Mnożenie macierzy algorytmem klasycznym oraz rekurencyjnym Strassena

Paweł Zaręba, Marcin Mikuła

Operacje na macierzach

1. Dzielenie macierzy wejściowej na mniejsze macierze

```
public static void divideMatrix(int[][] parent, int[][] child, int
rowShift, int colShift) {
    for (int i = 0; i < child.length; i++) {
        for (int j = 0; j < child.length; j++) {
            child[i][j] = parent[i + rowShift][j + colShift];
        }
    }
}
```

Wartości z wejściowej macierzy **parent** wpisywane są do odpowiedniej macierzy **child**. Zależy to od kolejnych dwóch argumentów metody: **rowShift** oraz **colShift**. Są to indeksy wiersza i kolumny wyznaczające fragment macierzy parent do wpisania w macierz child.

2. Dodawanie

```
public static int[][] addMatrix(int[][] A, int[][] B) {
    int n = A.length;
    int[][] C = new int[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
    return C;
}
```

Implementacja klasycznego dodawania macierzy, na wejściu metoda przyjmuje macierz **A** oraz macierz **B** i jako wynik dodawania zwraca nową macierz **C**.

3. Odejmowanie

```
public static int[][] subtractMatrix(int[][] A, int[][] B) {
    int n = A.length;
    int[][] C = new int[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = A[i][j] - B[i][j];
        }
    }
    return C;
}
```

Implementacja odejmowania, analogiczna do dodawania. Na wejściu macierze **A** i **B**, na wyjściu macierz wynikowa **C.**

4. Łączenie

```
public static void combineMatrix(int[][] parent, int[][] child,
int rowShift, int colShift) {
    for (int i = 0; i < child.length; i++) {
        for (int j = 0; j < child.length; j++) {
            parent[i + rowShift][j + colShift] = child[i][j];
        }
    }
}
```

Metoda służąca do przepisywania obliczonego wyniku w macierzach **child** do odpowiedniego miejsca (zależnego od parametrów **rowShift** i **colShift)** macierzy wynikowej **parent**.

5. Mnożenie metodą Strassena

```
public static int[][] multiply(int[][] A, int[][] B) {
    int n = A.length;

    // Warunek stopu rekurencji
    if (n == 1) {
        int[][] C = new int[1][1];
        C[0][0] = A[0][0] * B[0][0];
        return C;
    } else {
```

Warunek stopu rekurencji, gdy algorytm dojdzie do macierzy o rozmiarze 1 x 1, przestaje rozbijać kolejne macierze na mniejsze.

```
        // Dzielenie macierzy na cztery podmacierze
        int[][] A11 = new int[n/2][n/2];
        int[][] A12 = new int[n/2][n/2];
        int[][] A21 = new int[n/2][n/2];
        int[][] A22 = new int[n/2][n/2];

        int[][] B11 = new int[n/2][n/2];
        int[][] B12 = new int[n/2][n/2];
        int[][] B21 = new int[n/2][n/2];
        int[][] B22 = new int[n/2][n/2];
```

Stworzenie czterech podmacierzy dla macierzy wejściowych **A** i **B**

```
        //Przepisywanie wartości z wejściowej do mniejszych macierzy
        divideMatrix(A, A11, 0, 0);
        divideMatrix(A, A12, 0, n/2);
        divideMatrix(A, A21, n/2, 0);
        divideMatrix(A, A22, n/2, n/2);
```

```
        divideMatrix(B, B11, 0, 0);
        divideMatrix(B, B12, 0, n/2);
        divideMatrix(B, B21, n/2, 0);
        divideMatrix(B, B22, n/2, n/2);
```

Podzielenie (przepisanie wartości) wejściowych macierzy **A** i **B** do nowo utworzonych podmacierzy.

```
        //Nowe wartości które zmniejszają liczbę mnożeń z 8 do 7
int[][] M1 = multiply(addMatrix(A11,A22),addMatrix(B11,B22));
int[][] M2 = multiply(addMatrix(A21, A22), B11);
int[][] M3 = multiply(A11, subtractMatrix(B12, B22));
int[][] M4 = multiply(A22, subtractMatrix(B21, B11));
int[][] M5 = multiply(addMatrix(A11, A12), B22);
int[][] M6 = multiply(subtractMatrix(A21, A11),addMatrix(B11,
B12));
int[][] M7 = multiply(subtractMatrix(A12, A22),
addMatrix(B21,B22));
```

