Algorytmy Macierzowe
Sprawozdanie 1
Rekurencyjne mnożenie macierzy

Szymon Paszkiewicz
Przemysław Węglik

6 listopada 2022

Spis treści

1 Algorytm Binét'a

1.1 Opis algorytmu

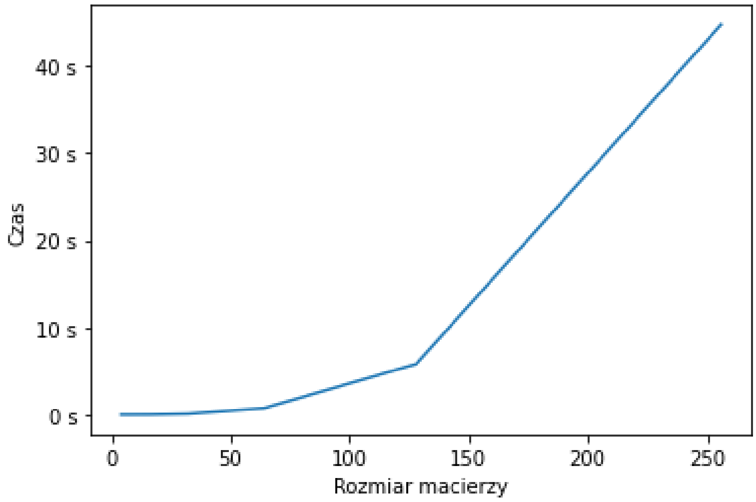
Polega na rekurencyjnym rozbijaniu macierzy na 4 mniejsze i obliczaniu wyników dla tych podproblemów. Tam ponownie będziemy musieli użyć mnożenia i wykorzystamy procedurę. To klasyczne podejście nosi nazwę "divide and conquer". Stosujemy wzór:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} (A_{11}B_{11} + A_{12}B_{21}) & (A_{11}B_{21} + A_{12}B_{22}) \\ (A_{21}B_{11} + A_{22}B_{21}) & (A_{21}B_{12} + A_{22}B_{22}) \end{bmatrix}$$

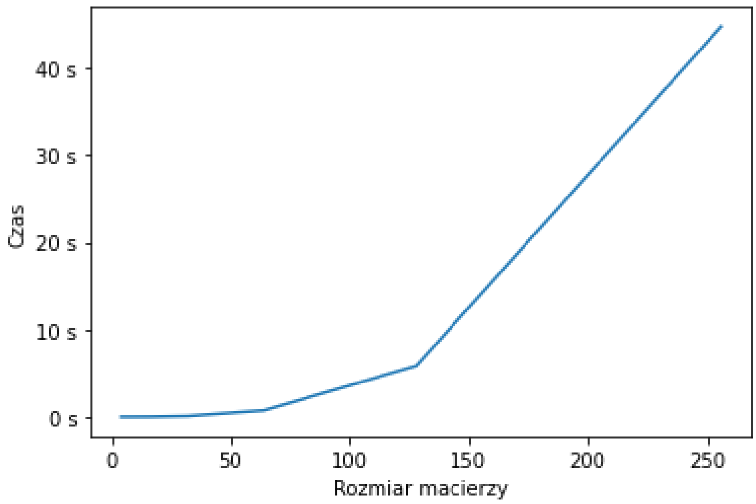
1.2 Kod algorytmu

```
def binet_core_algorithm(
    A: np.ndarray,
    B: np.ndarray,
    counter: Counter
) -> np.ndarray:
    add = counter.add
    sub = counter.sub
    mul = counter.mul
    if A.size > 1:
        split_at = A.shape[0]//2
        A11, A12, A21, A22 = split(A, split_at, split_at)
        B11, B12, B21, B22 = split(B, split_at, split_at)

        C11 = add(binet_core_algorithm(A11, B11, counter),
            binet_core_algorithm(A12, B21, counter))
        C12 = add(binet_core_algorithm(A11, B12, counter),
            binet_core_algorithm(A12, B22, counter))
        C21 = add(binet_core_algorithm(A21, B11, counter),
            binet_core_algorithm(A22, B21, counter))
        C22 = add(binet_core_algorithm(A21, B12, counter),
            binet_core_algorithm(A22, B22, counter))

        return np.concatenate(
            [
                np.concatenate([C11, C12], axis = 1),
                np.concatenate([C21, C22], axis = 1)
            ],
            axis = 0)
    else:
        return mul(A, B)
```

