

Name: _____

(Use this page as the cover sheet when you turn in your project report)

Project #1

CS3310 Design and Analysis of Algorithms

| | | |
|-------------------------|---|--|
| 1. (40 points) | Date sets, test strategies and results | |
| 2. (15 points) | Theoretical Complexity Comparisons | |
| 3. (15 points) | Classical Matrix Multiplication Vs Divide-and-Conquer Matrix Multiplication | |
| 4. (15 points) | Strength and Constraints | |
| 5. (15 points) | Program Correctness | |
| 6. (-10 points per day) | Late Penalty | |
| (100 points) | Total | |

- 1: 40 points
- 2: 15 points
- 3: 15 points
- 4: 15 points
- 5: 15 points

Total: 100 points

100

Data Sets:

The data values inside the matrix make up anywhere from 0 to 100 which are randomly generated. In the main class, I call generateMatrix() method and pass in a size as an argument. The method generates a matrix of size passed in argument and fills the matrix with randomly generated values. The actual collected data is displayed in the Results category.

Test Strategies:

First, we are only dealing with $n \times n$ matrices where n is a power of 2. Now, I am computing the three different methods of same size eight times and am getting the average of the computation time for the three approaches. This process is repeated by increasing the matrix size by power of two so long until the computer crashes. The data values inside the matrix make up anywhere from 0 to 100 which are randomly generated. The result would a print out the average calculated run times for classical, divide and conquer, and Strassen.

Results:

| Size (N x N) | Classical (ns) | Divide & Conquer (ns) | Strassen (ns) |
|--------------|------------------|-----------------------|-------------------|
| 2 | 1, 920 | 13, 155 | 1, 493 |
| 4 | 8, 204 | 181, 128 | 48, 684 |
| 8 | 45, 469 | 804, 931 | 433, 319 |
| 16 | 325, 611 | 7, 004, 488 | 1, 520, 684 |
| 32 | 1, 217, 800 | 33, 981, 661 | 5, 715, 970 |
| 64 | 638, 552 | 183, 184, 321 | 34, 195, 280 |
| 128 | 3, 579, 043 | 1, 474, 287, 134 | 227, 212, 450 |
| 256 | 45, 548, 821 | 11, 081, 743, 208 | 1, 526, 415, 002 |
| 512 | 497, 280, 615 | 88, 354, 718, 583 | 10, 749, 479, 096 |
| 1024 | 2, 951, 782, 324 | 383, 522, 593, 970 | 38, 956, 976, 834 |

Classical still beat all the other algorithms but became noticeably slower as array size increased. Divide and conquer was by far the most inefficient with Strassen leading it from beginning. This is due to the fact that divide and conquer has one additional recursive call for each method call compared to Strassen which means more object creation, more operations, and slower run time. Strassen on the other hand performed much better to my surprise even in the early sizes. Having one less recursive call and some addition/subtraction operations which can be performed faster than multiplication had a great increase in performance compared to divide and conquer. One thing that I did notice is that as size increased, Strassen started catching up to the classical method. I think that if I ran it to a larger n (maybe with a faster computer) Strassen would eventually outperform the classical algorithm (as it should due to their time complexities). Unfortunately, I was only able to gather data up to $n = 1024$. This was due to the slowness of each algorithm as it takes a very long time running. Example output looks like:

