# Algorytmy macierzowe

## Laboratorium 1

### Sprawozdanie

**Łukasz Stępień, Szymon Urbański**

# 1. Temat zadania

Laboratorium polegało na zaimplementowaniu i przetestowaniu rekurencyjnych algorytmów mnożenia macierzy metodą Binét'a  oraz Strassena. Należało również narysować wykres zależności czasu i ilości obliczeń od rozmiarów macierzy oraz określić złożoność obliczeniową tych algorytmów.

# 2. Rozwiązanie

### 2.1. Metoda Binéta

Korzystając z informacji przedstawionych na wykładzie zaimplementowano rekurencyjne mnożenie macierzy dla dowolnego, możliwego przypadku.

- **Pseudokod algorytmu:**

```
recursvie_matrix(A, B):
   if B jest puste Then Return pusta macierz (1)
   if A ma jeden wiersz Or B ma jedną kolumnę Then
      Return A * B (według definicji)

   Podziel macierz A na macierze A11, A12, A21, A22 (2)
   Podziel macierz B na macierze B11, B12, B21, B22

   C11 = dodaj recursvie_matrix(A11, B11) i recursvie_matrix(A12, B21) (3)
   C12 = dodaj recursvie_matrix(A11, B12) i recursvie_matrix(A12, B22)
   C21 = dodaj recursvie_matrix(A21, B11) i recursvie_matrix(A22, B21)
   C22 = dodaj recursvie_matrix(A21, B12) i recursvie_matrix(A22, B22)

   Połącz C11, C12, C21, C22 (4)
   Return C
```

- **Złożoność obliczeniowa:**

Niech T(n) oznacza czas procedury *recursvie_matrix* dla dwóch macierzy *n*x*n*.

(1) W przypadku bazowym (*n=1*), wykonujemy jedno skalarne mnożenie, więc:

$$T(n) = \Theta(1)$$

(2) Zakładamy, że podział wykonuje się w stałym czasie $\Theta(1)$.

(3) Wykonujemy łącznie 8 wywołań rekurencyjnych procedury *recursvie_matrix*. Ponieważ w każdym tym wywołaniu mnożymy macierze $\frac{n}{2}$ x $\frac{n}{2}$, co wnosi $T\left(\frac{n}{2}\right)$ do łącznego czasu działania, czas tych ośmiu wywołań to $8T\left(\frac{n}{2}\right)$. Następnie uwzględniamy 4 dodawania macierzy zawierających

$\frac{n^2}{4}$ elementów. Każde z czterech dodawań wymaga $\Theta(n^2)$. Ponieważ liczba dodawań macierzy jest stała, łączny czas ich wykonywania to $\Theta(n^2)$.

(4) Zakładamy, że łączenie wykonuje się w stałym czasie $\Theta(1)$.

Rekurencja opisująca czas działania algorytmu Bineta przedstawia się równaniem:

$$T(n) = \begin{cases} \Theta(1) & dla\ n = 1 \\ 8T\left(\frac{n}{2}\right) + \Theta(n^2) & dla\ n > 1 \end{cases}$$

Rozwiązaniem tej rekurencji jest T(n) = $\Theta(n^3)$

- **Najważniejsze fragmenty kodu:**

```
def add_m(A, B):
    global cnt_a

    if (A == [[]]):
        return B
    if (B == [[]]):
        return A

    C = [[0 for _ in range(len(A[0]))] for _ in range(len(A))]
    for i in range(len(A)):
        for j in range(len(A[0])):
            C[i][j] = A[i][j] + B[i][j]
            cnt_a += 1
    return C
```

Fragment kodu odpowiedzialny za dodanie do siebie dwóch macierzy.

```
if l == 1:
    C = [[0] for _ in range(n)]
    for i in range(n):
        for j in range(m):
            C[i][0] += A[i][j] * B[j][0]
            cnt_m += 1
            cnt_a += 1
    return C

if n == 1:
    C = [[0 for _ in range(l)]]
    for i in range(m):
        for j in range(l):
            C[0][j] += A[0][i] * B[i][j]
            cnt_m += 1
            cnt_a += 1
    return C
```