Wyliczenie elementów **M** algorytmu Strassena, redukujących liczbę mnożeń z 8 do 7 względem klasycznego algorytmu mnożenia macierzy.

```
        // Obliczanie podmacierzy wynikowej
int[][] C11 = addMatrix(subtractMatrix(addMatrix(M1, M4), M5), M7);
int[][] C12 = addMatrix(M3, M5);
int[][] C21 = addMatrix(M2, M4);
int[][] C22 = addMatrix(addMatrix(subtractMatrix(M1, M2), M3), M6);
```

Obliczenie podmacierzy wynikowych na podstawie macierzy **M.**

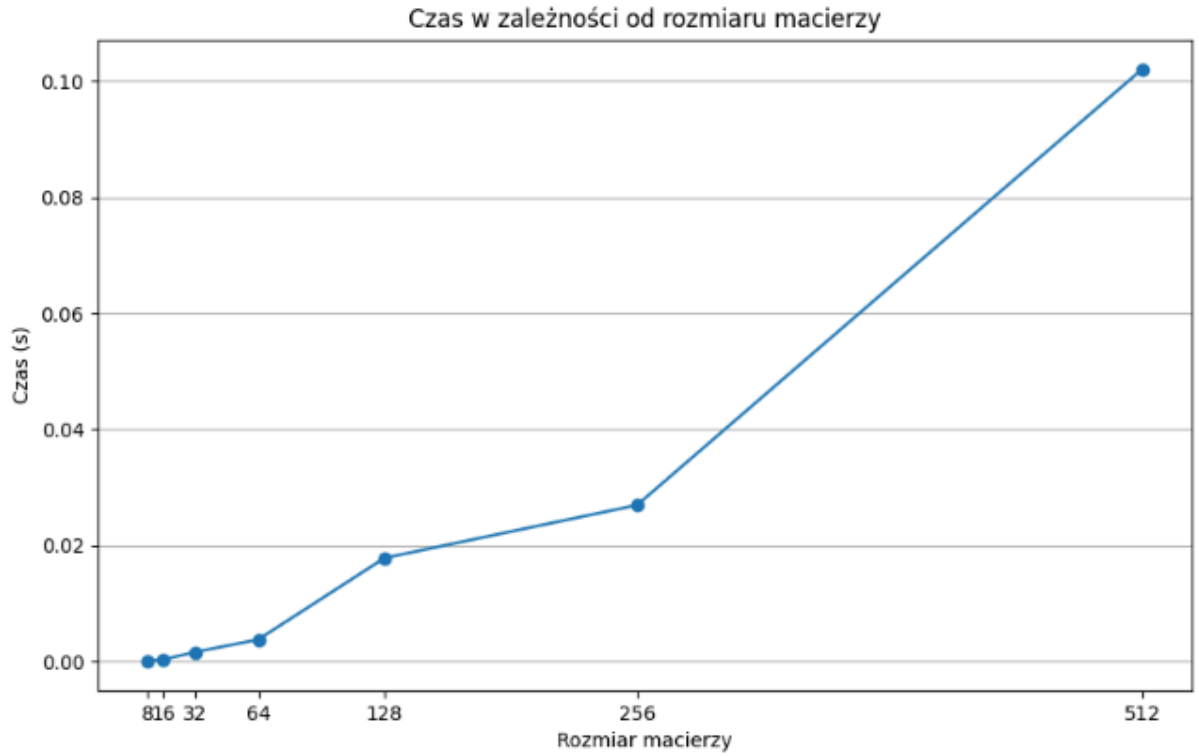
```
        // Łączenie podmacierzy wynikowej w jedną macierz
        int[][] C = new int[n][n];
        combineMatrix(C, C11, 0, 0);
        combineMatrix(C, C12, 0, n/2);
        combineMatrix(C, C21, n/2, 0);
        combineMatrix(C, C22, n/2, n/2);

        return C;
    }
}
```

Połączenie podmacierzy wynikowych w jedną macierz z wynikiem całego mnożenia.