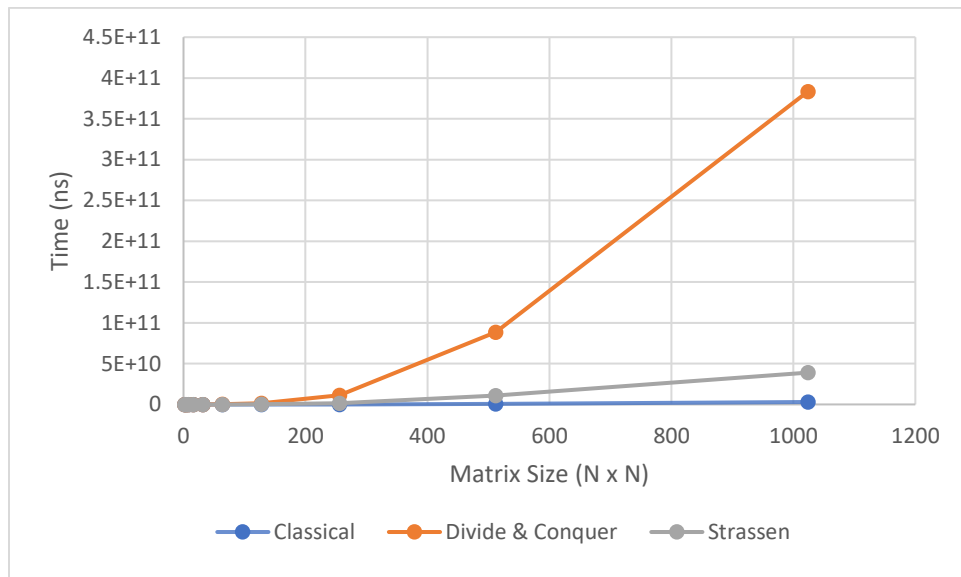
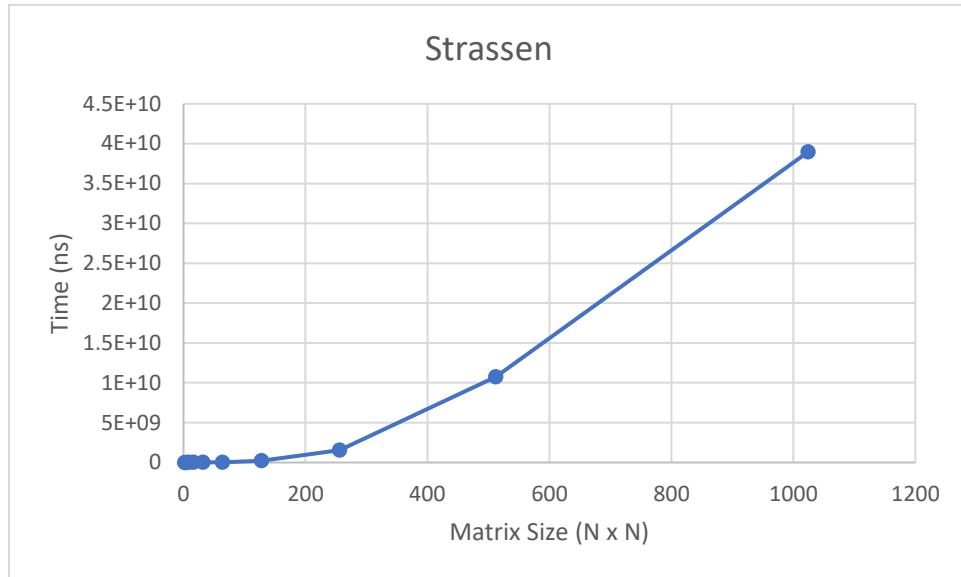
For $n = 2$:

Classic Matrix Multiplication time: 1991 nanoseconds.

Divide and Conquer Matrix Multiplication time: 12800 nanoseconds.

Strassen's Matrix Multiplication time: 1493 nanoseconds. ...

Classical and Divide and Conquer graph are in Classical Matrix Multiplication Vs Divide-and-Conquer Matrix Multiplication section.



Theoretical Complexity Comparisons:

Divide and Conquer matrix multiplication recurrence relation:

$$T(n) = 8T\left(\frac{n}{2}\right) + an^2$$

Apply the Master's Theorem: $a=8$, $b=2$, and $f(n) = \Theta(n^2)$

$$\text{Since } f(n) = O\left(n^{\log_b(a) - \epsilon}\right) = O\left(n^{\log_2 8 - \epsilon}\right)$$

$$T(n) = \Theta\left(n^{\log_b(a)}\right) = \Theta\left(n^{\log_2(8)}\right) = \boxed{O(n^3)}$$

Strassen's matrix multiplication recurrence relation:

$$T(n) = 7T\left(\frac{n}{2}\right) + an^2$$

Apply the Master's Theorem: $a=7$, $b=2$, and $f(n) = \Theta(n^2)$

$$\text{Since } f(n) = O\left(n^{\log_b(a) - \epsilon}\right) = O\left(n^{\log_2 7 - \epsilon}\right)$$