Fragment kodu odpowiedzialny za wykonanie mnożenia w przypadkach granicznych.

```
A11 = [row[:m // 2] for row in A[:n // 2]]
A12 = [row[m // 2:] for row in A[:n // 2]]
A21 = [row[m // 2:] for row in A[n // 2:]]
A22 = [row[:m // 2] for row in A[n // 2:]]

A = [[A11, A12], [A22, A21]]

B11 = [row[:l // 2] for row in B[:m // 2]]
B12 = [row[l // 2:] for row in B[:m // 2]]
B21 = [row[l // 2:] for row in B[m // 2:]]
B22 = [row[:l // 2] for row in B[m // 2:]]

B = [[B11, B12], [B22, B21]]
```

Fragment kodu odpowiedzialny za podzielenie macierzy na 4 części.

```
for i in range(len(A)):
    for j in range(len(B[0])):
        for k in range(len(B)):
            T[k] = recursive_multiplication(A[i][k], B[k][j])
        C[i][j] = add_m(T[0], T[1])

res = []
for i in range(2):
    for j in range(len(C[i][0])):
        res.append(C[i][0][j] + C[i][1][j])
return res
```

Fragment kodu odpowiedzialny za wyznaczenie wyniku mnożenia i połączenie składowych wynikowej macierzy.

### 2.2. Metoda Strassena

Zaimplementowano także rekurencyjne mnożenie macierzy o wymiarach NxN, gdzie N jest potęgą liczby 2 z wykorzystaniem metody Strassena.

- **Pseudokod algorytmu:**

  recursvie_matrix_strassen(A, B):
  if B jest puste Then Return pusta macierz (1)
  if A ma jeden wiersz Or B ma jedną kolumnę Then
  Return A * B (według definicji)

  Podziel macierz A na macierze A11, A12, A21, A22 (2)
  Podziel macierz B na macierze B11, B12, B21, B22

  Oblicz rekurencyjnie P1, P2, ..., P7 (3)

  Oblicz C11, C12, C21, C22 (4)

  Połącz C11, C12, C21, C22 (5)

  Return C

- **Złożoność obliczeniowa:**

  Niech T(n) oznacza czas procedury *recursvie_matrix_strassen* dla dwóch macierzy *nxn*.

  (1) W przypadku bazowym (*n=1*), wykonujemy jedno skalarne mnożenie, więc:

  $$T(n) = \Theta(1)$$

(2) Zakładamy, że podział wykonuje się w stałym czasie $\Theta(1)$.

(3) Wykonujemy łącznie 7 wywołań rekurencyjnych procedury *recursvie_matrix*. Ponieważ w każdym tym wywołaniu mnożymy macierze $\frac{n}{2}$ x $\frac{n}{2}$, co wnosi $T\left(\frac{n}{2}\right)$ do łącznego czasu działania, czas tych siedmiu wywołań to $7T\left(\frac{n}{2}\right)$.

(4) Uwzględniamy dodawanie i odejmowanie macierzy zawierających $\frac{n^2}{4}$ elementów. Każde z tych działań wymaga $\Theta(n^2)$, a ich liczba jest stała. Oznacza to, że łączny czas ich wykonywania to $\Theta(n^2)$.

(5) Zakładamy, że łączenie wykonuje się w stałym czasie $\Theta(1)$.

Rekurencja opisująca czas działania algorytmu Strassena przedstawia się równaniem:

$$T(n) = \begin{cases} \Theta(1) & dla\ n = 1 \\ 7T\left(\frac{n}{2}\right) + \Theta(n^2) & dla\ n > 1 \end{cases}$$

Rozwiązaniem tej rekurencji jest $T(n) = \Theta(n^{lg7}) \approx \Theta(n^{2.8074})$ .