1.3 Benchmarki



Rysunek 1: Wykres czasu od rozmiaru macierzy



Rysunek 2: Wykres ilości operacji od rozmiaru macierzy

1.4 Analiza złożoności

W każdym mnożeniu macierzy o rozmiarze n wykonujemy 8 podwywołań algorytmu na macierzach o rozmiarach $n/2$. Taka rekurencja prowadzi do złożoności $O(n^{log_2(8)}) = O(n^3)$ (twierdzenie o rekurencji uniwersalnej).

1.5 Porównanie z matlabem

Matlab:

```
A = [1 3 5; 2 4 6; 3 5 7];
B = [1 2 3; 4 5 6; 7 8 9];

disp(A * B);

% output
```

```
>> matrix_mul
48        57        66
60        72        84
72        87       102
```

Python:

```
A = np.array([[1, 3, 5], [2, 4, 6], [3, 5, 7]])
B = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(binet_algorithm(A, B, Counter()))

# output
[[ 48  57  66]
 [ 60  72  84]
 [ 72  87 102]]
```

2 Algorytm Strassen'a

2.1 Opis algorytmu

W porównaniu z algorytmem Binét'a w kazdym podwywołaniu użyjemy tylko 7 mnożeń, co pozwala na osiągnięcie lepszej złożoności obliczeniowej. Ponownie używamy "divide and conquer". Stosujemy wzory: Stosujemy wzór:

$$\begin{aligned}
P_1 =& (A_{11} + A_{22})(B_{11} + B_{22}) \\
P_2 =& (A_{21} + A_{22})B_{11} \\
P_3 =& A_{11}(B_{12} - B_{22}) \\
P_4 =& A_{22}(B_{21} - B_{11}) \\
P_5 =& (A_{11} + A_{12})B_{22} \\
P_6 =& (A_{21} - A_{11})(B_{11} + B_{12}) \\
P_7 =& (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}$$

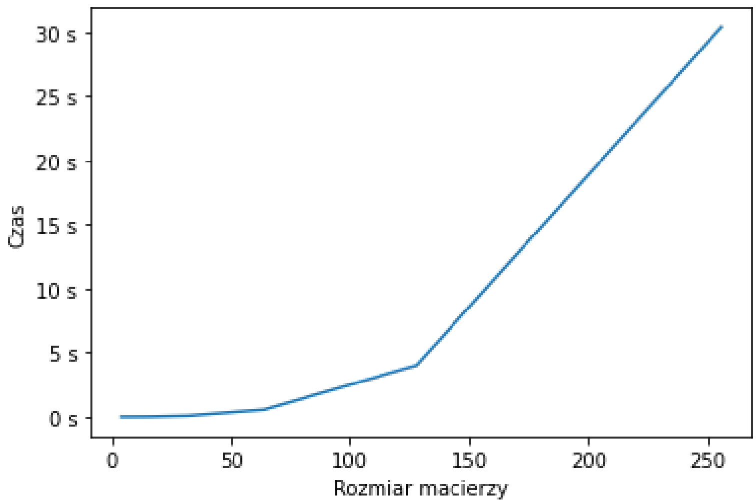
$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} (P_1 + P_4 - P_5 + P_7) & (P_3 + P_5) \\ (P_2 + P_4) & (P_1 - P_2 + P_3 + P_6) \end{bmatrix}$$