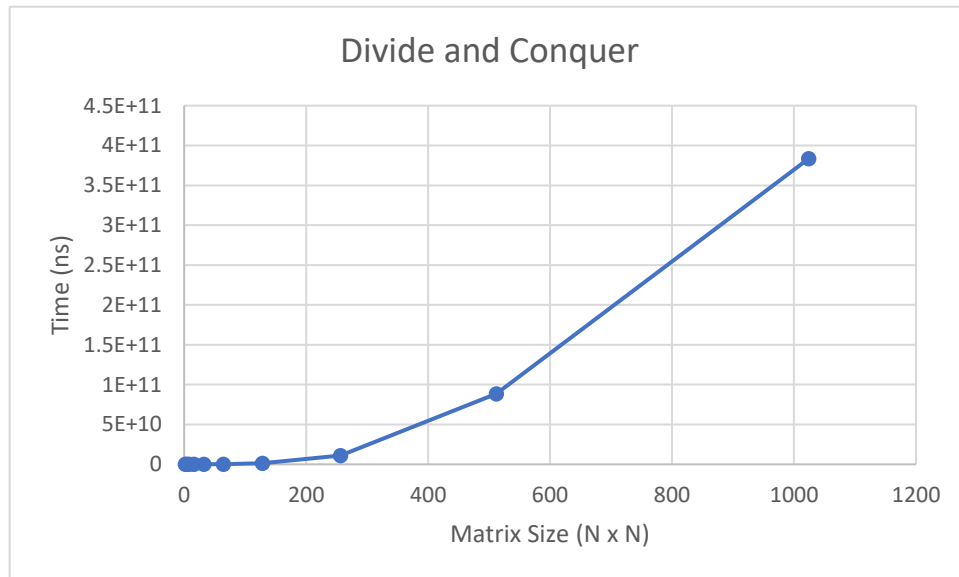
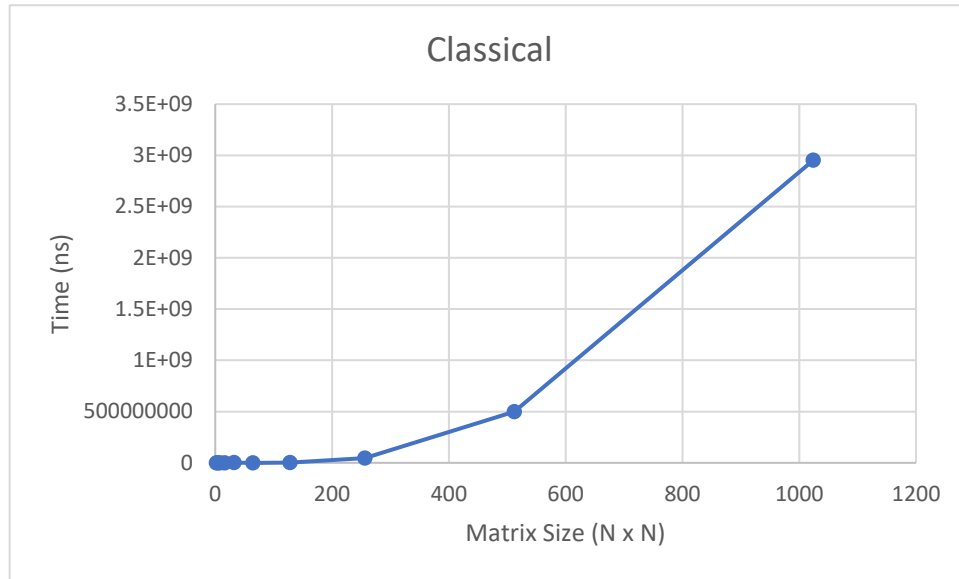
$$T(n) = \Theta\left(n^{\log_b(a)}\right) = \Theta\left(n^{\log_2(7)}\right) \approx \boxed{O(n^{2.81})}$$

Classical matrix multiplication is obviously $\boxed{O(n^3)}$ because it has three for loops inside.