- **Najważniejsze fragmenty kodu:**

```
def sub_m(A, B):
    global cnt_a

    if (A == [[]]):
        C = [[0 for _ in range(len(B[0]))] for _ in range(len(B))]
        for i in range(len(B)):
            for j in range(len(B[0])):
                C[i][j] = -B[i][j]
                cnt_a += 1
        return C
    if (B == [[]]):
        return A

    C = [[0 for _ in range(len(A[0]))] for _ in range(len(A))]
    for i in range(len(A)):
        for j in range(len(A[0])):
            C[i][j] = A[i][j] - B[i][j]
            cnt_a += 1
    return C
```

Fragment kodu odpowiedzialny za odejmowanie dwóch macierzy.

```
P1 = recursive_multiplication(add_m(A1, A4), add_m(B1, B4))
P2 = recursive_multiplication(add_m(A3, A4), B1)
P3 = recursive_multiplication(A1, sub_m(B2, B4))
P4 = recursive_multiplication(A4, sub_m(B3, B1))
P5 = recursive_multiplication(add_m(A1, A2), B4)
P6 = recursive_multiplication(sub_m(A3, A1), add_m(B1, B2))
P7 = recursive_multiplication(sub_m(A2, A4), add_m(B3, B4))
```

Fragment kodu odpowiedzialny za kolejne $P_i$.

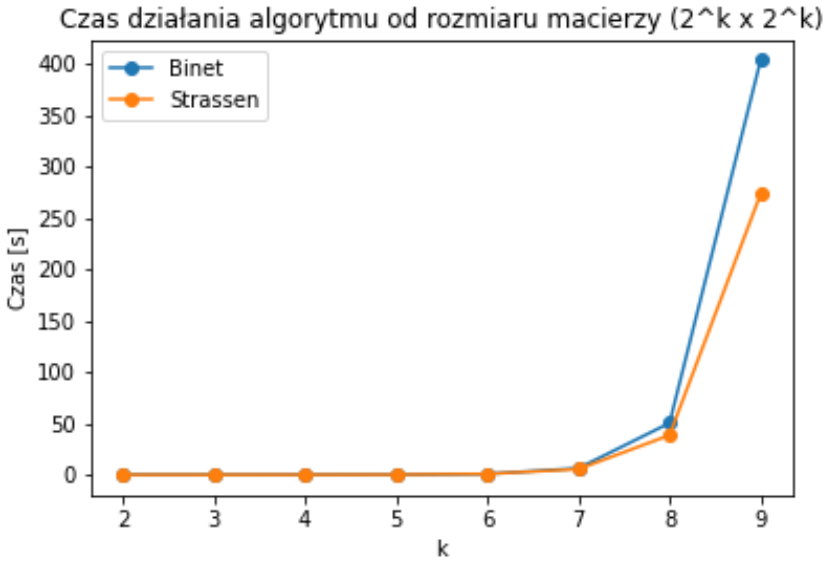
```
C = [[0 for _ in range(2)] for _ in range(2)]
C[0][0] = add_m(sub_m(add_m(P1, P4), P5),P7)
C[0][1] = add_m(P3, P5)
C[1][0] = add_m(P2, P4)
C[1][1] = add_m(add_m(sub_m(P1, P2), P3), P6)
```

Fragment kodu odpowiedzialny za wyznaczenie wyniku mnożenia.

