Practical 5: Matrix Multiplication using DnC

Aim

Implement both a **standard $O(n^3)$ matrix multiplication** algorithm and **Strassen's matrix multiplication** algorithm. Using empirical testing, try and estimate the constant factors for the runtime equations of the two algorithms. How big must n be before Strassen's algorithm becomes more efficient than the standard algorithm?

Algorithm

1. Standard Matrix Multiplication

For each row i of matrix A, and for each column j of matrix B:

a. Initialize a variable sum to zero.

b. For each index k from 1 to n:

- **i.** Multiply the element in the i-th row and k-th column of matrix A by the element in the k-th row and j-th column of matrix B.
- **ii.** Add the result to the variable sum.

c. Assign the value of the variable sum to the element in the i-th row and j-th column of matrix C.

d. Return the result matrix C.

The above algorithm follows the traditional method of matrix multiplication,

2. Strassen's Matrix Multiplication

- If the size of the matrices is 1, perform regular multiplication of the matrices to obtain the result matrix.
- If the size of the matrices is greater than 1, split the matrices into four smaller matrices of size n/2 x n/2.
- Recursively compute the products of the smaller matrices using the same algorithm.
- Combine the results of the smaller matrices to obtain the final result matrix.
- Create a result matrix C of size m x p, initialized with zeros.

Program

a. Standard Matrix Multiplication

```
package PRACTICAL5;

import java.util.*;

public class MatrixMultiply {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int i,j,k;

      MatrixMultiply obj = new MatrixMultiply();

      System.out.print("Enter no. of rows of 1st array: ");
      int row1 = sc.nextInt();
      System.out.print("Enter no. of columns of 1st array: ");
      int col1 = sc.nextInt();
```

```
        System.out.print("Enter no. of rows of 2nd array: ");
        int row2 = sc.nextInt();
        System.out.print("Enter no. of columns of 2nd array: ");
        int col2 = sc.nextInt();

        if (col1 == row2){
            // First Array
            System.out.println("\nEnter elements of 1st array");
            int a[][] = new int[row1][col1];
            for (i=0; i<row1; i++){
                for (j=0; j<col1; j++){
                    a[i][j] = (int)(Math.random()*(20)+1);
                }
            }
            System.out.println("1st Array: "+Arrays.deepToString(a));

            // Second Array
            System.out.println("\nEnter elements of 2nd array");
            int b[][] = new int[row2][col2];
            for (i=0; i<row2; i++){
                for (j=0; j<col2; j++){
                    b[i][j] = (int)(Math.random()*(50-21+1)+1);
                }
            }
            System.out.println("2nd Array: "+Arrays.deepToString(a));

            // Multiplying Array
            int c[][] = new int[row1][col2];
            for (i = 0; i < row1; i++) {
                for (j = 0; j < col2; j++) {
                    for (k = 0; k < row2; k++){
                        c[i][j] += a[i][k] * b[k][j];
                    }
                }
            }
            System.out.print("\nThe Multiplied array is ");
            System.out.println(Arrays.deepToString(c));
        }

        else{
            System.out.println("\nArrays can't be multiplied!!");
        }
        sc.close();
    }
}
```

Output:

```
Enter no. of rows of 1st array: 3
Enter no. of columns of 1st array: 2
Enter no. of rows of 2nd array: 2
Enter no. of columns of 2nd array: 3

Enter elements of 1st array
1st Array: [[5, 10], [18, 14], [7, 19]]

Enter elements of 2nd array
2nd Array: [[5, 10], [18, 14], [7, 19]]

The Multiplied array is [[190, 305, 250], [310, 548, 614], [351, 552, 415]]
```