Strassen's matrix multiplication is a clever approach to multiplying matrices. Although the frequency count of addition increases significantly, Strassen's method is still faster. The total implementation's running time is described by the recurrence relation: $T(N) = 7T\left(\frac{n}{2}\right) + O(n^2)$.

Solving the recurrence relation using back-substitution provided on the lecture slides or using the master's theorem provides the worst-case time complexity of $O(n^{\lg 7}) \approx O(n^{2.81})$.

Classical Matrix Multiplication Vs Divide-and-Conquer Matrix Multiplication:



Classic matrix multiplication is a very straightforward implementation using a total of three loops and having addition and multiplication operations inside yielding the worst-case time complexity of $O(n^3)$. Divide and Conquer matrix multiplication is another implementation that divides the matrix into sub-pieces and work with the smaller subsets to ultimately piece back the larger, original size answer. The total running time of the implementation is described by the recurrence relation: $T(N) = 8 T\left(\frac{n}{2}\right) + O(n^2)$. Solving the recurrence relation using master's theorem provides the worst-case time complexity of $O(n^3)$.


In our project, classical matrix multiplication is always faster (as shown in the data tables) simply because of few reasons. Both classical and divide and conquer approach has complexity of $O(n^3)$. So, now we need to consider space complexity. Divide and conquer is implemented using recursion meaning it uses stack and memory is in use. That in itself increases the time compared to the classical implementation because it's iterative, not recursive. Therefore, divide and conquer is slower than the classical method as described in the data table and graphs.

Strengths and Constraints:

My computer information:

Windows edition

Windows 10 Home
© 2018 Microsoft Corporation. All rights reserved.



System

| | |
|-------------------------|---|
| Manufacturer: | Acer |
| Model: | Swift SF314-52 |
| Processor: | Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz |
| Installed memory (RAM): | 8.00 GB (7.88 GB usable) |
| System type: | 64-bit Operating System, x64-based processor |
| Pen and Touch: | No Pen or Touch Input is available for this Display |



The project involves implementing and analyzing the three distinct methods of multiplying two matrices learned in class: classical, divide and conquer, and Strassen. I used the Java language and coded and tested the project in Eclipse software.

Overall, the program is running quite accurately and accomplishes all the aspects of the project well without any errors. Based on the data, I can depict the pattern of the three different methods and relate it back to the theoretical component of them respectively. It does not match perfectly because to be able to correlate one to one the experiment to theoretical, I would need more data of larger matrix sizes that are in thousands or even hundreds of thousands to clearly see that the Strassen is the fastest, then comes classical, and finally divide and conquer because of its recursion implementation. The best part of the project is the actual implementation of the three approaches. The implementation is very clear, easy to understand, and efficient. It very closely follows the pseudocode provided in the lecture notes for all the three approaches. If I were to do the project again, there isn't much to alter since the implementation aspect is quite efficient and perfectly follows the requirements. If I had more time, I would re-run my program on a desktop computer with perhaps a better way of generating my matrices. However, I would aspire and obtain the test results for matrix multiplication of all the three different approaches on matrices of very large sizes. I would need a computer that is excellent in speed and one where there are no concerns with memory usage. Generally speaking, the faster the computer, the better. The reason that I do not have matrices of sizes in thousands listed is simply because it takes a lot of processing power and memory to calculate them. Obtaining the highest size listed here already takes couple hours. Thus, I manually stop the program as per instruction mentioned in the class because of the fact that it takes a long time to compute.

Although the data does not appear to perfectly mirror the time complexity of each approach, it is accurate representation of the method and the reason for slight deviation from expected is due to the nature of matrix size. Better expected results would come with larger size that's not being reflected in the results.

Program Correctness:

To verify that the implementation of classical, divide and conquer, and Strassen is accurate, I coded a method called displayMatrix which prints a matrix passed in as a parameter. I verified that the matrix multiplication of all three methods is accurate by passing the resultant matrix and printing it. I apply the basic concept of matrix multiplication and verify that the resultant matrix is the product of two matrices multiplied. The general procedure for multiplying matrices is:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

For example, my matrices of size 2 x 2 are such:

$$A: \begin{bmatrix} 44 & 53 \\ 1 & 57 \end{bmatrix} \times B: \begin{bmatrix} 22 & 10 \\ 92 & 98 \end{bmatrix} = C \begin{bmatrix} 5844 & 5634 \\ 5266 & 5596 \end{bmatrix}$$

This is the same for classical, divide and conquer, and Strassen. Therefore, all the three methods are accurately implemented.