2.2 Kod algorytmu

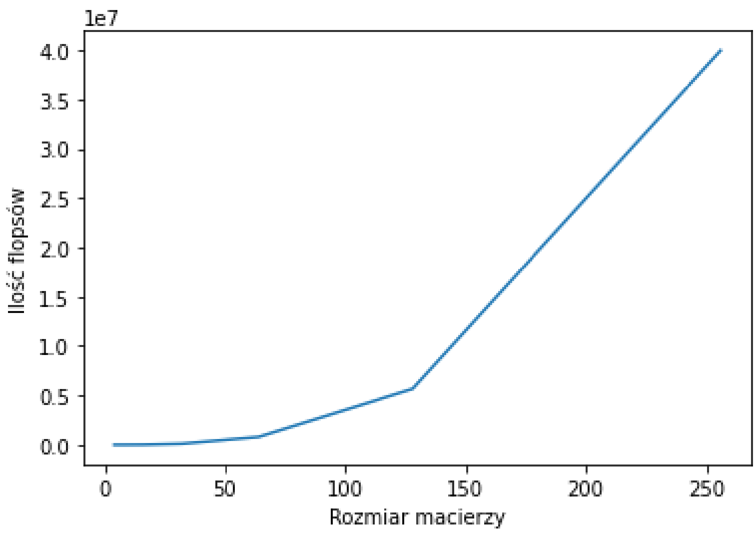
```
def strassen_core_algorith(
    A: np.ndarray,
    B: np.ndarray,
    counter: Counter
) -> np.ndarray:
    add = counter.add
    sub = counter.sub
    mul = counter.mul
    if A.size > 1:
        split_at = A.shape[0]//2
        A11, A12, A21, A22 = split(A, split_at, split_at)
        B11, B12, B21, B22 = split(B, split_at, split_at)

        M1 = strassen_core_algorith(add(A11, A22),
            add(B11, B22), counter)
        M2 = strassen_core_algorith(add(A21, A22),
            B11, counter)
        M3 = strassen_core_algorith(A11,
            sub(B12, B22), counter)
        M4 = strassen_core_algorith(A22,
            sub(B21, B11), counter)
        M5 = strassen_core_algorith(add(A11, A12),
            B22, counter)
        M6 = strassen_core_algorith(sub(A21, A11),
            add(B11, B12), counter)
        M7 = strassen_core_algorith(sub(A12, A22),
            add(B21, B22), counter)

        C11 = add(sub(add(M1, M4), M5), M7)
```

```
        C12 = add(M3, M5)
        C21 = add(M2, M4)
        C22 = add(add(sub(M1, M2), M3), M6)

        return np.concatenate(
            [
                np.concatenate([C11, C12], axis = 1),
                np.concatenate([C21, C22],axis = 1)
            ],
            axis = 0)
    else:
        return mul(A, B)
```

2.3 Benchmarki



Rysunek 3: Wykres czasu od rozmiaru macierzy



Rysunek 4: Wykres ilości operacji od rozmiaru macierzy

2.4 Analiza złożoności

Analogicznie jak w algorytmie Binét'a macierz dzielona jest i rozpatywana jako 4 podmacierze, jednak w tym przypadku nastepuj 7 podwywołan algorytmu, co ostatecznie daje złożonosć $O(n^{log2(7)}) \approx O(n^{2.87})$.

2.5 Porównanie z matlabem

Matlab:

```
A = [1 3 5; 2 4 6; 3 5 7];
B = [1 2 3; 4 5 6; 7 8 9];

disp(A * B);

% output
>> matrix_mul
48    57    66
60    72    84
72    87   102
```

Python:

```
A = np.array([[1, 3, 5], [2, 4, 6], [3, 5, 7]])
B = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(strassen_algorith(A, B, Counter()))

# output
[[ 48  57  66]
 [ 60  72  84]
 [ 72  87 102]]
```