b. Strassen's matrix multiplication

```
package PRACTICAL5;

import java.util.*;

class MMStrassens {

    static int ROW_1 = 3,COL_1 = 2, ROW_2 = 2, COL_2 = 4;

    public static void printMat(int[][] a, int r, int c){
        for(int i=0;i<r;i++){
            for(int j=0;j<c;j++){
                System.out.print(a[i][j]+" ");
            }
            System.out.println();
        }
        System.out.println();
    }

    public static void print(String display, int[][] matrix,int start_row, int start_column, int end_row,int
end_column)
    {
        System.out.println(display + " =>\n");
        for (int i = start_row; i <= end_row; i++) {
            for (int j = start_column; j <= end_column; j++) {
                //cout << setw(10);
                System.out.print(matrix[i][j]+" ");
            }
            System.out.println();
        }
        System.out.println();
    }
```

```
    public static void add_matrix(int[][] matrix_A,int[][] matrix_B,int[][] matrix_C, int split_index)
    {
        for (int i = 0; i < split_index; i++){
            for (int j = 0; j < split_index; j++){
                matrix_C[i][j] = matrix_A[i][j] + matrix_B[i][j];
            }
        }
    }

    public static void initWithZeros(int[][] a, int r, int c){
        for(int i=0;i<r;i++){
            for(int j=0;j<c;j++){
                a[i][j]=0;
            }
        }
    }

    public static int[][] multiply_matrix(int[][] matrix_A,int[][] matrix_B)
    {
        int col_1 = matrix_A[0].length;
        int row_1 = matrix_A.length;
        int col_2 = matrix_B[0].length;
        int row_2 = matrix_B.length;

        if (col_1 != row_2) {
            System.out.println("\nError: The number of columns in Matrix A  must be equal to the number
of rows in Matrix B\n");
            int[][] temp = new int[1][1];
            temp[0][0]=0;
            return temp;
        }

        int[] result_matrix_row = new int[col_2];
        Arrays.fill(result_matrix_row,0);
        int[][] result_matrix = new int[row_1][col_2];
        initWithZeros(result_matrix,row_1,col_2);

        if (col_1 == 1){
            result_matrix[0][0] = matrix_A[0][0] * matrix_B[0][0];
        } else {
            int split_index = col_1 / 2;

            int[] row_vector = new int[split_index];
            Arrays.fill(row_vector,0);

            int[][] result_matrix_00 = new int[split_index][split_index];
            int[][] result_matrix_01 = new int[split_index][split_index];
            int[][] result_matrix_10 = new int[split_index][split_index];
            int[][] result_matrix_11 = new int[split_index][split_index];
            initWithZeros(result_matrix_00,split_index,split_index);
            initWithZeros(result_matrix_01,split_index,split_index);
            initWithZeros(result_matrix_10,split_index,split_index);
```

```
        initWithZeros(result_matrix_11,split_index,split_index);

        int[][] a00 = new int[split_index][split_index];
        int[][] a01 = new int[split_index][split_index];
        int[][] a10 = new int[split_index][split_index];
        int[][] a11 = new int[split_index][split_index];
        int[][] b00 = new int[split_index][split_index];
        int[][] b01 = new int[split_index][split_index];
        int[][] b10 = new int[split_index][split_index];
        int[][] b11 = new int[split_index][split_index];

        initWithZeros(a00,split_index,split_index);
        initWithZeros(a01,split_index,split_index);
        initWithZeros(a10,split_index,split_index);
        initWithZeros(a11,split_index,split_index);
        initWithZeros(b00,split_index,split_index);
        initWithZeros(b01,split_index,split_index);
        initWithZeros(b10,split_index,split_index);
        initWithZeros(b11,split_index,split_index);


        for (int i = 0; i < split_index; i++){
            for (int j = 0; j < split_index; j++) {
                a00[i][j] = matrix_A[i][j];
                a01[i][j] = matrix_A[i][j + split_index];
                a10[i][j] = matrix_A[split_index + i][j];
                a11[i][j] = matrix_A[i + split_index][j + split_index];
                b00[i][j] = matrix_B[i][j];
                b01[i][j] = matrix_B[i][j + split_index];
                b10[i][j] = matrix_B[split_index + i][j];
                b11[i][j] = matrix_B[i + split_index][j + split_index];
            }
        }

        add_matrix(multiply_matrix(a00, b00),multiply_matrix(a01, b10),result_matrix_00,
split_index);
        add_matrix(multiply_matrix(a00, b01),multiply_matrix(a01, b11),result_matrix_01,
split_index);
        add_matrix(multiply_matrix(a10, b00),multiply_matrix(a11, b10),result_matrix_10,
split_index);
        add_matrix(multiply_matrix(a10, b01),multiply_matrix(a11, b11),result_matrix_11,
split_index);

        for (int i = 0; i < split_index; i++){
            for (int j = 0; j < split_index; j++) {
                result_matrix[i][j] = result_matrix_00[i][j];
                result_matrix[i][j + split_index] = result_matrix_01[i][j];
                result_matrix[split_index + i][j] = result_matrix_10[i][j];
                result_matrix[i + split_index] [j + split_index] = result_matrix_11[i][j];
            }
        }
    }
```

```
        return result_matrix;
    }

    public static void main (String[] args) {
        int[][] matrix_A = { { 1, 1, 1, 1 },
                { 2, 2, 2, 2 },
                { 3, 3, 3, 3 },
                { 2, 2, 2, 2 } };

        System.out.println("Array A =>");
        printMat(matrix_A,4,4);

        int[][] matrix_B = { { 1, 1, 1, 1 },
                { 2, 2, 2, 2 },
                { 3, 3, 3, 3 },
                { 2, 2, 2, 2 } };

        System.out.println("Array B =>");
        printMat(matrix_B,4,4);

        int[][] result_matrix =  multiply_matrix(matrix_A, matrix_B);

        System.out.println("Result Array =>");
        printMat(result_matrix,4,4);
    }
}
```

Output:

```
Array A =>
1 1 1 1
2 2 2 2
3 3 3 3
2 2 2 2

Array B =>
1 1 1 1
2 2 2 2
3 3 3 3
2 2 2 2

Result Array =>
8 8 8 8
16 16 16 16
24 24 24 24
16 16 16 16
```

Analysis of Algorithm

The program implements the standard matrix multiplication algorithm in Java.

The time complexity of the **standard algorithm** is $\mathbf{O(n^3)}$, where n is the size of the input matrices. The constant factor for this algorithm will depend on the specific implementation and hardware used.

For **Strassen's algorithm**, the time complexity is $\mathbf{O(n^{log2(7)})}$, which is approximately $\mathbf{O(n^{2.81})}$. The constant factor for this algorithm is typically larger than the standard algorithm due to the additional overhead of recursive calls and matrix additions/subtractions.

The crossover point at which Strassen's algorithm becomes more efficient than the standard algorithm depends on the constant factors for each algorithm and the size of the input matrices. Generally, Strassen's algorithm becomes more efficient for large matrices (n > 100-200), but the specific crossover point will depend on the implementation and hardware used.

In practice, a hybrid algorithm that uses Strassen's algorithm for large matrices and the standard algorithm for small matrices is often used to achieve better performance across a range of input sizes.