Meetkumar Patel - CS 3310.02 - Project 1

```
import java.util.Random;
/*
 * The purpose is to demonstrate matrix multiplication in three different ways:
 * Classical, Divide and Conquer, and Strassen's approach.
 */

public class MatrixMultiplication
{
    /*
     * Perform classical matrix multiplication using three loops.
     * @param A - A matrix to multiply
     * @param B - A matrix to multiply
     * @param n - the size of the matrix. Assumes no ragged one, it's a square.
     * @return a new matrix, C, that's result of A times B
     */
    public int[][] classicMatrixMultiplication(int[][] A, int[][] B, int n)
    {
        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++)
                {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }

        return C;
    }

    /*
     * Perform divide and conquer matrix multiplication.
     * @param A - A matrix to multiply
     * @param B - A matrix to multiply
     * @param n - the size of the matrix. Assumes no ragged one, it's a square.
     * @return a new matrix, C, that's result of A times B
     */
    public int[][] divideConquerMatrixMultiplication(int[][] A, int[][] B, int n)
    {
        int[][] C = new int[n][n];

        if (n == 1)
        {
            C[0][0] = A[0][0] * B[0][0];
            return C;
        }
        else
        {
            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];
```


Meetkumar Patel - CS 3310.02 - Project 1

```

        deconstructMatrix(A, A11, 0, 0);
        deconstructMatrix(A, A12, 0, n / 2);
        deconstructMatrix(A, A21, n / 2, 0);
        deconstructMatrix(A, A22, n / 2, n / 2);
        deconstructMatrix(B, B11, 0, 0);
        deconstructMatrix(B, B12, 0, n / 2);
        deconstructMatrix(B, B21, n / 2, 0);
        deconstructMatrix(B, B22, n / 2, n / 2);

        int[][] C11 = addMatrix(divideConquerMatrixMultiplication(A11, B11, n / 2),
                                divideConquerMatrixMultiplication(A12, B21, n / 2), n / 2);
        int[][] C12 = addMatrix(divideConquerMatrixMultiplication(A11, B12, n / 2),
                                divideConquerMatrixMultiplication(A12, B22, n / 2), n / 2);
        int[][] C21 = addMatrix(divideConquerMatrixMultiplication(A21, B11, n / 2),
                                divideConquerMatrixMultiplication(A22, B21, n / 2), n / 2);
        int[][] C22 = addMatrix(divideConquerMatrixMultiplication(A21, B12, n / 2),
                                divideConquerMatrixMultiplication(A22, B22, n / 2), n / 2);

        constructMatrix(C11, C, 0, 0);
        constructMatrix(C12, C, 0, n / 2);
        constructMatrix(C21, C, n / 2, 0);
        constructMatrix(C22, C, n / 2, n / 2);
    }

    return C;
}

/*
 * Perform Strassen's matrix multiplication.
 * @param A - A matrix to multiply
 * @param B - A matrix to multiply
 * @param n - the size of the matrix. Assumes no ragged one, it's a square.
 * @return a new matrix, C, that's result of A times B
 */
public int[][] strassenMatrixMultiplication(int[][] A, int[][] B, int n)
{
    int[][] C = new int[n][n];
    strassenHelper(A, B, C, n);

    return C;
}

/*
 * Actually perform the multiplication according to the Strassen's approach.
 * @param A - A matrix to multiply
 * @param B - A matrix to multiply
 * @param C - The matrix that is A x B
 * @param n - the size of the matrix. Assumes no ragged one, it's a square.
 */
private void strassenHelper(int[][] A, int[][] B, int[][] C, int n)
{
    if (n == 2)
    {
        C[0][0] = (A[0][0] * B[0][0]) + (A[0][1] * B[1][0]);
        C[0][1] = (A[0][0] * B[0][1]) + (A[0][1] * B[1][1]);
        C[1][0] = (A[1][0] * B[0][0]) + (A[1][1] * B[1][0]);
        C[1][1] = (A[1][0] * B[0][1]) + (A[1][1] * B[1][1]);
    }
    else