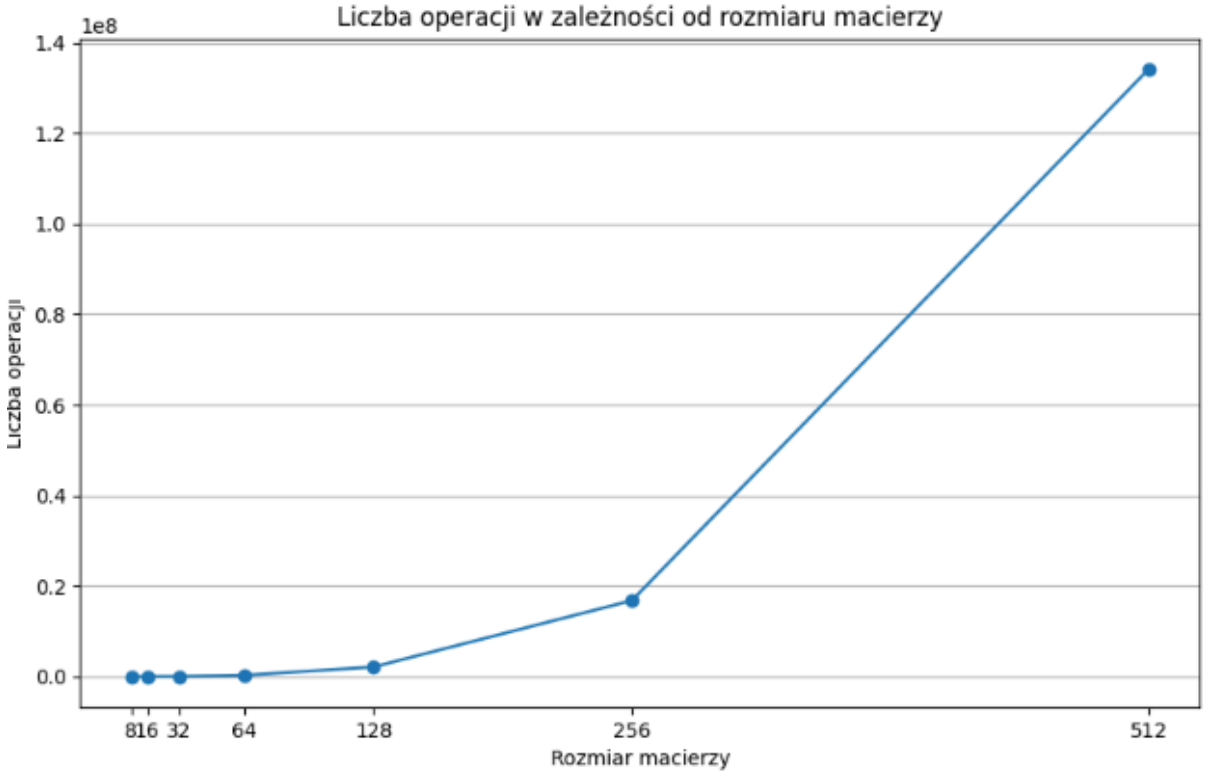
6. Mnożenie klasyczne

```java
public static int[][] multiply(int[][] A, int[][] B) {
    int n = A.length;
    int sum = 0;
    int[][] C = new int[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                sum = sum + A[i][k] * B[k][j];
            }
            C[i][j] = sum;
            sum = 0;
        }
    }
    return C;
}
```

Implementacja klasycznego mnożenia macierzy, wynik mnożenia wiersza i kolumny agregowany jest w zmiennej **sum**, która następnie jest wpisywana w odpowiednią komórkę macierzy wynikowej **C**.

