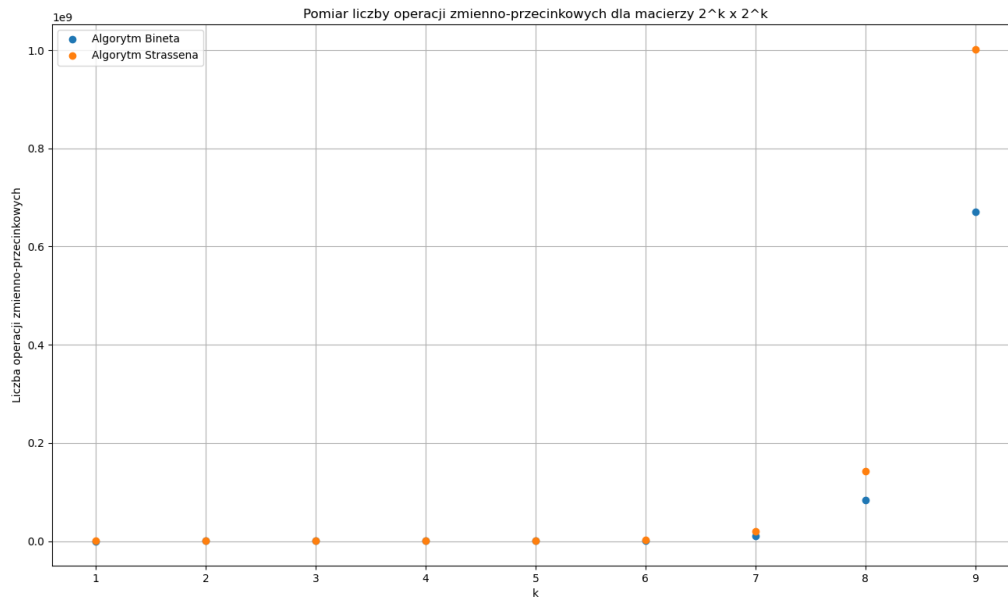
Do pomiaru czasu wykonania programu wykorzystano metodę `perf_counter_ns()` z pakietu `time` ze standardowej biblioteki języka Python, która pozwala na pomiar czasu z dokładnością do nanosekund w oparciu o systemowy zegar wysokiej precyzji. Na poniższym wykresie przedstawiono zależność czasu wykonania od rozmiaru macierzy. Ze względu na wysoki czas wymagany do przeprowadzenia obliczeń, testy wykonano dla macierzy o rozmiarach od 2×2 do $2^9 \times 2^9$.



Rysunek 4: Pomiar czasu wykonania

2.4 Pomiar ilości operacji zmiennoprzecinkowych

Zaimplementowane przez nas metody `strassen_with_count()` oraz `binet_with_count()` po zakończeniu działania zwracają liczbę wykonanych operacji zmiennoprzecinkowych. Wyniki zostały przedstawione na poniższym wykresie.

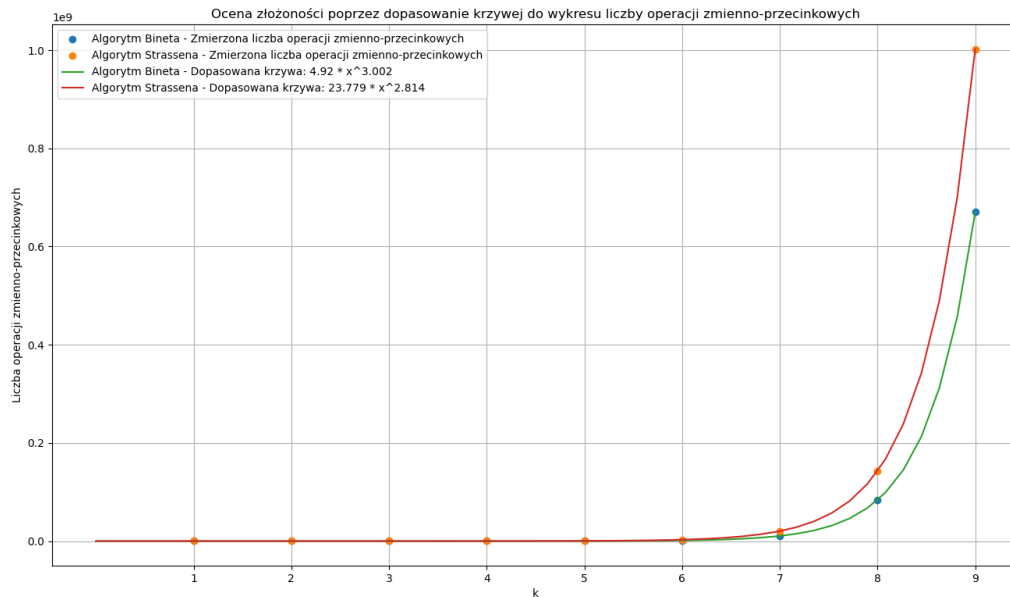


Rysunek 5: Pomiar ilości operacji zmiennoprzecinkowych

Analizując uzyskane dane możemy zauważyć, że pomimo większej ilości operacji, algorytm Strassena wymaga mniej czasu celem uzyskania wyniku dzięki mniejszej ilości rekurencyjnych wywołań (7 zamiast 8).

2.5 Szacowana złożoność obliczeniowa

Na podstawie uzyskanych danych ilości operacji zmiennoprzecinkowych, dokonaliśmy oszacowania złożoności obliczeniowej algorytmów Strassena oraz Bineta. Użyliśmy do tego funkcji `curve_fit()` z biblioteki `scipy` dokładnie z modułu `optimize`.



Rysunek 6: Ocena złożoności poprzez dopasowanie krzywej

Jak widać na wykresie dopasowane krzywe to:

- $4.92x^{3.002}$ dla algorytmu Bineta, co zgadza się z teoretyczną złożonością $O(n^3)$
- $23.779x^{2.814}$ dla algorytmu Strassena, co również odpowiada złożoności $O(n^{2.81})$ wyznaczonej teoretycznie.

Opis i wyznaczenie teoretycznej złożoności: https://pl.wikipedia.org/wiki/Algorytm_Strassena

Podsumowując, oszacowana na podstawie wyników złożoność naszej implementacji algorytmów, zgadza się ze złożonością teoretyczną, co świadczy o poprawności implementacji.