```

```

{
    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];

    int[][] P = new int[n / 2][n / 2];
    int[][] Q = new int[n / 2][n / 2];
    int[][] R = new int[n / 2][n / 2];
    int[][] S = new int[n / 2][n / 2];
    int[][] T = new int[n / 2][n / 2];
    int[][] U = new int[n / 2][n / 2];
    int[][] V = new int[n / 2][n / 2];

    deconstructMatrix(A, A11, 0, 0);
    deconstructMatrix(A, A12, 0, n / 2);
    deconstructMatrix(A, A21, n / 2, 0);
    deconstructMatrix(A, A22, n / 2, n / 2);
    deconstructMatrix(B, B11, 0, 0);
    deconstructMatrix(B, B12, 0, n / 2);
    deconstructMatrix(B, B21, n / 2, 0);
    deconstructMatrix(B, B22, n / 2, n / 2);

    strassenHelper(addMatrix(A11, A22, n/2), addMatrix(B11, B22, n/2), P, n/2);
    strassenHelper(addMatrix(A21, A22, n / 2), B11, Q, n / 2);
    strassenHelper(A11, subtractMatrix(B12, B22, n / 2), R, n / 2);
    strassenHelper(A22, subtractMatrix(B21, B11, n / 2), S, n / 2);
    strassenHelper(addMatrix(A11, A12, n / 2), B22, T, n / 2);
    strassenHelper(subtractMatrix(A21, A11, n/2), addMatrix(B11, B12,n/2), U,n/2);
    strassenHelper(subtractMatrix(A12, A22, n/2), addMatrix(B21, B22,n/2), V,n/2);

    int[][] C11 = addMatrix(subtractMatrix(addMatrix(P, S, P.length), T, T.length), V, V.length);
    int[][] C12 = addMatrix(R, T, R.length);
    int[][] C21 = addMatrix(Q, S, Q.length);
    int[][] C22 = addMatrix(subtractMatrix(addMatrix(P, R, P.length), Q, Q.length), U, U.length);

    constructMatrix(C11, C, 0, 0);
    constructMatrix(C12, C, 0, n / 2);
    constructMatrix(C21, C, n / 2, 0);
    constructMatrix(C22, C, n / 2, n / 2);
}

}

/*
 * Piece back a matrix provided the given matrix.
 * @param givenMatrix - the matrix to add to the newMatrix
 * @param newMatrix - the matrix that the givenMatrix adds itself to.
 * @param givenMatrixInitialRowPosition - begin from the row position
 * @param givenMatrixInitialColumnPosition - until the column position
 */
private void constructMatrix(int[][] givenMatrix, int[][] newMatrix,
    int givenMatrixInitialRowPosition, int givenMatrixInitialColumnPosition)

```

Meetkumar Patel - CS 3310.02 - Project 1

```
{
    int temp = givenMatrixInitialColumnPosition;
    for (int i = 0; i < givenMatrix.length; i++)
    {
        for (int j = 0; j < givenMatrix.length; j++)
        {
            newMatrix[givenMatrixInitialRowPosition][temp++] = givenMatrix[i][j];
        }
        temp = givenMatrixInitialColumnPosition;
        givenMatrixInitialRowPosition++;
    }
}

/*
 * Break the givenMatrix and fill up the newMatrix.
 * @param givenMatrix - the matrix to add to the newMatrix
 * @param newMatrix - the matrix that the givenMatrix adds itself to.
 * @param givenMatrixInitialRowPosition - begin from the row position
 * @param givenMatrixInitialColumnPosition - until the column position
 */
private void deconstructMatrix(int[][] givenMatrix, int[][] newMatrix,
                               int givenMatrixInitialRowPosition, int givenMatrixInitialColumnPosition)
{
    int temp = givenMatrixInitialColumnPosition;
    for (int i = 0; i < newMatrix.length; i++)
    {
        for (int j = 0; j < newMatrix.length; j++)
        {
            newMatrix[i][j] = givenMatrix[givenMatrixInitialRowPosition][temp++];
        }
        temp = givenMatrixInitialColumnPosition;
        givenMatrixInitialRowPosition++;
    }
}

/*
 * Add the two matrix: A and B
 * @param A - A matrix to add
 * @param B - A matrix to add
 * @param n - the size of the matrix. Assumes no ragged one, it's a square.
 * @return - A resultant matrix C of the operation
 */
private int[][] addMatrix(int[][] A, int[][] B, int n)
{
    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;
}
/*
```