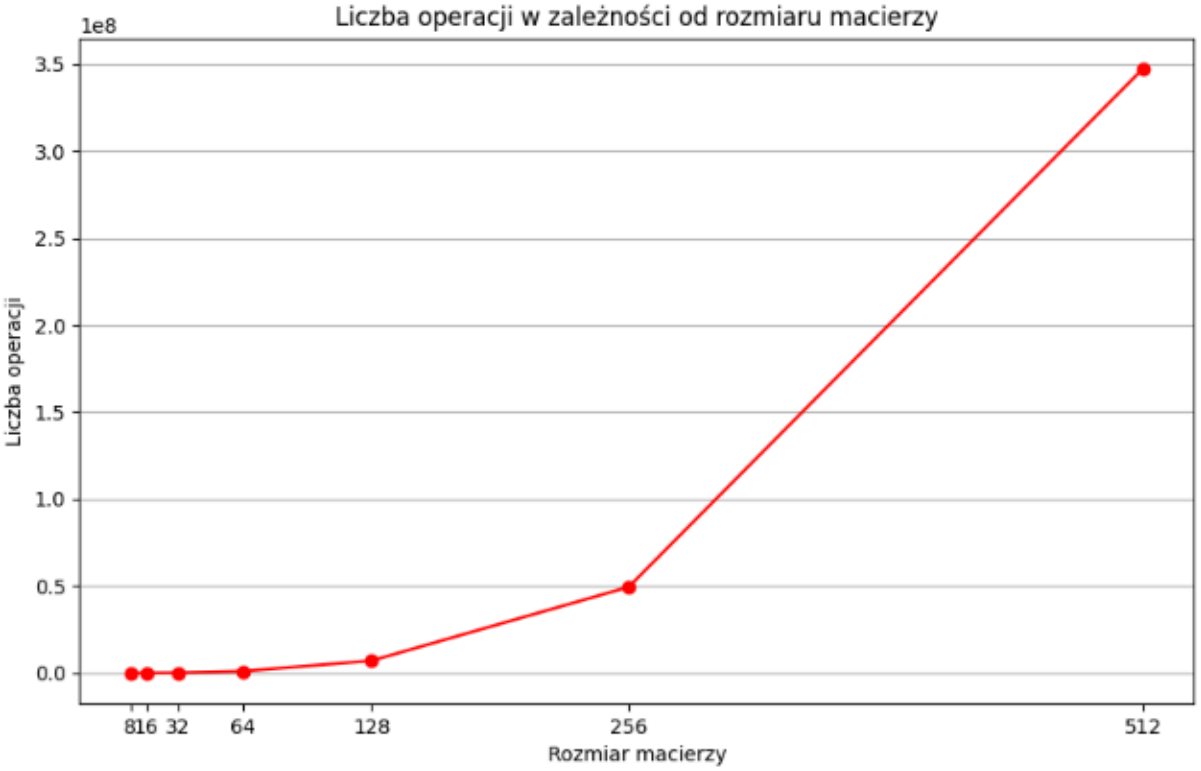


Rysunek 1. Czas w zależności od rozmiaru macierzy dla metody klasycznej.