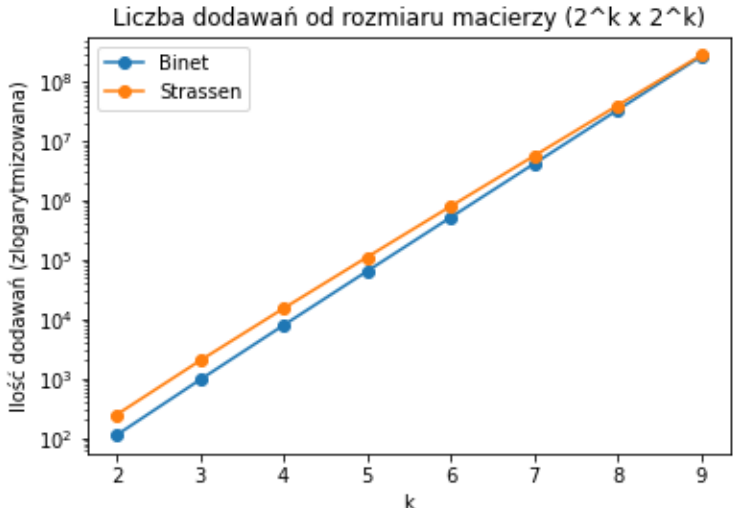
W implementacji tego algorytmu fragmenty kodu odpowiedzialne za dodawanie macierzy, warunki brzegowe rekurencji, podział macierzy na cztery części oraz łączenie wyniku zaimplementowano tak samo jak w algorytmie Binét'a.

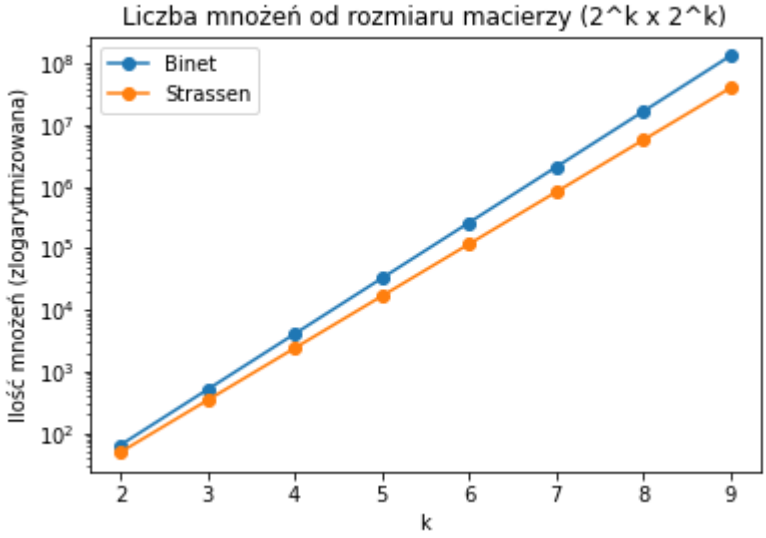
# 3. Wyniki

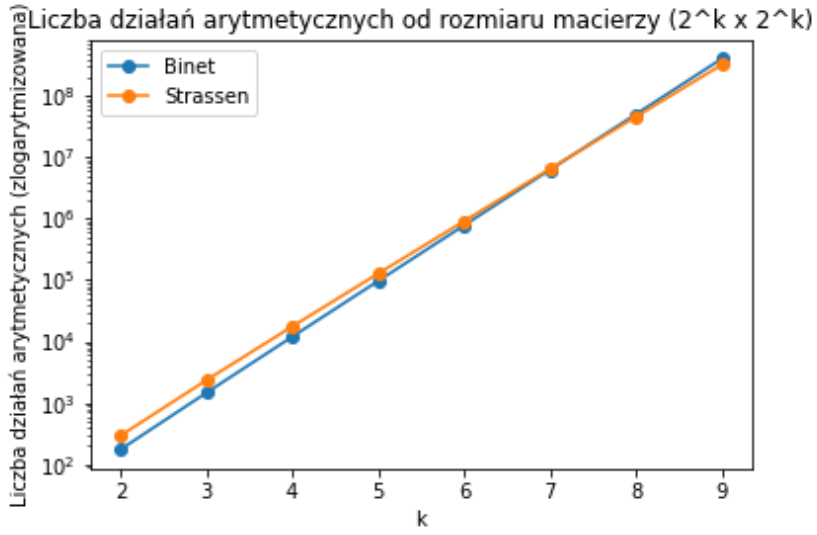
Na poniższym wykresie przedstawiono zależność czasu wyznaczania wyniku mnożenia macierzy w zależności od ilości elementów tej macierzy.