Meetkumar Patel - CS 3310.02 - Project 1

```
* Subtract the two matrix: A and B
* @param A - A matrix to subtract
* @param B - A matrix to subtract
* @param n - the size of the matrix. Assumes no ragged one, it's a square.
* @return - A resultant matrix C of the operation
*/
private int[][] subtractMatrix(int[][] A, int[][] B, int n)
{
    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;
}

/*
 * Actually generates a new matrix with randomly generated ints from 0 to 100 of size n.
 * @param n - the size of the matrix. Assumes no ragged one, it's a square.
 * @return the matrix which is generated of n by n
 */
public int[][] generateMatrix(int size)
{
    Random integer = new Random();
    int[][] matrix = new int[size][size];

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            matrix[i][j] = integer.nextInt(100);
        }
    }

    return matrix;
}

/*
 * For testing purposes to display the any givenMatrix
 * @param givenMatrix - input a matrix to display
 * @param size - the size of the matrix
 */
public void displayMatrix(int[][] givenMatrix, int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            System.out.printf("%10d", givenMatrix[i][j]);
        }
        System.out.println();
    }
}
}
```

Meetkumar Patel - CS 3310.02 - Project 1

```
/*
 * To test the MatrixMultiplication class that consists of three different approaches.
 */
public class Test
{
    /*
     * The main program that runs an infinite loops until the computer crashes. The size of the
     * matrix that it generates is of power of 2 and is of size n x n. I am running each
     * algorithm 10 times to find the average time spent by each algorithm of distinct sizes.
     */
    public static void main(String[] args)
    {
        MatrixMultiplication MM = new MatrixMultiplication();

        final int REPEAT_TIMES = 10;
        int size;
        long startTime, endTime;
        long totalTimeClassic = 0, totalTimeDivideConquer = 0, totalTimeStrassen = 0;
        int[][] matrixA, matrixB;

        for (int i = 1; i > 0; i++)
        {
            size = (int) Math.pow(2, i);

            matrixA = MM.generateMatrix(size);
            matrixB = MM.generateMatrix(size);

            for (int j = 0; j < REPEAT_TIMES; j++)
            {
                startTime = System.nanoTime();
                MM.classicMatrixMultiplication(matrixA, matrixB, matrixA.length);
                endTime = System.nanoTime();
                totalTimeClassic += endTime - startTime;

                startTime = System.nanoTime();
                MM.divideConquerMatrixMultiplication(matrixA, matrixB, matrixA.length);
                endTime = System.nanoTime();
                totalTimeDivideConquer += endTime - startTime;

                startTime = System.nanoTime();
                MM.strassenMatrixMultiplication(matrixA, matrixB, matrixA.length);
                endTime = System.nanoTime();
                totalTimeStrassen += endTime - startTime;
            }

            totalTimeClassic = totalTimeClassic / REPEAT_TIMES;
            totalTimeDivideConquer = totalTimeDivideConquer / REPEAT_TIMES;
            totalTimeStrassen = totalTimeStrassen / REPEAT_TIMES;

            System.out.println("For n = " + size + ": " +
                "\n\tClassic Matrix Multiplication time: " + totalTimeClassic + " nanoseconds." +
                "\n\tDivide and Conquer Matrix Multiplication time: " + totalTimeDivideConquer + " nanoseconds." +
                "\n\tStrassen's Matrix Multiplication time: " + totalTimeStrassen + " nanoseconds.\n");
        }
    }
}
```