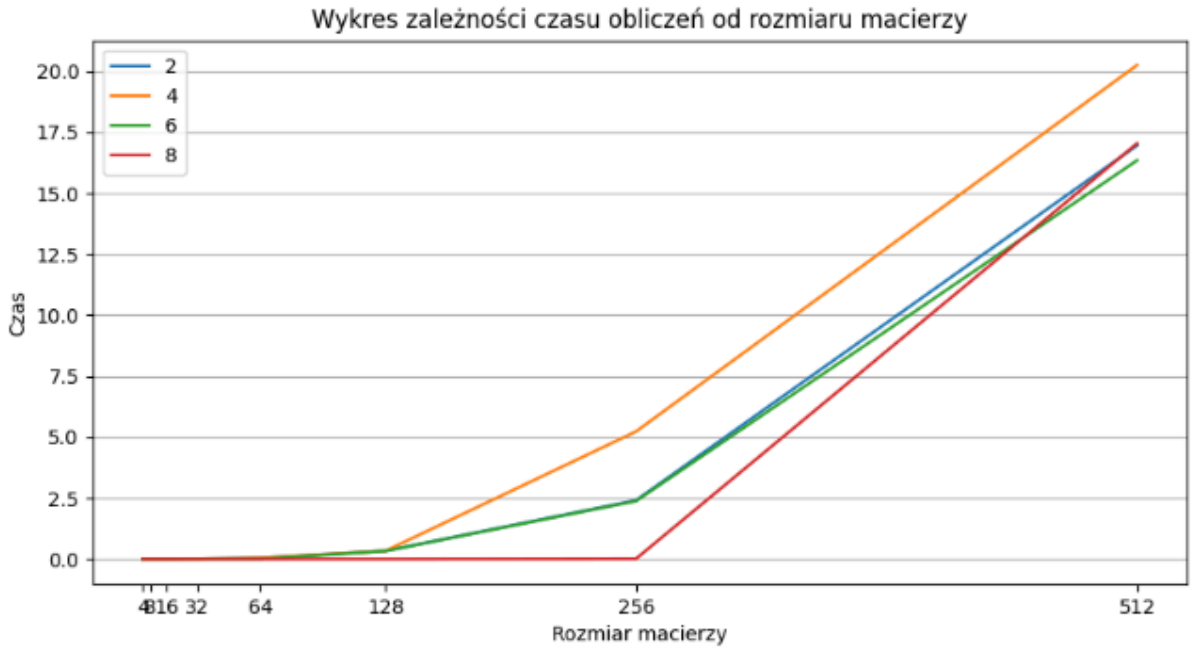


Rysunek 2. Liczba operacji w zależności od rozmiaru macierzy dla metody klasycznej.

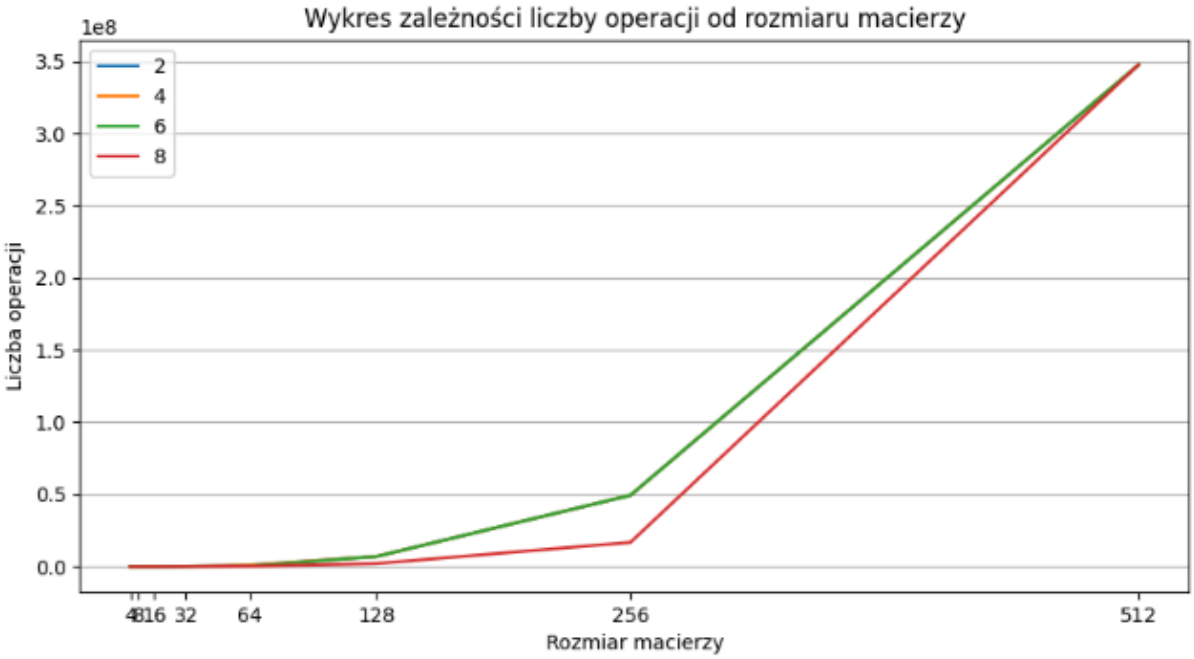
Rysunek 3. Czas w zależności od rozmiaru macierzy dla metody Strassena.



Rysunek 4. Liczba operacji w zależności od rozmiaru macierzy dla metody Strassena.



Rysunek 5. Czas w zależności od rozmiaru macierzy dla obu metod i różnych l.



Rysunek 6. Liczba operacji w zależności od rozmiaru macierzy dla obu metod i różnych l.

Jak widać na wykresach, metoda Strassena działa zauważalnie wolniej od metody klasycznej. Jest to spodziewane zachowanie, gdyż metoda Strassena jest rekurencyjna, co niesie za sobą dodatkowe koszta czasowe - zapamiętanie ramek wywołań rekurencyjnych oraz alokacja nowych zasobów pamięci dla każdego wywołania.
Ze względu na implementację rozwiązania w języku programowania Java, wymienione wyżej operacje są kosztowne czasowo - co powoduje różnicę w wydajności.