Analogicznie wygenerowano wykres przedstawiający liczbę wykonanych operacji arytmetycznych w zależności od wielkości macierzy.



Dodatkowo dla dwóch macierzy 2x2:

$$A = \begin{bmatrix} 0.9125387713535192 & 0.45230209561129187 \\ 0.07747586362850696 & 0.1503258457223823 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.6765535423510954 & 0.943849754188833 \\ 0.4181924832761693 & 0.46715985459335185 \end{bmatrix}$$

zaimplementowane algorytmy zwróciły następujące wyniki:

- Binet:

$$C_{Binét} = \begin{bmatrix} 0.8065306748466412 & 1.0725968762478382 \\ 0.11528170870781015 & 0.14335177507062458 \end{bmatrix}$$

- Strassen:

$$C_{Strassen} = \begin{bmatrix} 0.806530674846641 & 1.0725968762478382 \\ 0.11528170870781015 & 0.14335177507062458 \end{bmatrix}$$

Iloczyn ten przeliczono również w programie *MATLAB,* który zwrócił wynik:

$$C_{MATLAB} = \begin{bmatrix} 0.806530674846641 & 1.072596876247838 \\ 0.115281708707810 & 0.143351775070625 \end{bmatrix}$$

# 4. Wnioski

Patrząc na wykresy można stwierdzić, że szybszy czas wykonania mnożenia macierzy charakteryzuje metoda Strassena. Liczba operacji arytmetycznych w miarę zwiększania rozmiaru macierzy ma większą tendencję spadkową niż w przypadku algorytmu Binéta.

- liczba dodawań: dla małych k mniej operacji wykonuje algorytm Binéta, lecz przy k>=9 tę cechę wykazuje algorytm Strassena
- liczba mnożeń: dla wszystkich k algorytm Strassena wykonuje mniej operacji niż algorytm Binéta
- liczba wszystkich działań arytmetycznych: dla k<8 mniej działań wykonuje algorytm Binéta, lecz dla k>=8 algorytm Strassena wychodzi na prowadzenie

Złożoność obliczeniowa w zależności od rozmiaru macierzy (*nxn*) dla obu algorytmów przedstawia się następująco:

- Binét: $\Theta(n^3)$,
- Strassen: $\Theta(n^{2.8074})$ .

Przy wyprowadzaniu tych złożoności w celu ułatwienia rachunków założono, że dzielenie oraz łączenie macierzy odbywa się w czasie stałym. Jednak zaimplementowane tutaj algorytmy nie posiadają tej cechy, gdyż w każdym wywołaniu rekurencyjnym tworzymy nowe macierze. Fakt ten może wpływać na nieefektywne wykonywanie obu algorytmów, ponieważ już przy macierzach o rozmiarach $2^9 x 2^9$ czas ten wynosi powyżej czterech minut. Dodatkowo

czas ten może wydłużać implementacja algorytmów na listach w języku Python, które cechują się wolnym czasem operacji.

Porównując iloczyny dwóch macierzy o wymiarach 2x2 wyliczonymi przez oba algorytmy oraz program MATLAB można stwierdzić, że z dokładnością do błędów numerycznych wszystkie trzy iloczyny są identyczne. Można z tego wywnioskować, że zaimplementowane algorytmy działają poprawnie.

# 5. Bibliografia


- wykład z przedmiotu „Algorytmy macierzowe” przygotowany przez prof. dr hab. Macieja Paszyńskiego
- https://en.wikipedia.org/wiki/Strassen_algorithm
- Thomas H. Cormen - „Wprowadzenie do algorytmów”